Using Tracing to Enhance Data Cache Performance in CPUs

The creation of a Trace-Assisted Cache to increase cache hits and decrease runtime

Jonathan Paul Rainer

A thesis presented for the degree of
Doctor of Philosophy (PhD)

Computer Science
University of York
September 2020

Abstract

The processor-memory gap is widening every year with no prospect of reprieve. More and more latency is being added to program runtimes as memory cannot satisfy the demands of CPUs quickly enough. In the past, this has been alleviated through caches of increasing complexity or techniques like prefetching, to give the illusion of faster memory. However, these techniques have drawbacks because they are reactive or rely on incomplete information. In general, this leads to large amounts of latency in programs due to processor stalls. It is our contention that through tracing a program's data accesses and feeding this information back to the cache, overall program runtime can be reduced. This is achieved through a new piece of hardware called a Trace-Assisted Cache (TAC). This uses traces to gain foreknowledge of the memory requests the processor is likely to make, allowing them to be actioned before the processor requests the data, overlapping memory and computation instructions.

Comparing the TAC against a standard CPU without a cache, we see improvements in runtimes of up to 65%. However, we see degraded performance of around 8% on average when compared to Set-Associative and Direct-Mapped caches. This is because improvements are swamped by high overheads and synchronisation times between components. We also see that benchmarks that exhibit several qualities: a balance of computation and memory instructions and keeping data well spread out in memory fare better using TAC than other benchmarks on the same hardware. Overall this demonstrates that whilst there is potential to reduce runtime via increasing the agency of the cache through Trace Assistance, it requires a highly efficient implementation to be competitive otherwise any potential gains are negated by the increase in overheads.

Contents

List of Tables

List of Figures

List of Listings

Research Data

All of the research data, including automation and hardware code, generated in this project can be found in the following locations:

Software

- Kuuga - The Source Code for each of the hardware variants used to generate the runtime data seen within this thesis - [172] - Jonathan Rainer. *Jonathanrainer/Kuuga: Initial Release*. Zenodo. Sept. 2020. DOI: `10.5281/ZENODO.4045227`
- Ichijou - The Source Code for the automation framework that takes a C benchmark file, generates the required hardware and outputs the results - [171] - Jonathan Rainer. *Jonathanrainer/Ichijou: Initial Release*. Zenodo. Sept. 2020. DOI: `10.5281/ZENODO.4045225`

- Sawatari - The Source Code for the visualisation tool that produces the Memory Activity Diagrams seen in Chapter 6 - [173] - Jonathan Rainer. *Jonathanrainer/Sawatari: Initial Release*. Zenodo. Sept. 2020. DOI: 10.5281/ZENODO.4045229

Data

- Runtime Data - [170] - Jonathan Rainer. *Experimental Data (Including Hardware Variants) Measuring Runtime for Trace Assisted Caching*. en. Sept. 2020. DOI: 10.5281/ZENODO.4040337
- Memory Activity Data - [169] - Jonathan Rainer. *Experimental Data (Including Hardware Variants) Measuring Memory Activity for Trace Assisted Caching*. en. Sept. 2020. DOI: 10.5281/ZENODO.4042892

Acknowledgements

As I come to write this final section of my PhD I think back over the last 4 years with a great deal of fondness. Throughout those years I've met and received help from so many people, that to thank them all by name would have doubled the size of this thesis! So now as I write these acknowledgements I apologise if I've missed anyone out. Please know that even if you don't appear by name in what follows, if you have offered me so much as a word or kind gesture during the last four years you have my eternal gratitude.

My first thanks have to go to my supervisor Professor Neil Audsley and my internal examiner Professor Alan Burns. Their sage guidance and advice on the construction of this thesis has brought forth something that I would never have thought possible when I began in 2016. I'd also like to thank my external examiner Professor Geoff Merrett for undertaking this role in unusual circumstances due to the Coronavirus Pandemic. Further to that I'd like to thank the Engineering and Physical Sciences Research Council (1796038) for the funding of the first three years of my PhD, without which my PhD would not have been possible.

I'd also like to extend my warmest thanks to the Department of Computer Science at York, and every student within it. Since I arrived in the department in 2011, as a young student who'd never programmed before, the department has nurtured me and taught me so much. Specifically I'd like to thank Dr. Ian Gray and Dr. Russell Joyce for not only being the most helpful whenever I had problems with the Xilinx tools; but also for providing endless hours of discussion, when I probably should have been writing my thesis. My thanks also extend to all the administrative staff in the Department of Computer Science, and specifically Debra Lashua, Jo Maltby and Claire Fox. All of you encouraged me in so many aspects of my PhD and helped me navigate the administrative minutiae that so often bedevils and waylays us.

During my PhD I had the opportunity to teach a whole range of modules in a multitude of capacities, so I'd like to thank Dr. Mike Freeman, Dr. Steve King, Dr. Lillian Blot, Dr. Anna Bramwell-Dicks and all the other academics with whom I had the pleasure to teach. Your commitment to your students is exemplary and some of my favourite memories of my PhD were teaching in your classes. I'd also like to thank Jane Dalton and Karen Clegg for running the York Learning and Teaching Award (YLTA) while I was working on my PhD. In completing the award I learned more than I ever thought I would about how to construct learning experiences and become a better teacher, something that I've taken with me into everything I've done since.

Of course writing a PhD doesn't just require academic support but support from our friends and family. As such I'd like to thank Richard Sharp, Dr. Jamie Wood, Dr. Rob Alexander, Dr. Greg Reynolds, Dr. Sam Simpson, Jo Maltby and Nicola Peard for being such wonderful friends throughout the last 4 years. I've lost count of the number of times a bad day of experiments was made so much better by going on adventures with my friends.

For the final year of my PhD I moved into a new phase of life, working full-time for Anaplan. I would like to thank the company as a whole for being so supportive of my research, but would also like to thank the individuals within the company for being so welcoming. My particular thanks go to Louis Rose, Graeme Neath, Cliff Evans, Katherine Carvey, Chloe Taylor, Ben Speight and Jason Reich, for providing me with such a warm welcome and for further supporting me in the writing of this thesis.

My final and biggest thanks has to go to my wife Lauren and my parents, John and Julie Rainer. I got married to Lauren in 2018, almost exactly half way through my PhD and she could not have been more supportive. Even through the trials and adversity of a career-shift, and everything else life has thrown at her over the last four years, I've never doubted her love and support for all I do. To my parents also, who first encouraged me when I wanted to take computers apart with a screwdriver, thank you so much for everything, I would not be here without you.

> *There are worlds out there where the sky is burning, where the sea's asleep, and the rivers dream. People made of smoke, and cities made of song. Somewhere there's danger, somewhere there's injustice, and somewhere else the tea's getting cold. Come on, Ace, we've got work to do!* - The Doctor - Survival - Episode 3

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Part I

Introduction

1 Introduction

This chapter outlines the motivation behind the programme of research into TACs and lays out the aims and research questions. A structure of the thesis is also presented.

1.1 Motivation

At its most basic, a computer consists of a CPU, a store of memory and an I/O system to facilitate the movement of data between the first two parts. The operation of a computer is then to repeatedly fetch an instruction from memory (along with any operands the instruction might refer to), execute the instruction and then possibly write back any results to memory [162]. Main memory therefore plays an integral role in the operation of a computer, and so the memory access time[1] is one of the key factors that dictates the overall execution time of a program [83, 154].

With that in mind, the aim when designing memory systems is to try and keep access times comparable to CPU clock speeds. So that when an instruction is issued, the data is available immediately without having to stall the whole system. However, this can become problematic very quickly. Static Random Access Memory (SRAM), the fastest type of off-chip memory available commercially, provides access times of 0.5 - 2.5ns but costs somewhere in the region of £400 - 800 per GB. By comparison, Dynamic Random Access Memory (DRAM) memory is substantially cheaper, at between £8 - £16 per GB but only boasts access times of 50 - 70ns [83], a difference of two orders of magnitude in price and access time. In addition, for the same amount of silicon more DRAM than SRAM can be fabricated. As a result, having a very large store of very fast access memory is impossible if you want it to be economically viable.

Moreover, since the invention of the first computing machines, there has been a gap between the speed at which a processor can issue instructions and the speed at which memory can satisfy them [227]. This has become known as the 'processor-memory gap', with processors improving in speed exponentially thanks to Moore's Law, but

[1]The time between issuing a request to memory and getting a valid response.

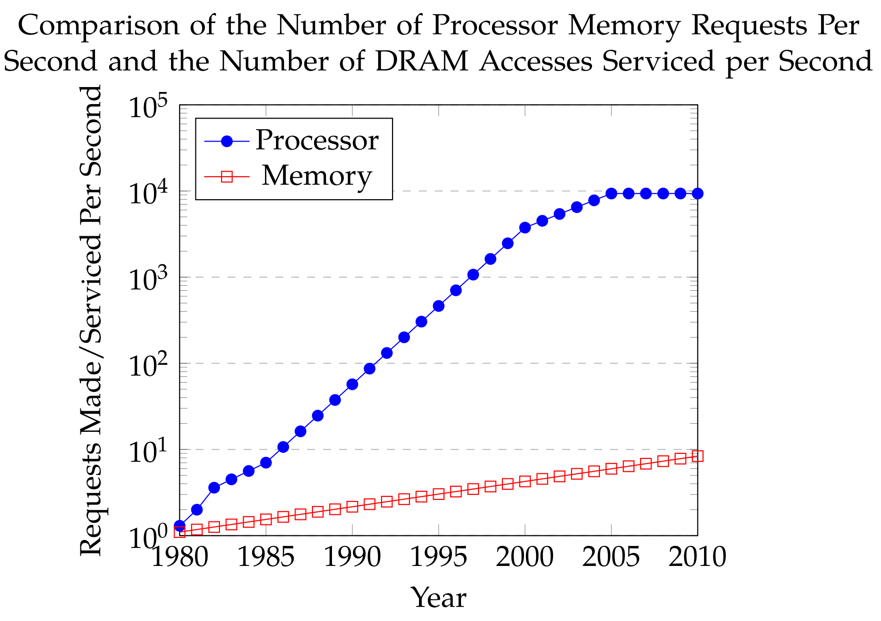


Figure 1.1: This graph (taken from Hennessy and Patterson [82]) shows the number of requests to memory made by processors (on average) against the number of DRAM accesses per second. In an ideal situation the number of requests memory could satisfy per second would be above the number of requests the processor could issue per second (the red line would be above the blue line). But as can be seen that is not the case, and up until 2005 this gap was ever widening. The levelling off in processor performance is mostly due to the move to multi-core architectures (and the corresponding drop in the development of ever faster single cores), and consequently memory has somewhat caught up between 2005 and 2010. However it's worth emphasising that the the scale used is logarithmic, consequently the gap between the performance of memory and processors is still 3 orders of magnitude. At the current rates it would take over 100 years to close the gap, assuming that processors don't further increase in performance between now and then, showing that finding mechanisms to alleviate this large performance gap are still important today. Further details of this graph can be seen in Hennessy and Patterson [82].

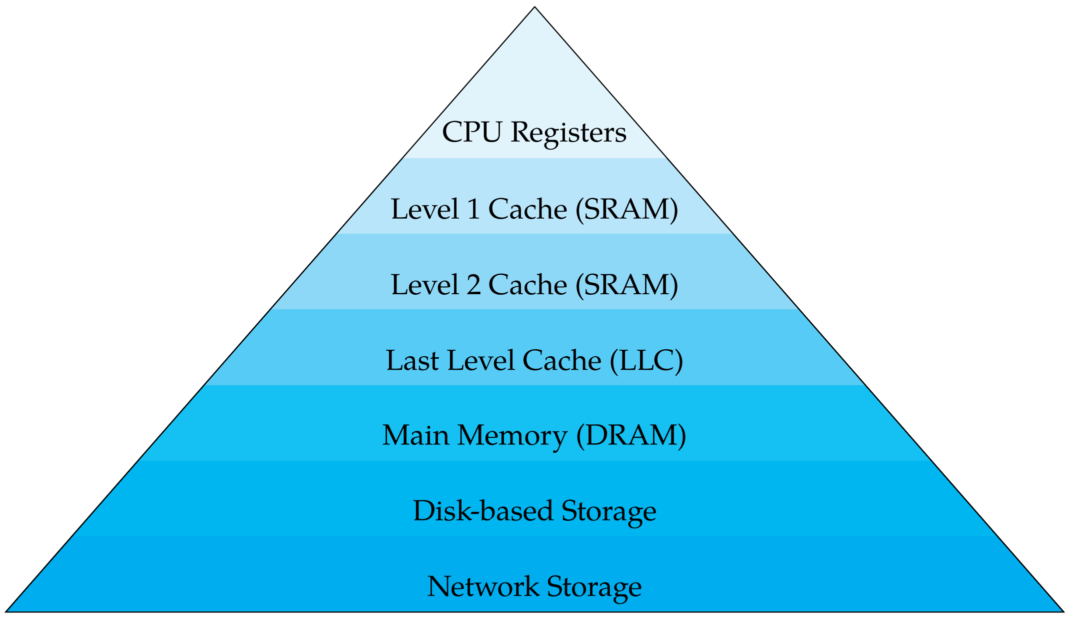


Figure 1.2: The pyramid shows an example memory hierarchy for a desktop computer where height is proportional to cost and inversely proportional to speed of access. The decreasing size of each pyramid step also shows the capacity of each type of memory present in the system.

memory technologies increasing at a fraction of that rate, as Figure 1.1 shows. This is a problem because the abstraction presented to programmers abstracts away time, making it appear that each statement or instruction takes an equal amount of time to execute. Therefore, a choice has to be made either to remove the abstraction and include time as a concern for programmers or to hide the latency and restore the abstraction in the vast majority of cases. Since most programming languages and tools are unsuited to modelling time [67], the second approach is most often taken, with caches and memory hierarchies being the most common vehicle by which this is achieved.

1.1.1 Caches & Memory Hierarchies

Memory hierarchies, first seen in super-computers in the 1960's [83], structure main memory as a hierarchy of different memory technologies [226] (see Figure 1.2). When this is combined with caching it gives the illusion of very fast memory accesses [194]. Caching works by setting aside a small portion of very fast memory to contain a copy of some of the elements of main memory. When a processor requests a piece of data, if the data is already in the cache, the request can be dealt with almost instantly leading to very low memory latency[2]. Alternatively, if the data is not in the cache

[2]This is known as a cache hit.

then the cache communicates with main memory to get the required data and either fills an empty space in the cache or evicts data currently occupying a slot in the cache[3]. Caches move data in and out based on their replacement policy and this can vary from a very simple First-In-First-Out (FIFO) policy, to something much more complicated [238]. If the cache replacement policy is designed well, then in the majority of cases any data requested by the CPU will be in the cache when it is requested. This leads to a drop in average memory access time and then the 'processor-memory gap' is alleviated. This reduces costs because caches are very small by comparison to main memory but if deployed correctly can give the illusion that main memory is as fast as something much more expensive.

Nevertheless, caching does introduce a number of problems. Some cache misses are inevitable as the cache becomes full or because the cache is initially empty, but these still significantly impact performance [238]. This is exacerbated in modern processors due to the use of deep pipelines to increase instruction-level-parallelism. In pipelined architectures a stall in a memory operation can cause a subset (or in the worst case every) instruction in the pipeline to be stalled, meaning that cache misses not only impact the instruction they occur as part of, but surrounding instructions as well [83]. At the same time, memory operations take up between 30% and 40% of the instructions in a typical program [24, 116], which leaves the majority of instructions not requiring a reference to external memory (apart from the initial fetch). This means there are many occasions where the memory bus is idle between satisfying memory operations, known as slack time. This has been recognised by Out-of-Order (OoO) processors for a long time. It is addressed by using static analysis and compile time techniques to allow instructions to be re-ordered without changing the semantics of the program [82], for example memory `LOAD` operations may be scheduled while the processor is doing computation. However, this technique is limited, because those techniques reveal nothing about the dynamic behaviour of a program, meaning that not all the slack can be utilised [225].

1.1.2 Predicting Dynamic Behaviour

A natural question to ask is whether we could exploit the slack that exists within programs without the effort of changing the order their instructions execute in? The problem is that no processor is clairvoyant. In other words, a processor is only 'conscious' of the instruction it is executing at the current time. It has no concept of a state before or after the execution of the current instruction and simply keeps applying the fetch, decode, execute cycle until the power is turned off. As a

[3]This is known as a cache miss.

consequence there needs to be some 'intelligence' applied somewhere else in the process, as changing the way the processor works, at such a deep level, would be very challenging.

To get round this problem, authors have implemented techniques like cache preloading [146], which loads the cache with data before the first instruction has been executed. Others have used prefetching [69]. Here, when a cache miss happens, not only is the requested data fetched, but a selection of other data too. The problem with these approaches is that they are ill-suited to caching data, as opposed to instructions, which is where this thesis focuses. Prefetching and preloading both work very well for instruction data due to its characteristics [113, 194]. However, both of these methods lack a way of feeding back information from the program itself to inform the choice of data to preload. Rather, they rely on heuristics or other methods, and even if they do offer a feedback mechanism is is often limited in its applications (i.e. compiler-directed prefetching relying on loop structures to work effectively).

1.1.3 Tracing & Trace Assisted Caching

Tracing is the act of recording every action taken by a processor in the execution of a program. It is a technique often used for debugging and profiling applications and the traces are often recorded during execution and then examined offline due to their size and complexity. However, in recent times, many commercial tracing solutions have started to appear [41, 140], with very condensed trace formats and fast hardware buffers. This innovation has developed to such a point that it would almost be possible to consider trace data online, as the processor produced it. However, very little research has been done in this area [159], as many commercial solutions do not offer the level of introspection that would be required.

However, with many new CPU designs being released under the OpenHardware [144] umbrella it is now possible to see how complex processors manipulate data and track various events that occur within the processor without breaking any rules around intellectual property. Consequently the following question must be asked: if we could take the information we can measure from a trace and feed it back to a more intelligent cache, would it be able to make better decisions about which pieces of data to keep in the cache and which to evict?

Suppose we have a CPU with a Harvard Architecture and a piece of hardware that can accurately track all accesses to data memory made by this CPU. If this information could be stored and communicated to a cache, it should be possible to base the cache replacement policy on this data. The cache could then act in a predictive fashion, executing `LOAD`s before they are required and dealing with write-backs for `STORE`s.

Therefore, there should never need to be a cache miss in the ideal case. This approach would better utilise the available slack time within a program and lead to reductions in the execution time of programs. I have dubbed this approach 'Trace Assisted Caching' and the development and implementation of this technique is detailed in forthcoming chapters.

1.2 Thesis Aims

This thesis presents evidence to explore the following research questions:

1. Can feeding trace information back to a cache be used to decrease the runtime of programs?
2. Can this approach outperform a processor running the same computation but with a standard cache?
3. Does this approach introduce any overheads when compared to a processor with a standard cache?
4. Under what circumstances does the addition of trace information give the maximal benefit?

In order to answer these questions a hardware platform will be constructed that implements a TAC. We can then measure the runtime of programs in this hardware platform as compared to a platform with only a processor. This should give us insight in to the first of the four questions, then extending this investigation to a processor with a standard cache will give us insight into the second and third questions. At that point it should be possible to extract particularly positive or negative cases and the commonalities between these should give insight into the final question.

All these questions are going to be explored in the context of embedded systems that contain caches of some variety. We are not interested in systems that have direct access to memory because these are often very simple systems that do not require complex caching. In addition, because of the context, when considering resource usage we are not considering energy usage as a resource. Of course, systems of this size often have to concern themselves with energy usage, as it is often limited, but we are not considering trying to design a more energy efficient cache, rather one that is more performant from a runtime perspective. There is an expectation that we will have to sacrifice some extra on chip resources (i.e. space) to make this a reality; but it's hoped that the trade-off will be commensurate with an increase in performance, and this will be tracked explicitly by the answers to the third research question. That being said there is a case to be made that a design of this nature may well be applicable to High Performance Computing (HPC) as well but this comes with a different set of assumptions around number of cores and caching architecture and consequently that will not be addressed in this thesis. This thesis focuses on medium-sized embedded systems that contain caching to some degree, even if those caches are relatively simple.

1.3 Thesis Structure

With these questions and context in mind the rest of the thesis is structured as follows:

- Chapter 2 introduces the background literature for this topic and explains the gaps that necessitate this research direction.
- Chapter 3 presents a more thorough explanation of the Trace Assisted Caching technique.
- Chapter 4 explains the implementation of the Trace-Assisted Caches.
- Chapter 5 discusses the experiments conducted to measure the performance of the TAC and presents the results.
- Chapter 6 analyses the results in detail and provides more insight into the performance observed in the experiments.
- Chapter 7 summarises the work conducted, and its contribution, and then sets out ideas for future work to extend the effectiveness of the TAC.

Part II

Background

2 Literature Review

This chapter presents a review of the literature surrounding memory latency reduction. It first focuses on more traditional methods, considering caching and then other methods of controlling latency, such as prefetching or scheduling. In the second section the focus moves to newer techniques that involve tracing, and then finishes with a summary and evaluation of the position the literature presents.

2.1 Introduction

Memory hierarchies are necessitated by the inverse correlation of memory speed and memory cost. This means that, in general, faster memory is more expensive while slower memory is cheaper. But that does lead to the question of exactly how much slower each of the levels of the memory hierarchy actually are. This is illustrated in Figure 2.1 with an example of a mobile device with just one level of caching. As we can see, as we go down the memory hierarchy the time to access the memory increases but not at a uniform rate. For example the difference in speed between the the L1 and L2 cache is an order of magnitude but between main memory and storage is 3 orders of magnitude. Of course this is one simple example of a memory hierarchy, in more complex systems you may have up to 3 or more levels of caching and potentially multiple types of persistent storage, as shown in Figure 2.2.

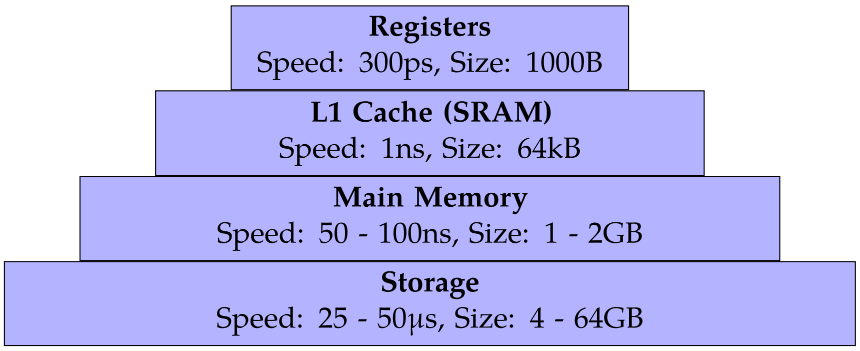

Registers
Speed: 300ps, Size: 1000B
L1 Cache (SRAM)
Speed: 1ns, Size: 64kB
Main Memory
Speed: 50 - 100ns, Size: 1 - 2GB
Storage
Speed: 25 - 50μs, Size: 4 - 64GB

Figure 2.1: A memory hierarchy for a simple embedded system with one level of caching. This could be likely be something be a smartphone. This is adapted from data presented in Hennessy and Patterson [82].

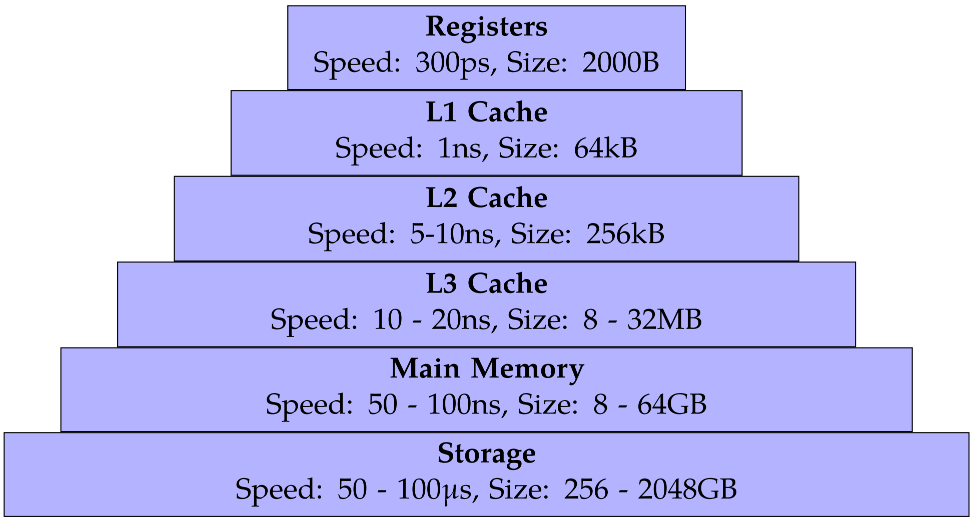

Registers
Speed: 300ps, Size: 2000B
L1 Cache
Speed: 1ns, Size: 64kB
L2 Cache
Speed: 5-10ns, Size: 256kB
L3 Cache
Speed: 10 - 20ns, Size: 8 - 32MB
Main Memory
Speed: 50 - 100ns, Size: 8 - 64GB
Storage
Speed: 50 - 100µs, Size: 256 - 2048GB

Figure 2.2: A memory hierarchy for a more complex desktop system used to run multiple workloads at once. This is adapted from data presented in Hennessy and Patterson [82].

The figures presented in Figures 2.1 and 2.2 are of course only guides. Other factors too can influence the exact amount of latency that each portion of the memory hierarchy experiences. For example the state of the connection between different levels of the hierarchy is very important, as is the physical location of the memory source. Not only that but there are also design choices around memory hierarchies, such as the relative size of each component. Some processors choose to prioritise a very large L1 cache to keep data as close to the processor's registers as possible, while others choose to increase the size of the LLC to try and keep more on chip for longer as in general off or cross-chip communication is much slower than on-chip communication. Not only that but the size of the memory itself also influences the speed of access. As memory size increases the amount of address decoding logic also increases so larger memory at certain levels of the hierarchy can increase access time as well. All in all the construction of a memory hierarchy is a complex and multi-faceted optimisation problem that does not have a simple answer. That being said this thesis focuses on how we can use the existing resources to their best effect so we will not be focusing explicitly on trying to solve this optimisation problem. However it's important to identify the major sources of latency that exist in this hierarchy, so that we can then find targeted solutions to resolve the problems they introduce.

By way of circumscribing the discussion, in this review we will not consider the implementation of memory hardware, despite it being the largest source of latency. This is because, as Hennessy and Patterson [83] describes, choosing a memory technology is a trade-off between speed and cost, with the general principle being that faster memory is more expensive per unit. In my review of the literature I did not find any papers proposing to have resolved this trade-off. All propose improvements to existing technologies or new ones that fit into the existing cost/speed model. If we want to improve overall memory latency the most efficient approach is to reduce the amount of times that main memory is accessed as this will have more of an impact, if it can be reduced enough, than piecemeal improvements to memory access speed.

Further to this point, it is appreciated that there are an enormous amount of factors that can impact latency and overall program runtime. These include, but are not limited to, the compiler chosen for the code, the structure of the programs, the memory layout decided by the linker, the programming language and so on. In addition a lot of these factors are linked together, for example the choice of compiler can produce code that is more or less amenable to the hardware it runs on, i.e. using a vendor specific compiler as opposed to a general purpose one. This is possible because the compiler has knowledge of the internal structure of the processor that the general purpose one does not. As a consequence of the large number of potential factors we will focus on the cache itself and the environment within which it operates, acknowledging the interlinked factors as we go but avoiding in depth discussion of every factor that could potentially influence latency and program runtime. Finally it's worth re-iterating that our context is medium-sized embedded systems, similar to those presented in Figure 2.1 above, consequently even though we might consider work that implements web caches or Operating System (OS) caches we're looking at them in the context of how the ideas might be applied to embedded systems, rather than attempting to suggest that ideas used at one scale of caching are instantly applicable to another.

Due to this literature review covering a broad range of topics, most in considerable depth, the diagram presented in Figure 2.3 maps out the various topics and their relationship to each other. Reading the review in concert with this diagram can help to locate yourself within the research presented and see the relationships between the topics included.

2.2 Cache Intrinsic Techniques

Since their first introduction to super-computers in the 1960's [83] caches have become a standard part of almost any memory architecture. Iterative improvements, thanks to Moore's law, have pushed the performance of caches to higher and higher levels, hiding more and more latency as their information density has increased. Despite advances in technology, it is still the case that the cache replacement policy is the most

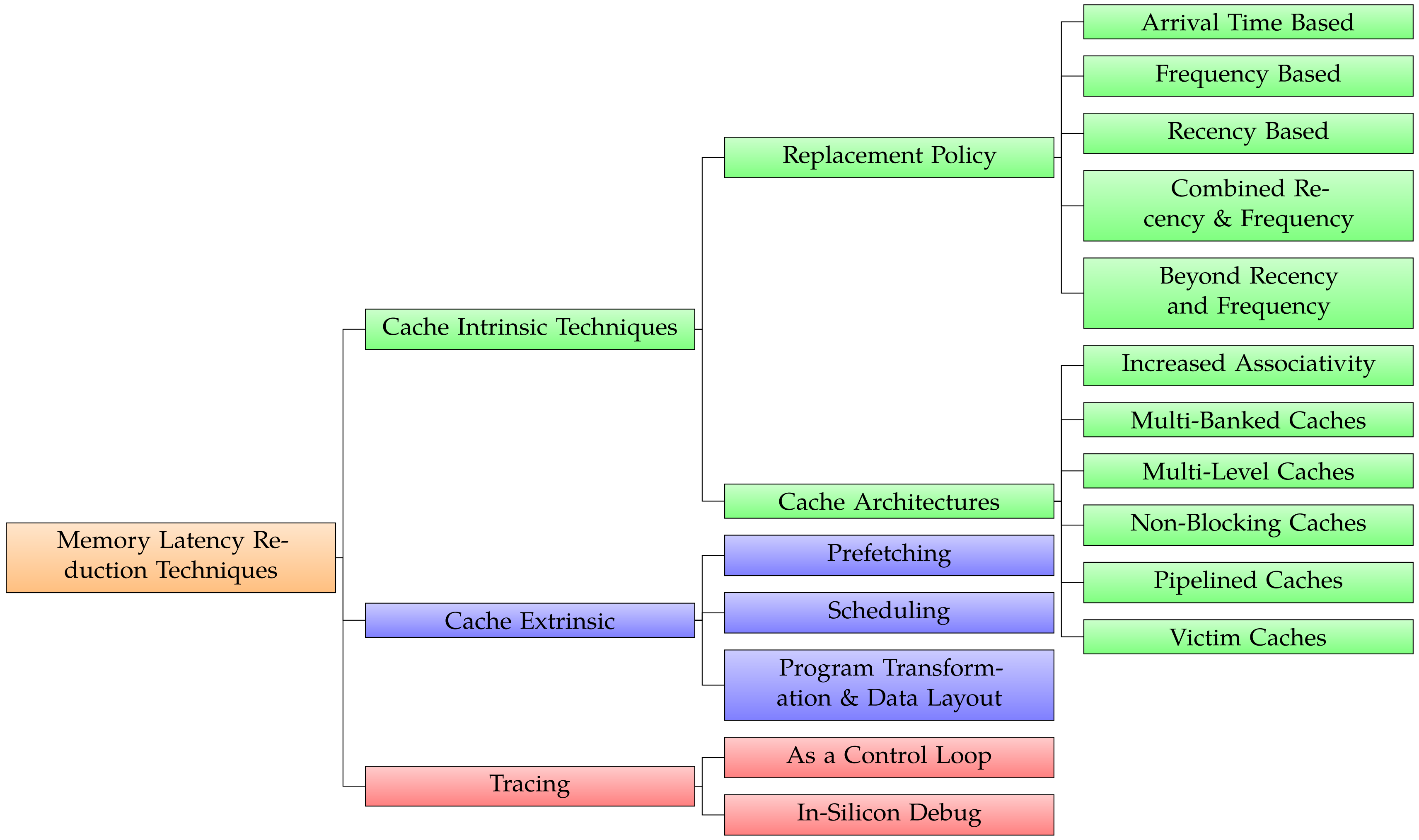


Figure 2.3: The structure of the Literature Review showing the topics and subtopics covered, organised into sections. The review is conducted via a depth-first search of these topics.

significant factor in determining how effective a cache will be at reducing memory latency [82].

2.2.1 Cache Replacement Policy

In its simplest form a cache replacement policy is a way of deciding which cache line is replaced when a cache reaches capacity. For Direct-Mapped caches they are not important, as the structure of the cache dictates the policy, but in Set-Associative caches the choice of replacement policy is absolutely crucial [82]. As a consequence there has been much research into which policies yield the best outcomes across a variety of metrics.[1]

However, before we begin discussing these policies in depth it is useful to have in mind two policies that are often referred to in the literature, but are not actually implemented. The first of these is known as `OPT` [91, 127] a theoretical optimal replacement policy, which can perfectly predict which cache block will be needed furthest in the future. This means `OPT` will provably suffer the lowest number of cache misses of any cache policy [127]. The second theoretical replacement policy is that of random replacement or `RAND`. Under this scheme when a decision on replacement has to be made, the choice is made completely at random without reference to any other information [21]. Again this method is not usually implemented by cache designers when optimising for latency reduction [99], but its utility lies as the lower extreme of a continuum, bounded by `OPT` at the upper extreme. This continuum allows policy designers to empirically assess new policies in relation to `OPT` and `RAND`, as well as against similar replacement policies.

Arrival-Time Based Techniques

One of the more simple cache replacement policies is to decide which cache line is to be replaced based on when the cache line entered the cache. This is known as the cache line's arrival time. The most common formulation of this is a policy that removes the item inserted furthest back in the past, implemented using a FIFO queue. This technique is often chosen because it has a very low hardware requirement [151], which leads to a low cost. On the other hand it does not perform well at reducing latency [208, 238], often performing similarly to `RAND`, despite the slight increase in hardware. FIFO is also susceptible to Belady's Anomaly [22], which

[1]In this discussion of cache policies reducing miss rate and reducing latency can be thought of as synonymous, due to cache misses being the main source of latency. For example, as we saw in Figure 2.1 and 2.2 the difference between accessing data stored in the cache and in main memory can be up to 2 orders of magnitude (1ns for the L1 cache as compared to 100ns for main memory). As a result the more cache misses can be converted to cache hits the more the latency will be reduced.

means there is no guarantee that larger caches will produce lower miss rates. Due to the low performance, research into using FIFO queues has been limited, with FIFO often being used a baseline [63] rather than being implemented in its own right.

However, some work has been done to improve FIFO. Turner and Levy [210] develops the idea into Segemented First-In-First-Out (SFIFO), which partitions main memory into two sections. This creates a pseudo multi-level cache (see Section 2.2.2), but at a lower hardware cost. In the same vein, Deville [55] augments FIFO with a usage counter per set, to do the partitioning in a more granular way. In more recent times Wei-Che Tseng et al. [223] have experimented with combining a FIFO policy and cache-line locking. All these policies perform comparably to Least Recently Used (LRU) (see subsection on Recency Based Techniques) and use less hardware. Overall, FIFO is a good baseline to build from and a viable option to implement if resources are limited. However, we can use more information to make better decisions if we consider frequency of access.

Frequency Based Techniques

A slightly more sophisticated approach to cache replacement is counting the number of times a cache block or line has been accessed and then evicting the one with the lowest frequency of access. This approach is known as Least Frequently Used (LFU). In terms of implementation, the most common form is to turn the cache into a priority queue, where keys are calculated according to a variety of formulae [158]. In addition, implementations choose between perfect LFU, where every object is uniquely tracked across replacements, and in-cache LFU, where counts are only tracked when items are in the cache, with the latter option being most common [158].

Despite its simplicity of concept, standard LFU has several problems. The first is cache pollution [99], where a cache block has a high number of accesses very early on, but is never referenced again. Having built up a high frequency count, the block stays in the cache for a long time, effectively reducing its capacity. The second problem is that often many different cache blocks have the same frequency count and therefore require tie-breaking arbitration [158]. Moreover, the hardware to keep track of all frequency counts, potentially across multiple replacements, gives a very high hardware overhead and increased energy consumption [151].

To be specific, LFU requires $N * \log_2 X$ bits per cache set, where N is the number of elements per set and X is the accuracy of the counter you require, as well as the circuitry required to implement a priority queue [151]. This is compared to LRU which only requires $N * \log_2 N$ bits [151] and the circuity required to implement a linked list, which is much less as all that is required are some pointers and the data, as opposed to the self organising structure of a priority queue. Not only that but comparing the two equations it becomes obvious that the difference comes down to the resolution of the counter X. It might be tempting to think that setting $X = N$ would suffice and then there would be parity of cost (excluding the cost of the priority queue). However, this is not optimal because the required counter resolution is intimately related to the dynamics of the program itself. To explain, if you choose N so that the number of bits is matched between LRU and LFU but the program references some memory items 100s of times, LFU's utility is severely diminished. To extend this example, imagine a 128 line cache, split into 16 sets. Using LRU this would consume an extra $8 * \log_2(8) = 24$ bits, but if we make the assumption we require counter resolutions in the 100s, LFU would require $8 * \log_2(128) = 8 * 7 = 56$ extra bits. This demonstrates how much extra resource could be consumed by LFU to give the same utility as LRU.

Some work has attempted to address the shortcomings of LFU with a variety of augmentations, most of which relate to adding ageing parameters to counter cache pollution. Arlitt et al. [14] suggests calculating the keys (K_i) in the priority queue that powers LFU using the formula $K_i = C_i * F_i + L$. In this formulation C_i is the cost of bringing an object into the cache, F_i is the frequency that LFU tracks and L is equal to K_f where f is the most recently evicted cache element. They dub this policy, Least Frequently Used with Dynamic Aging (LFUDA). Others, like Robinson and Devarakonda [181], choose to age slightly differently by protecting new entries to the cache and ageing the entire cache by reducing all reference counts C to $\lceil \frac{C}{2} \rceil$ whenever the average reference count exceeds a predefined maximum value. Both of these techniques produce results comparable to LRU, but the hardware cost is much higher due to the number of counters. In addition both approaches are concerned with the size of objects in the cache, something that is not a concern in this work.

A further augmentation of LFU from Kelly, Jamin and MacKie-Mason [101] allows weighting parameters that come from the memory system to indicate how 'useful' the caching of an element is. The problem, the authors admit, is the difficulty of obtaining the weights and additionally that this approach still requires the implementation of LRU to resolve ties. A final interesting approach to frequency type statistics is from Mekhiel [129], which proposes a two level cache with the Most Frequently Used (MFU) elements going into the equivalent of an L1 cache and the LFU elements going into an L2 cache. This allows frequently accessed data to be easily available to the CPU and not easily evicted. This still suffers from the same cache pollution problems and high hardware overhead mentioned previously.

A slightly different approach is to take inspiration from probability theory as Least Frequently Used (with k previous references) (LFU-K) [195] does, to predict the number of occurrences of an element (page, cache block etc.) in a reference string. The development from LFU is that it adds extra terms into the estimation formula to account for the changing probability of referencing an element over time. In this work Sokolinsky demonstrates that in terms of reducing cache miss rate, LFU-K outperforms LFU and LRU. However, the effectiveness of this technique is intrinsically linked to the estimation of two parameters, m and h, neither of which is a trivial task. In addition this augmentation targets database systems, so complex calculations can be handled by a collection of powerful CPUs. As our context is medium sized embedded systems this technique is less immediately applicable, unless a method of estimating the probability functions easily could be found.

LFU is an improvement on simple FIFO policies, but suffers from the problem of cache pollution and a very high hardware requirement, particularly in the perfect case [158]. Considering recency rather than frequency has long been considered a better metric to approximate how far in the future a piece of data will be needed; therefore, the next section covers techniques that consider recency.

Recency Based Techniques

Recency, as a general class of algorithms orders the elements in a cache by the time they were last referenced. This leads to two very general categories of recency-based algorithms, Most Recently Referenced (MRRe) and Least Recently Referenced (LRRe). MRRe algorithms are much less common than LRRe and in general are less performant due to their poor temporal locality [151]. Therefore, we will not focus on them in this thesis. The most popular LRRe algorithm is LRU [99, 157, 194], which makes use of temporal locality, and given a few simplifying assumptions, is very easy to implement.

In terms of the performance of LRU if we consider the work of Robinson and Devarakonda [181], for a 2MB cache `OPT` has a Miss Ratio of around 15% when run using the UNIX trace. By comparison for a cache of the same size `LRU` has a Miss Ratio of around 25%. This gap is much more pronounced at smaller cache sizes where for the same trace `OPT` displays a Miss Ratio of 22% while LRU is 40% so this is particularly a problem for the medium sized embedded systems we're interested in as their memory size and by consequence their cache size is very small.

LRU has other problems as well, it performs very badly in a shared data environment or when using virtual memory [19]. In addition, when the working set [53] of the program exceeds the size of the cache, cache-thrashing [54] occurs. This is most commonly seen in large loops [118]. There is also the problem of dead blocks [120], where blocks read into memory are never referenced again, but take time to be evicted, clogging up the cache. Finally, as Lin and Reinhardt [118] describe, LRU lacks a desirable property: as associativity in the cache increases the miss rate should decrease; however, the opposite is often true.

Statistical Inference

Due to these shortcomings, the important aspects of LRU from our point of view are the extensions to LRU to overcome them. The first of these is to try and use statistical inference or measured history of accesses to predict the future behaviour of a program and act accordingly. This is the approach taken by O'Neil, O'Neil and Weikum [141] where the Least Recently Used (with k previous references) (LRU-K) algorithm is described. This technique uses Bayesian methods to estimate the inter-arrival times of memory references from the collected set of past references. Vakali [214] continues this idea with the History Least Recently Used (HLRU) policy. Under this scheme a function $hist(x, h)$ is defined, which returns the h^{th} past reference to the cache object x. It then uses the maximum value of $hist$ from among the cached objects to decide on a replacement.

Wong and Baer [228] describes two algorithms Profile Reference Locality (PRL) and Online Reference Locality (ORL). The essential idea of these algorithms is to favour lines that exhibit temporal locality by marking them using special instructions. PRL does the identification offline using profiling, but ORL does the profiling at runtime, keeping a table of hits to non-Most Recently Used (MRU) lines and using this to set temporal bits. This then allows lines that exhibit temporal locality to be favoured at eviction time over those that don't. All of these policies show increased performance over LRU to varying degrees, but the problem with all of them is the extra hardware and book-keeping required. In addition, none of these algorithms address some of the underlying flaws in LRU, such as its susceptibility to flooding [71]. The next set of approaches address these concerns.

Overcoming Flooding

Flooding is a phenomenon particularly associated with LRU, which happens when applications try to access a large address space in a sequential fashion, and is most commonly seen in large loops. As more addresses are accessed and the cache capacity is exhausted, old elements (those at the beginning of the space) are evicted from the cache. Then when the loop begins again old elements are no longer present leading to further unnecessary memory accesses, this process is shown in Figure 2.4. This means that the cache policy has no positive impact on the execution at all. To address

```
 1  ...
 2  start:
 3      addi    a0, 100
 4      addi    t0, 0x2000
 5      add     t6, zero, zero
 6  loop_start:
 7      lw      t2, 0(t0)
 8      lw      t3, 4(t0)
 9      lw      t4, 8(t0)
10      lw      t5, 12(t0)
11  computation:
12      add     a1, t1, t2
13      and     a1, a1, t3
14      sub     a1, a1, t4
15      mul     a1, a1, t5
16  loop_check:
17      addi    t6, 1
18      add     t0, t0, 16
19      bne     t6, a0, loop_start
20  ....
```

Cache - First Iteration - Line 10
0x4512 - (Addr: 0x2000)
0x2378 - (Addr: 0x2004)
0x9174 - (Addr: 0x2008)

Cache - Second Iteration - Line 7
0x7170 - (Addr: 0x200c)
0x2378 - (Addr: 0x2004)
0x9174 - (Addr: 0x2008)

Cache - Second Iteration - Line 8
0x7170 - (Addr: 0x200c)
0x4512 - (Addr: 0x2000)
0x9174 - (Addr: 0x2008)

Figure 2.4: Flooding usually occurs in large loops, where the working set of the program exceeds the size of the cache, as the example above demonstrates. On the first three loads of the first iteration, everything proceeds as usual. Once Line 10 is hit however, the element from address `0x2000` is evicted under the LRU policy, leading to the situation we see in the second example cache above. Then when Line 7 executes in the second iteration, we are forced to evict the element from address `0x2004`. Consequently, the cache is useless in this situation, as every memory access will be a miss. This is flooding.

this Glass and Cao [71] proposes the `SEQ` algorithm which records long sequences of requests for sequential pages and applies MRU replacement to those sequences. Otherwise it defaults to simple LRU. However, this has a high overhead to implement compared to LRU and only considers one cause of flooding.

Smaragdakis, Kaplan and Wilson [193] further develops these ideas to create the Early Eviction Least Recently Used (EELRU) algorithm. Here the definition of 'sequential' is weakened to make the algorithm more amenable to data structures that are not contiguous in memory. It does this via an adaptive feature in which the behaviour of the algorithm changes between standard LRU and what Smaragdakis, Kaplan and Wilson refer to as the Wood, Fernandez and Long (WFL)[229] algorithm, which can evict pages before they become the LRU page. EELRU merges these together by calculating probabilistically whether WFL will have more hits than measured from LRU over multiple sets of parameters for WFL. Midorikawa, Piantola and Cassettari [131] builds on these ideas further by proposing Least Recently Used with Working Area Restriction (LRU-WAR), which switches between LRU and MRU when sequential sections are detected.

A further approach to the problem of flooding is cache partitioning, as proposed by Kim et al. Here the cache is divided into three regions, a sequential region, a looping region and an other region. When a reference is requested it is classified into one of the three categories and then different replacement algorithms are run on each region accordingly. This is implemented in software, however as our context is medium-sized embedded systems this would be very difficult to implement because not only would you require all the tracking apparatus as mentioned in our discussion of LFU but also the decision mechanism to decide which part of the cache the new arrival should reside in. An even more interesting attempt at partitioning, using LRU in one portion of the cache and `RAND` in the other to break the predictable way LRU responds to flooding, is proposed by Das et al. [49].

All of these techniques are at least comparable to LRU and many perform at least as well, but all have their problems. Several of them have very high hardware requirements, particularly Kim et al. [110] as 3 separate replacement policies are implemented. In addition, very few guarantees can be made on the behaviour in Das et al. [49] because of the use of `RAND`. The main problem that a lot of these policies have though is that because they are attempting to classify memory access in an online fashion they have to store a lot of state. In order to cut this down some authors have suggested the use of static analysis and this what the next section explores.

Static Analysis

Static analysis and the use of compiler techniques to enhance caching, augment compilers and programs with cache hints, adding to the amount of information available to a cache when a replacement decision needs to be made. Jain et al. [88] tackles this by augmenting the Instruction Set Architecture (ISA) of a processor with `KILL`, `KEEP` and `COND-KILL` instructions, which are used instead of normal `LOAD` and `STORE` when a variable is considered dead. The `KILL` instruction is used for short-lived variables or ones that are only accessed once and `KEEP` for long-lived variables. This method shows an increased hit rate over multiple levels of associativity. Wang et al. [218] simplifies this further through the use of an ‘evict-me’ bit which is set when a reference is accessed that is “sufficiently far away” [218] or has no reuse. This is done by issuing a different instruction, so that the compiler is responsible for setting the evict-me bit. The problem with techniques of this kind is that they are difficult to implement for existing systems because they require changes to the ISA. Moreover there is a lack of integration between compilers, processors and caches, which would mean crafting a new toolchain for approaches like this to work.

Adding Costs to Misses

A consistent assumption of all the techniques seen so far is that cache misses all have a uniform cost to them. However, research in the recent past has shown this is not the case [167]. This may be for simple reasons, such as the data is stored in the L2 cache rather than in main memory, or it may be that memory access times are variable dependent upon the particular placement of data in memory. Consequently, several authors have attempted to integrate cost functions into LRU algorithms to improve performance by prioritising low-cost misses. Jeong and Dubois [90] is an early example of this, where each block is not only associated with a last reference time, but also with a cost of replacement. This way the algorithm can make a choice between evicting the LRU element or evicting the non-LRU element with a lower cost. Over several iterations of the algorithm, a 15-18% increase on LRU is recorded, with minimal extra hardware requirement, up to 35-bits extra for a 4-way associative cache with 64-byte blocks.

This approach is developed further in Kharbutli and Sheikh [107] with the introduction of the Locality-Aware Cost-Sensitive (LACS) algorithm. Instead of using a 2-cost model, where a miss is cheap or expensive, this technique records latency when a miss happens for a particular element. These costs are incremented and decremented as other events occur in the cache, such as a hit to an element or a different element not being accessed for a long period of time. This results in large drops in the miss rate, but depends a lot on the working set size relative to the size of the cache. Das and Kapoor [47] takes a similar approach but use latencies calculated from a Network-on-Chip (NoC) rather than a traditional hardware arrangement. The big problem with these approaches is how the cost function is arrived at and what variables it considers. The examples previously presented use different criteria, but the potential list of criteria is endless, making it difficult to draw out the salient metrics for reducing latency. Also, none of these authors contend with the problem of *calculating* the cost as a program is running; Jeong and Dubois [90] uses information that can easily be inferred from memory addresses and Kharbutli and Sheikh [107] uses simple counters.

Insertion Policies

One of the consistent problems of LRU is that in reality it is only a small subset of cache elements that actually get referenced after they are read into the cache [168]. This means that a lot of elements sit in LRU caches for a long time reducing effective cache capacity. The root cause is that under standard LRU all elements are inserted in the MRU position and then progress to the LRU position over time. Qureshi et al. [165, 168] asks whether changing this might lead to better cache utilisation and proposes a suite of new insertion policies to achieve this. The policy they eventually arrive at is known as the Dynamic Insertion Policy (DIP), which combines standard LRU with a policy known as the Bimodal Insertion Policy (BIP). The algorithm switches between these two policies when one performs better than the other. In order to track the utility of switching, a technique known as set-duelling is used, where a small part of the cache is dedicated to each policy and the miss-rate for each part is calculated before it is decided which algorithm should be used for the rest of the cache. In terms of effectiveness, DIP represents a 21.3% reduction in Misses per Thousand Instructions (MPKI) as compared to LRU, representing an over 60% reduction of the gap to `OPT`, which itself improves on the performance of LRU by 32.2%.

Sreedharan and Asokan [197] take a similar approach, but use a reference count attached to each cache block. If it is high, the insertions happen at the MRU position. Gu and Ding [74] take a slightly different approach they call collaborative caching. Here the cache is split in half into MRU and LRU sections so that data can be inserted in different places. The problem with the last two solutions particularly concerns how to gain the information to decide when to use each instruction on the fly. In Qureshi et al. [168] and Sreedharan and Asokan [197] there are mechanisms to decide; however, this does not exist in Gu and Ding [74] as it is a more theoretical paper. All these policies improve performance and some close the gap to `OPT` significantly, but there are still times where caching policies like these will make mistakes due to the relative paucity of information. Therefore, our next set of solutions addresses this problem through mistake correction.

Mistake Correction

Kampe, Stenstrom and Dubois [98] suggest that the wide gap between `OPT` and LRU is indicative of LRU making too many mistakes. They demonstrate this through experimentation with the SPEC95[176] where in the case of the `su2` and `tomcatv` benchmarks `OPT` improves LRU by 60% and 80% respectively. In response they propose a policy of self-correcting LRU, which adds a feedback loop to the LRU policy and starts with the goal that no mistake should occur more than once. To do this they employ a shadow directory to track when blocks are evicted too early and a mistake history table to persist the information even after blocks are removed from the shadow cache. There is also an MRU victim cache, which catches blocks that bypass the cache and ones that are evicted from the MRU position so that miss-predictions can be quickly recovered from. These improvements together give a 24% improvement in miss rates in their experiments, but this is at the cost of quite a lot of extra hardware and bookkeeping to manage this extra state.

Defining New Metrics

Some policies take a slightly different view of resolving the problems with LRU by defining new recency metrics built on LRU. One of the first examples of this is the Low Inter-reference Recency Set (LIRS)[92], which instead of tracking the time each block was last referenced tracks the Inter-Reference Recency (IRR).[2] The assumption of the policy is that if the IRR is high for a block it will continue as such, so it is safe to replace, because it will not be needed in the near future. Choo, Lee and Yoo [36] scales back this idea slightly and defines a Degree of Inter-reference Gap (DIG) scheme that tracks the number of references between consecutive accesses to a block multiple times instead of just once as in the case of LIRS. Jaleel et al. [89] develops from a policy of Not Recently Used (NRU) by expanding the number of counter bits available to encode more history in the counter. This technique is known as Re-reference Interval Prediction (RRIP) and has static and dynamic variants.

Having now toured many different manifestations of LRU and other recency based policies we can see a pattern emerging. When compared to `OPT`, LRU will always make mistakes and miss-predictions, so will never track `OPT`'s performance perfectly. There are many techniques to close the gap somewhat, but no technique has done it perfectly up to now. This implies that recency is not the only piece of information needed to make the best caching decisions; therefore, the next set of policies to be considered combine recency and frequency to close the information gap with `OPT` and attempt to match its performance.

Methods Combining Recency and Frequency

Having considered recency and frequency in isolation it makes sense to ask, can the two sources of information be usefully combined? Many authors have attempted to bridge this gap and their solutions fall into a few key categories.

LRU with Cache Partitioning

Cache partitioning involves logically subdividing the cache into multiple regions, where each region has a different probability of replacement. Consequently, some elements become more protected than they would under a standard LRU or LFU policy. This is often combined with frequency counts, which is the approach taken by Robinson and Devarakonda [181], where the cache is partitioned into `new`, `middle` and `old`. Elements start in `new` when they are first referenced, and here they are protected from reference counter increases. As time goes one they slowly move towards `old`

[2]For a block A, the number of blocks that are referenced between subsequent references to A.

as the time since their last reference increases. When it is time for replacement, the element with the lowest reference count in the `old` section is selected, with LRU used to break ties. Karedla, Love and Wherry [99] takes a similar approach, but only divides the cache into two sections and abstracts the frequency count to either 1 or 2 or more. Osawa, Yuba and Hakozaki [145] meanwhile uses generational caching to split the cache into N generations with cache elements moving towards generation N on every hit. Also presented is the addition of a small history list, which means that if an entry is found there on insertion, it can be inserted into generation 2 instead of 1, as it is more recent than something the cache has never seen.

Juan and Chengyan [97] takes a similar approach to Osawa, Yuba and Hakozaki in using N partitions of the cache, but they apply it to Chip Multi-Processors (CMPs) and so use the cache organisation to give each core a part of the cache, while allowing stealing between cores if that is of benefit. The problem with a lot of these schemes is whilst they are often very good, (for example, Robinson and Devarakonda [181] boasts of closing 34% of the gap between `OPT` and LRU), they rely on "user-specified magic parameters" [19] to set the size of the generations for highest effect. Implementing these in reality would require a lot of performance tuning or guessing to get the correct size of generations. If this were wrongly set, a lot of accesses in a short period could mean that elements were protected, when in reality it might only be a particular short burst of accesses that required that element. As a result cache partitioning is often used as an auxiliary tool to enhance other algorithms, as opposed to being used on its own.

New Cache Structures

One of the more popular techniques to integrate recency and frequency is to introduce new structures into the cache to re-organise the data. These structures are mostly logical in nature, but can be very effective. The first of these is `2Q` [93], which divides the cache into two queues known as A_m and A_1. A_1 is further subdivided in two A_{1in} and A_{1out}. The essential principle is to admit "only hot pages to the main buffer" [93] so A_1 acts as a filter. If a page is not re-referenced while in A_1 it is unlikely to be hot and so is evicted. This approach is expanded even further in Zhou, Philbin and Li [237] where the idea is expanded to n LRU queues where n is a tunable parameter. This has a low overhead compared to LRU and boasts a moderate improvement of between 5-10% when $n = 2$.

Megiddo and Dharmendra S. [128] present the Adapative Replacement Cache (ARC), a self-tuning cache that uses a cache directory to list elements that have been accessed either once recently (L_1), or twice or more (L_2). The algorithm then attempts to keep p pages from L_1 and $c - p$ pages from L_2 in the cache, where c is the size of the cache and p is altered as hits and misses occur on different elements. This was further advanced by Bansal and Modha [19], which combines the algorithm with `CLOCK` [40], turning L_1 and L_2 into clocks in order to remove some of the disadvantages of LRU not addressed by `ARC`. The adaptability of `ARC` is only triggered on a page miss, so Chen et al. [35] advanced the algorithm even further by basing it on hits to cache elements instead. This improved performance compared to `ARC`, because it eliminates the lag on the adaptability due to waiting for the first page to miss before it triggers.

In work by Li, Liu and Bi [115], only one queue is used as a cache directory and each cache block is associated with an R and F value to track recency and frequency respectively. This cache directory (Q_{out}) maintains two counters O and H. When an entry is found in Q_{out} before insertion into the cache, H is incremented. O is incremented otherwise. The ratio between these counters decides whether the R or the F value is used when a replacement is required. Zhang and Xue [234] split each 'way' in an n-way cache into k groups where elements bubble up these groups on a hit and removals are taken from the set of elements at the bottom of each of the groups. Taking LRU as a baseline they measured that `OPT` reduced MPKI by 30% while their solution reduced it by 13%, closing the gap by 47%, one of the best reductions in this subsection.

New Objective Functions

Rather than introducing new structures into their cache architecture some research attempts to integrate frequency and recency together by defining a new objective function to use for replacement. In strict LRU the objective function can be formulated in natural language as "which element has been accessed least recently?", but some research attempts to change that to make better replacement decisions.

Reddy and Fletcher [175] creates an objective function that applies a weighting to the contribution of recency and frequency, using a parameter α to control the weighting. Donghee Lee et al. [59] takes this a stage further and subsumes this all into a simple function controlled by a parameter λ which is shown to subsume all weightings of frequency and recency. This was later developed by Cui and Samadzadeh [42]. AbdelFattah and Samra [1] takes a similar approach, but calculates all the weights relative to every other element in the cache and then sums them using constants for weights, concluding that weighting frequency five times as much as recency gives the best performance. This is further optimised by AnandKumar et al. [11]. Das and Banerjee [45] takes another similar approach, but simply takes the product of frequency and recency allowing each to weight the other.

All previous papers on objective functions use methods that weight the contribution to replacement of frequency and recency, but other papers try to define entirely new metrics that move beyond this. Tian and Liebelt [205] considers effectiveness as "the rate of re-use of the block over [a] future time period" [205], which is realised as $\frac{r * R_{count}}{f * E_{count}}$ where r and f are the recency and frequency weights, the R_{count} is a count of re-references and E_{count} is a count of how long has elapsed since the last re-reference. These new objective functions have a wide spectrum of success, from improvements of 9% over LRU to outperforming ARC, which itself outperforms LRU by a considerable factor. The problem with implementing many of them would be the explosion of counters and calculation units that would be required, even in relatively simple cases with small caches. In addition, the problem of 'magic parameters'[19] recurs, most of the proposed solutions rely on having a good sense of what the workload for the cache will look like a-priori, which is simply impossible for a standard implementation.

Adding Frequency as a Second Criterion

A final category of integrations of recency and frequency are algorithms that simply augment LRU with information about frequency, adding it as a second criterion by which to select a replacement candidate. This is certainly true of Chang, McGregor and Holmes [29], which uses a policy LRU* where cache hits increment a counter on each cache element. On a replacement every item checked has its hit counter decreased by 1 and a replacement is only made when a counter hits zero for a particular element. This is further developed in Alghazo, Akaaboune and Botros [7], whereby for each replacement the LRU and second LRU are compared, with frequency counters used to decide if the LRU element should be saved. Dybdahl, Stenström and Natvig [62] rounds out this set of enhancements by increasing the frequency differently for reads and writes and implementing cache bypassing for particularly high frequency counter values to immediately promote elements to high levels of the cache hierarchy if necessary. In terms of utility, these schemes have similar problems to redefining the objective function, in that the proliferation of counters may make them a problem in CPU caches as opposed to simulations. The problem of 'magic parameters'[19] also persists, with many of these algorithms having multiple tuning parameters that would need to be estimated prior to use.

Beyond Recency and Frequency Combinations

In recent years there have been attempts to move beyond collecting frequency and recency information to create a replacement policy. These fall roughly into three categories, the first being tracking new metrics and using those to inform the replacement policy; the second is switching policies on the fly (based either on miss-rate or other metrics); and thirdly, reorganising the cache.

Defining New Metrics

Taking new metrics first, the earliest example of this is the work of Rizzo and Vicisano [180] which calculates a value (V) for each element in the cache as $V = \frac{C}{B} * P_r$, where C is the cost of retrieval, B is the benefit of removal and P_r is the probability of re-reference. Each of these values has other factors that feed into its calculation, which allows the policy to shape itself around these multiple factors. In Kharbutli and Solihin [108] two new metrics are proposed the Access Interval Predictor (AIP) and Live Time Predictor (LvP). AIP increments a counter for a cache line whenever an access is made to another line in the same cache set. LvP, on the other hand, counts the number of accesses to a line in a single generation (time from insertion to eviction from the cache). In either case once the counter reaches a particular threshold the line is evicted. Keramidas, Petoumenos and Kaxiras [103] takes a slightly different approach and attempts to predict the reuse distance or the number of accesses to the L1 cache between address references. All these techniques suffer from similar problems, in that many of them rely on 'magic parameters'[19] which have to be set a priori.

The work of Duong et al. [61] and to some extent Tada [203] takes a slightly different approach in that they abstract specific metrics into an overall 'score' for each cache element. When it comes time to evict an element it's the one with the lowest score that is evicted. Scores are set initially on entry into the cache and are then altered as different events occur, with Duong et al. [61] having tunable parameters for how events affect overall scores. In general, these policies show that increasing the information available to the caching policy will increase its ability to make better decisions, something that we will return to in our analysis of the literature and ideas to move forward.

Dynamically Changing Policy

The second type of policy in this area is an attempt to take the best elements from multiple cache policies by changing the policy that the cache applies dynamically based upon observable conditions within the cache itself. An early example of this is in Altman, Agarwal and Gao [8] where the adaptivity happens offline through the use of genetic algorithms, but this does not account for changing policy as a program executes, as later work does. The real genesis of this idea, in an online form, is ACME [13, 73, 177], which uses small virtual caches to test the effect of a suite of policies on the miss rate of the cache and then computes a set of weights that minimises the miss rate via machine learning. Subramanian, Smaragdakis and

Loh [202] develop a similar idea, but give a more realistic implementation than the original presentation of ACME by focussing on combining only 2 cache policies and implementing partial tags to reduce the hardware cost. This approach sees reductions in miss rates across a wide variety of benchmarks and cache configurations, and represents some of the best that replacement policies can achieve in terms of miss-rate reductions. Not only that, but since this approach is adaptable, it requires no 'magic parameters'[19].

A sub-variant of this idea is presented in Jongmoo Choi et al. [94] as well as other research [4, 5, 30, 31, 192], with the key difference between them being the metrics they choose to use to perform the adaptation. Jongmoo Choi et al. uses estimated forward distances, Chang, Chiang and Yu uses classification of Program Counter (PC) accesses into sequential and looping accesses and Aguilar and Leiss uses a variety of metrics that are all observable by the cache directly. All this put together gives some of the best results in this category of implementation, but there are still problems. For one, the virtual caches required or the extra metric tracking can take up a lot of extra hardware, particularly if a machine learning element is included. In addition, all of these policies are still under-performing if compared to `OPT` [127]. There is still a significant gap that needs to be closed in order to really attack the latency that caches introduce and sadly despite their promise, these techniques still do not achieve that.

Cache Reorganisation

A third attempt to move beyond recency and frequency combinations is to re-organise the cache in light of other information such as PC values. The first of these is Chaudhuri [32], which changes the insertion policy in order to make sure dead blocks are not kept in the cache longer than necessary, while the working set of the cache remains untouched. Manikantan, Rajan and Govindarajan [124] takes a slightly different approach by dividing up the ways in the cache to prioritise a set of delinquent PC values that contribute large numbers of misses to the overall count. This set of PCs is detected and changed adaptively as the program executes. Finally, Khan, Wang and Jiménez [105] builds on earlier work seen in Johnson and Shasha [93] to dynamically alter the size of the once referenced and more than once referenced cache partitions. All of these obtain some speed up (particularly when looking at processors with multiple cores) and most are adaptive, but the hardware overhead is still prohibitive when considering a CPU cache rather than a web or software cache.

Summary

In the last section we have seen various attempts to reduce the miss-rate of caches through the augmentation of their replacement policies but how do these different attempts compare? Figure 2.5 shows the improvement in cache miss rate, where LRU is used as a reference. This graph is necessarily incomplete as not all papers provide this data and care has been taken, where possible, to use comparable cache sizes and benchmarks for each measure but of course this was not possible in all cases.

Taking all these works in mind, how much further does it take us towards resolving the problem of increasing memory latency? A lot of the policies considered do make significant progress in reducing the number of cache misses (and hence the cache miss ratio) as Figure 2.5 shows, and therefore reducing effective latency. However, there are two problems that this body of work does not address, the ceiling placed upon effectiveness by `OPT`, and the lack of consideration of non-functional requirements by some paper authors.

OPT Ceiling

Having considered all these cache policies, one thing is consistent: none of the policies ever matches the theoretical performance of `OPT` for the same cache size. This is clearly shown in Figure 2.5, as OPT achieves a 70% improvement over LRU while the next nearest policy is only at 56%. While this makes sense, because it is impossible to argue that a perfect knowledge of the future is better than an imperfect knowledge of the present, this leaves us in a very frustrating situation because it cuts down the utility of trying to reduce memory latency with cache policies.

For example, let us consider the figures presented in Panda, Patil and Raveendran [151] and take an example program that consists of 100000 executed instructions. In line with the measurements made in Bienia et al. [24] and Limaye and Adegbija [116] let us assume that of these 100000 instructions, 35000 are memory instructions. Let us further assume, that as per our context, we only have a single level of set-associative caching, the processor has a single core and that memory accesses takes 2 cycles for a cache hit and 150 cycles for a cache miss. These figures are consistent with the figures presented in Hennessy and Patterson [82], assuming a 500MHz clock speed. Finally let us assume that it takes 4 cycles to execute an instruction. If we suppose that LRU under this configuration of hardware has a miss-rate of around 20%[3]. Ignoring for a moment the effects of pipelining, if we apply the measures stated above we get the following table.

[3]This is sensible because this rate is to be used as the reference, we could choose any value but this fits with data gathered in Qureshi et al. [168].

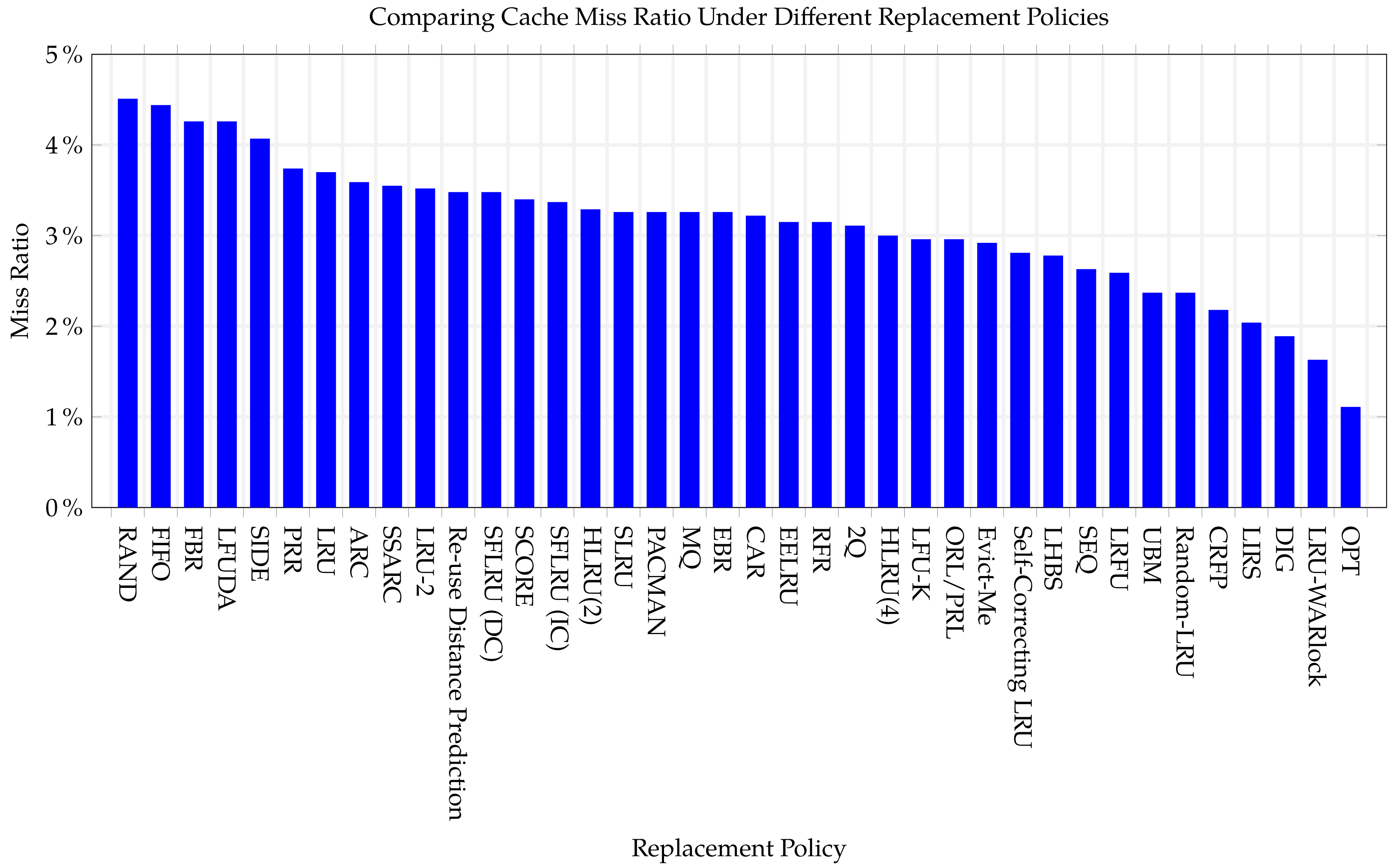


Figure 2.5: This graph shows a selection of the replacement policies covered in the previous section and the calculated miss ratio that policy achieves. This is extrapolated by taking the miss-rates presented in the papers and calibrating them against the miss rate presented in Hennessy and Patterson [82], for a 32KiB Cache, 4-way associative cache. As can be seen although many policies improve on LRU and some by significant amounts, OPT is still the lowest miss rate. This shows that despite all the progress in cache technologies replacement policies are still bounded above by `OPT` in terms of effectiveness. However even then, `OPT` does not reach a miss rate of 0% meaning there to reduce latency beyond that currently offered by cache policies we need to look beyond the cache to other mechanisms.

Replacement Policy	Miss Rate	Cache Misses	Estimated Runtime (Cycles)	Estimated Runtime (us)
RAND	24.40%	8540	1733920	3467.84
LRU	20.00%	7000	1506000	3012.00
SLRU	17.60%	6160	1381680	2763.36
LFU-K	16.00%	5600	1298800	2597.60
LRFU	14.00%	4900	1195200	2390.40
LRU-WARlock	8.80%	3080	925840	1851.68
OPT	6.00%	2100	780800	1561.60
Perfect	0.00%	0	470000	940.00

Table 2.1: This table shows the effect of changing replacement policies on the number of cache misses that could be incurred as well as the estimate runtime for the example program described in the paragraph above. It should be noted that there are a lot of simplifying assumptions that have been made but the overall message is clear, focusing a large amount of effort in closing the gap to `OPT` will not yield the biggest gains. If we want to make significant progress in reducing memory latency and therefore program runtime we need to be looking beyond `OPT` and therefore beyond cache replacement polices.

If we consider that `OPT` is provably optimal [127] then we are forced to accept that the maximum speed up we could obtain by focusing on cache policies is around a third, based on the difference between the best performing cache policies we have and `OPT`. However as we can also see from the table, even `OPT` is someway from a perfect cache policy since, despite it's clairvoyant nature, it's still essentially reactive. What this means is that it always takes action when requested, rather than querying it's knowledge of the future to arrange circumstances so misses could be avoided. Of course even in these cases it's possible that situations could arise whereby a miss is unavoidable (i.e. misses that occur when the cache is cold) but this table shows that in terms of potential gain we have to move beyond `OPT` rather than focusing on trying to close the gap between our best performing policies and `OPT`.

Implementation Concerns

If we take a broad look at the papers we've considered in the previous section we see that they fall into three distinct categories, with respect to implementing their ideas. The first, are papers that are purely theoretical, they don't consider an implementation and are simply concerned with presenting the the theoretical properties of the techniques they describe. Sadly this covers 38 of the 60 papers considered, and includes work by Ari et al. [13], Arlitt et al. [14], Das and Banerjee [45] and Gu and Ding [74]. This is a problem because as there is no implementation there is no consideration of non-functional properties, like time and space, which are incredibly important when we're considering applying this to a real system as it allows us to understand the domain into which these solutions could be deployed. For example some solutions are targeted at Web or OS caches, which have very different time and space requirements when compared to hardware caches. In addition they don't attempt to quantify any overheads their techniques might introduce which again makes it more challenging to assess the efficacy of the technique when compared to other approaches.

The second category are those that do consider implementation to some degree but then fall back to simulation for their experiments. This covers 18 of the 60 papers, and includes work by Das and Kapoor [47], Johnson and Shasha [93], Subramanian, Smaragdakis and Loh [202] and Wong and Baer [228]. The level to which this is done varies however, the vast majority consider the hardware costs of their implementation and some focus on time complexity as well. Of these that are considered though very few of these claims are backed up by experiment and again are still not implemented. This is clearly better than the previous category of not even considering the implementation, and were they to be implemented they would seem to be efficient, but still leads to problems because a lot of the claims made are simply assertions, they're not backed up by measurement in the vast majority of cases.

Finally the third category is those that actually do implement their cache policies but as we shall see even then these are not very helpful to us. 4 out of the 60 papers ([94, 110, 177, 237]) actually do this but of those 4, 3 of them implement their chosen technique in software on top of an OS, and the last one is implemented as a web proxy cache which is on a different scale to the kind of caching we're considering. Even in these cases, very little about non-functional requirements is written, again making it difficult to understand exactly how efficient these might be so again it's challenging to understand how they might aid or hinder our efforts to reduce memory latency in a real scenario.

This shows there is a large gap in research of cache policies around implementing the designs that are proposed, and consequently measuring the non-functional properties that these policies have. For example, as we're considering medium sized embedded systems it's important that we know how much extra hardware would be required, both for the storage of the extra data but also for the calculations associated with the cache. Some solutions simply require an extra bit to be added to a cache whereas others require calculations that may need extra functional units adding to the processor. Without these results it makes comparing the approaches very difficult, and also makes it more difficult to rule out unworkable solutions for our context if the paper does not make it clear.

2.2.2 Augmenting Cache Architectures

Having considered cache policies, we can now turn our attention to Cache Architectures and how effectively designing them can lead to reductions in latency. In the coming section we will consider: Associativity, Multi-Banking, Multi-Level Caches, Non-Blocking Caches, Pipelined Caches and Victim Caches.

At the outset of this section it is important to restate the purpose of this literature review, which is to explicitly search for evidence of a reduction in latency via the use of the above techniques. As a consequence, several papers in each of the areas just described will be omitted, as they are primarily concerned with saving energy in the cache rather than reducing memory access latency. It is also worth pointing out that trace caches [174, 182], though very useful for reducing latency in instruction caches, will not be included either, because this thesis focuses on reducing data cache latency and trace caches specifically target instruction caches.

Increasing Associativity

Put simply, the level of associativity a cache presents is the number of alternative places a cache block could be placed for a given cache index [147]. For example, if you have a cache with 128 entries that is divided into 8 equally sized sets then the cache has an associativity of 16, as there are 16 different places you could place an element for a given set. Associativity forms a continuum that ranges from 1, which we refer to as a Direct-Mapped cache, to an associativity of n where n is the size of the cache. This latter form of cache is known as fully associative.

Increasing associativity introduces a trade-off, because as associativity increases, the miss rate for a cache goes down. This is because there are multiple locations for each element inside the cache, cutting down on conflict misses.[4] However, because these locations all have to be searched when querying the cache, the access time for elements in the cache goes up on average [104]. Consequently many researchers have tried to find ways to balance this trade-off, with the ultimate goal of producing a cache with a high level of associativity and a correspondingly low access time.

There are three general approaches to solving this problem, the first is to accept that to have the benefits of associativity searches are inevitable, and therefore to focus on making them as fast as possible.

[4]Conflict misses occur when a new block is forced to displace one currently in the cache, because they map to the same cache address.

Fast Searching

Kessler et al. [104] is one of the first to attempt this approach and formulates a scheme whereby the tags for elements in the cache are stored in associative sets, but are ordered by MRU. This arrangement could allow some searches to be ended early, if MRU is a proxy for most likely to be accessed, and thus save time on average. As an alternative, the work also introduces partial comparison for tags. This means it is only necessary to consider the first $k = \lfloor \frac{t}{a} \rfloor$ bits, where t is the length of the tag in bits and a is the level of associativity. This means tags that are definitely not in the cache can easily be thrown out, which on average decreases the access time, as a search is avoided. However, although overall this approach gives associativity for a relatively low cost, there are many factors that govern its performance that are outside its control. These include the design of the program and various cache design parameters, including the length of tags, so it fails to be a good candidate for latency reduction. Calder, Grunwald and Emer [27] takes a similar approach, but uses a steering bit to direct the search towards one or the other bank within the cache. When this is correct it negates the need for a second costly search and so reduces the average case access time. The approach taken by Calder, Grunwald and Emer is described by Zhang, Zhang and Yan [233] as optimal for a 2-way Set-Associative cache.

Building on the aforementioned optimality, Zhang, Zhang and Yan presents an approach that provides multiple entry points into each set, known as "major locations" [233]. It then tracks "selected locations" [233], which are locations that have been loaded from main memory as a result of other misses to this major location. Consequently, it is possible to construct a simple search algorithm, that checks the major location first (which is kept updated with the MRU element) and then, if that fails, checks the selected locations until either a `LOAD` from main memory is required or the data has been found. This way the number of searches required is reduced as the likelihood is the check to the MRU element will succeed. The problem with these techniques is that although they improve access time on average, they introduce variable latency and complexity for very small gains. In addition, these are very much attempts to mitigate the problem of latency to provide parity with a Direct-Mapped cache, and as will become clear in later sections, there is much more to be done to bring significant latency reductions.

Removing Searching Entirely

An alternative approach to that discussed above is to try and remove the problem of search time entirely by making it unnecessary. This is the approach favoured by Agarwal and Pudar [2] which implements pseudo-associativity by having two hashing functions to reduce conflict misses. The index from the address is put through the first hash function and if this results in a clash it is then put through the second. This eliminates the need for searches and via some optimisations means the second lookup can easily be bypassed. Hallnor and Reinhardt [77] also uses hashes to create an Indirect Index Cache. This cache both eliminates searches and breaks the link between the placement of tags in the tag array and location of cache data in the data array. This is done by hashing tags directly from addresses and then looking up the result in a hash table which contains the index required to find the associated data. Unfortunately, from the results presented the performance is only competitive with current caches. There are no large gains to be made in reducing latency.

Seznec and Bodin [26, 186, 187] also use hashing functions to eliminate searching by using a separate hashing function for each way in the cache. Djordjalian [57] builds on Seznec, but builds to a higher level of associativity by grouping 4-ways into subsets of 2 and then re-using Seznec's technique on each of the sub-levels. Sanchez and Kozyrakis [184] takes this even further and uses multiple hashing functions to simulate arbitrary levels of associativity with the Z-Cache, but with much improved performance due to search reduction. The problem with a lot of these schemes is that they require a lot of extra hardware to implement, especially in terms of functional units if complicated hash functions are required. Not only that, but the high hardware cost does not have a matching large drop in latency associated with it; therefore, it is uneconomical to add such a high hardware cost for relatively little gain. In addition, the extra latency added from calculation of the hash function, while it may be less than searching a whole cache, is not trivial. Thus, in a lot of cases, it may be that one source of latency is simply being traded for another.

Selective Application

Some research attempts to maximise the benefits of associativity by only applying it when the program will benefit. For example, if a working set fits perfectly in a Direct-Mapped cache, then all the extra cycles spent on managing a fully associative cache are wasted. Batson and Vijaykumar [20] takes this idea and fuses together the ideas of direct-mapping and set-associativity by trying to map everything directly and falling back to a Set-Associative mapping only if that fails. This work also incorporates a predictor for the Set-Associative side to reduce probing delay. Aly, Nallamilli and Bayoumi [9] uses the idea of different mapping functions in parallel and varies the associativity on the fly, based on offline profiling of the program. The speed-up gained, however, is only 2%, with most of the other gains being in power reduction. In addition, the applicability is limited, because it requires a-priori knowledge of the code that is going to run, making this only applicable for a limited class of embedded systems.

Qureshi, Thompson and Patt [166] takes the idea of variable associativity but rather than using lookup tables to remap memory throughout the cache, the size of the tag store is increased, so there is no longer a 1-1 correspondence between a tag and a location in the data store. This is dubbed the V-Way Cache and these features mean associativity can increase and decrease as the workload demands. The V-Way cache is used as a component in Deepika and Lee [51], where the primary way for an index is directly mapped to the data, but the V-Cache is used for the other minor ways. This does lead to a large drop in miss rates, but at the cost of more hardware and higher power utilisation. Das and Kapoor [48] takes a slightly different approach and partitions the ways in an associative LLC into normal and reserve portions. Recently evicted blocks can then be shared between reserve portions to effectively increase the associativity of certain sets in the cache as the need arises. In the extension work [46] the sharing is limited to “fellow sets” [46], or sets that are ‘near’ to each other and miss rate reductions of 30% are recorded when compared against their baseline CMP.

Multi-Banked Caches

Multi-banked caches focus on architecting the cache into multiple portions that are all placed on a common interconnect. This interconnect allows them to interface with several `LOAD` and `STORE` units, increasing the potential for cache parallelism, and in the best case, spreading the data more evenly throughout the cache [178]. When a memory read or write is required the address is routed in two phases, first to the correct bank and then to the correct line. This allows for accesses that have very similar low order bits to be spread out amongst multiple banks to allow faster recall. Rivers et al. [178] combines this technique with memory re-ordering by the compiler to increase the efficacy of the approach. One of the problems with the interconnect approach is that bank conflicts can introduce non-determinism in the latencies seen when accessing data. Neefs, Vandierendonck and De Bosschere [137] solves this by extending the work of Rivers et al. [178], adding a predictor. This means that on a correct prediction the latency is constant and known in advance. However, the latency reductions are quite low, and in the optimal case rely on a clairvoyant prediction mechanism, which is impossible to implement.

Multi-Level Caches

One of the larger areas of research in increasing cache performance has been the use of multi-level caches. Their development follows a roughly chronological progression, necessitated by the ever-increasing nature of the processor memory gap, but little appetite to spend orders of magnitude more on memory hardware. Over time, it has become the pre-eminent mechanism for increasing cache effectiveness used in modern processors. The technique works by having multiple caches that increase in size and 'distance' from the processor. These are usually arranged into numbered levels (L1, L2 etc.), where the number increases with the distance and access time.

Early Work

The development of multi-level caches begins with the acceptance, after analysis in Przybylski, Horowitz and Hennessy [161], that there are limits to the effectiveness of a single cache, no matter how large it may be. Przybylski, Horowitz and Hennessy [160] characterises the optimal cache as having a short cycle time (serves hits quickly) and a low miss ratio. It suggests expanding to two levels of caching to reduce the miss penalty in the L1 cache without having a corresponding increase in cycle time [95]. Azimi, Prasad and Bhat [17] refines this notion and shows that while this technique is very good at reducing latency, the key is placement of cache objects within the cache hierarchy, in order that more latency is not introduced with second level cache misses. Moreover, Tang and So [204] discusses the benefits of making the L2 cache much larger than the L1 to reduce lengthy miss pauses, but also acknowledges the degree to which the workload itself determines the most effective cache hierarchy. In effect there is no silver bullet for every conceivable program.

Expanding Beyond Two Levels

Expanding beyond two levels of caching, Zhao et al. [236] considers four levels of caching, with the LLC being composed of DRAM, the same memory technology that main memory is usually constructed from. Although the capacity of main memory is reduced due to chip space constraints, lower latencies are seen, because the data has not got to travel off-chip. The problem with a large DRAM LLC is storing a large number of tags to address it. There are several options, but the optimal one is to only use partial tags and accept a string of false positives, while also using a sectored cache on chip. Wu et al. [230] extends this and proposes that every level of the cache should be constructed from a different memory technology to gain the most from the in-built advantages of each. A variety of cache designs are proposed, and even 3D stacking is introduced, which achieves an 18% Instruction-per-Cycle (IPC) increase compared to a standard three level cache. This does not address all concerns though, as problems with the endurance of memory technologies are not considered. There is also an acknowledgement that much better performance could be gained if the construction of the cache were tuned to the specific requirements of the program to run.

Hameed, Bauer and Henkel [78], extended in 2014 [79] takes a similar view, but proposes a hybrid LLC composed of DRAM and SRAM to act as its own independent cache. A MissMap [121] is used to decide on whether there is a hit in the DRAM or not and set-duelling is used to decide on a policy to refill the L3-DRAM, a large source of latency. There are even proposals to allow applications at runtime to decide whether to bypass the LLC [219]. This decision is made by monitoring the miss-rate or the miss latency for an LLC access and there are lots of technical challenges this poses, including issues of cache coherency that are yet to be addressed. All this culminates in the work of Tsai, Beckmann and Sanchez [207], which proposes that software should define the cache hierarchy from a pool of generic resources. This is very much the optimal utilisation of this technique, as some workloads will not benefit from deep hierarchies and actually a large L1 cache may be preferable, for example.

Replacement Policies in Light of Multi-Level Caches

Little work has been done on replacement policies that act across all levels of the cache hierarchy. Indeed, the opposite is often true, with attempts frequently made to keep caches self contained, simple and without an overall managing process, even if they exist as part of a hierarchy. Thus, most of the work on replacement policies simply adapts work seen in Section 2.2.1. Kelwade et al. [102] considers several policies including `PROMOTE`, `DEMOTE` and a hybrid method. The first two policies use the number of promotions and demotions in the hierarchy to decide which block should be replaced, but the latter combines those two measures to give better performance. Cache replacement policy over multiple levels of caching is certainly an area ripe for further research, particularly perhaps used as a supplement to existing techniques to identify hot and cold data and thus cache data more effectively.

Non-Blocking Caches

Non-blocking caches attempt to decouple the cache from synchronously accessing external memory. First proposed by Kroft [112] in 1981, this approach works by effectively buffering the memory operations that miss in the cache in Miss Information/Status Holding Registers (MSHRs). This allows stalls for data dependencies on reads to be ignored until they become essential, and via the use of write buffers, allows writes to memory to be dealt with asynchronously. Chen and Baer [34] found that write penalties could be eliminated entirely, as long as reads could bypass writes in a write buffer.

There are limits to the technique, however. Belayneh and Kaeli [23] discusses that in general, the more resources spent on these caches (wider buffers etc.), the better they become, but eventually other unrelated delays dominate the latency calculations, so there is a sense of diminishing returns. To be specific, in the examples shown in their paper, increasing the number of MSHRs to 6 virtually removes most structural hazards, but 29.4% of load misses are data dependant so this is unaffected by the number of MSHRs. Tuck, Ceze and Torrellas [209] develops this idea further and proposes a hierarchical MSHR structure to support many more outstanding misses. Each cache bank contains its own MSHR along with a bloom filter that links to a large MSHR file. This leads to a large increase in performance, but the limitation is then on the compiler and the program author. If the code consists of relatively few data hazards, this technique will work very effectively, but there is no guarantee of this. Closer hardware and software co-operation is needed to consistently produce code that can exercise these structures to their best effect.

Pipelined Caches

Applying the same techniques to caches as to CPU designs, pipelined caches [142, 143] have also been proposed as a way to counter latency. They work in a similar way to the multiple stage pipelines that are seen in CPUs, so any memory access required because of a cache miss or similar can be split into several smaller phases that can be interleaved. For example Gunadi and Lipasti [75] uses 4 stages: in the first stage subarray decoding for data and tag is performed. This is followed by the second stage where row decoders are activated and the tag array is read. In the third stage the data array is ready and partial tag comparison is performed. Finally in the fourth stage the rest of the tag bits are compared and then data is further selected, if the cache is an associative one. Breaking the process of caching into multiple stages allows caches to effectively hide some of the latency they experience in accessing memory and challenges the dichotomy that as cache size increases so does access time. Agarwal, Roy and Vijaykumar [3], Martin et al. [126] and Srivastava et al. [198] show examples of this. Unfortunately this technique has been rather ignored in the recent past, with only Hong and Kim [84] developing the idea. They use asymmetric pipelining to resolve the problem that SRAM cell access can be somewhat unpredictable in its latency, by extending the cell access phase in the pipeline. The SRAM also adds pseudo-multi-banking to regain any bandwidth lost.

Victim Caches

The final architectural technique employed to reduce latency is victim caches. First proposed by Jouppi [96] in 1990, they grew out of the idea of miss caching. In this technique, when a cache miss occurs, a small (2-5 entry) fully associative cache is also loaded with the data that is being loaded into the main cache. As a result, if in future the entry disappears from the main cache because of a conflict miss, the data will be retained in the miss cache, so retrieving it will be faster than requesting it from main memory. The victim cache takes this one step further by removing the duplication present in miss caching. Hence, in a scheme with victim caching, anything that is evicted from the main cache is stored in the victim cache instead of duplicating the main cache. This leads to a stark drop in the number of conflict misses, some benchmarks increasing from removing 0% of conflict misses to 45% of conflict misses and in one case removing conflict misses altogether.

This idea is further developed by Stiliadis and Varma [199] which adds a more selective element to the victim cache. This formulation acts very much like Jouppi [96]. However, if an access misses in the main cache, but hits in the victim cache, a prediction algorithm is invoked to decide if the main cache and victim cache should be swapped. Similarly, if the access misses in both caches the prediction algorithm is again used to decide which element should be evicted. This approach improves on Jouppi, but is deficient when turned towards data caches, because the elements retrieved are inherently less predictable than instructions. Hormdee, Garside and Furber [85] develops the idea into an asynchronous formulation, but achieves little in terms of latency reductions.

The next step change in victim caches comes from Khan et al. [106] which proposes virtualising victim caches through the re-use of dead blocks. The idea is that if a block is dead it is taking up valuable capacity with no benefit. Khan et al. uses this to suggest that these dead-blocks could be re-appropriated as a victim cache, so that all the benefits of retaining elements are kept, but without any extra hardware to store them. This leads to a significant decrease in MPKI, but is entirely dependent on the effectiveness of the dead block predictor, as a incorrect identification can lead to extra latency being incurred to re-load the evicted block. A selection of papers following this improve further on the concept. Asaduzzaman, Allen and Jareen [16] proposes a Smart Victim Cache which includes a level of cache locking to keep good data in the cache longer. Navarro and Hübner [136] proposes an adaptive victim cache that adapts its size and level of associativity as the program progresses. However, it does not provide any concrete methods to achieve this. Finally, Subha [201] suggests changing the victim cache from being fully associative to using a small register to cut down cache searches. Overall, victim caches are a very useful concept and have been extended to cover a wide range of use-cases. There needs to be more research into the adaptive models considered by Navarro and Hübner [136], as this could be a very fruitful area of research, but the extra resource they require needs to be justified and may not always be utilised to its fullest capacity.

2.2.3 Summary

We have now come to the end of our consideration of cache intrinsic techniques and have seen the plethora of techniques available. These techniques are summarised in the graph presented in Figure 2.6, which includes the extra entries considered in this section to give an overall picture of cache intrinsic changes and their effects on miss rates. Sadly there are many fewer entries to be included in this graph because the data for many techniques was not available in the papers referenced. In addition it is difficult to compare many of these techniques fairly because there are so many factors that can impact a cache's miss rate and these are varied to a greater or lesser degree by each author in different ways, making it difficult to establish a reliable baseline. To further compound the problem, as each author has a different focus when presenting these techniques it may be the case that metrics crucial to our comparisons are omitted, because they are not relevant to the discussions in the paper. The best way to accurately compare all these techniques would be to implement them all with a known fixed set of constraints and then measure them, however that would be prohibitively time consuming and would not add significantly to the discussion in this thesis.

Considering the techniques presented in Figure 2.6, many of them provide appreciable reductions in latency, however there are several fundamental problems that prevent any of these techniques entirely solving the problem we presented in Chapter 1.

Cache replacement policies are all bounded above by `OPT` and even with the best adaptive policies no researchers claim they have found a way to produce miss rates below what could be expected from `OPT`. Moreover, even `OPT` does not claim a miss rate of zero. It will still incur misses due to cache capacity for example; hence, even trying to implement `OPT` is not a complete solution, if our aim is to push latency as low as possible. Not only that but very few of these policies are actually implemented so it's very difficult to gain an insight into their impact on non-functional properties of the cache. In terms of Cache Architectures, we see a similar story. There are many good options here, but a lot of them aim for parity of latency with existing models, rather than trying to push the latencies lower. In addition most of the techniques that score best in terms of low cache miss-rates require very complex hardware to achieve,

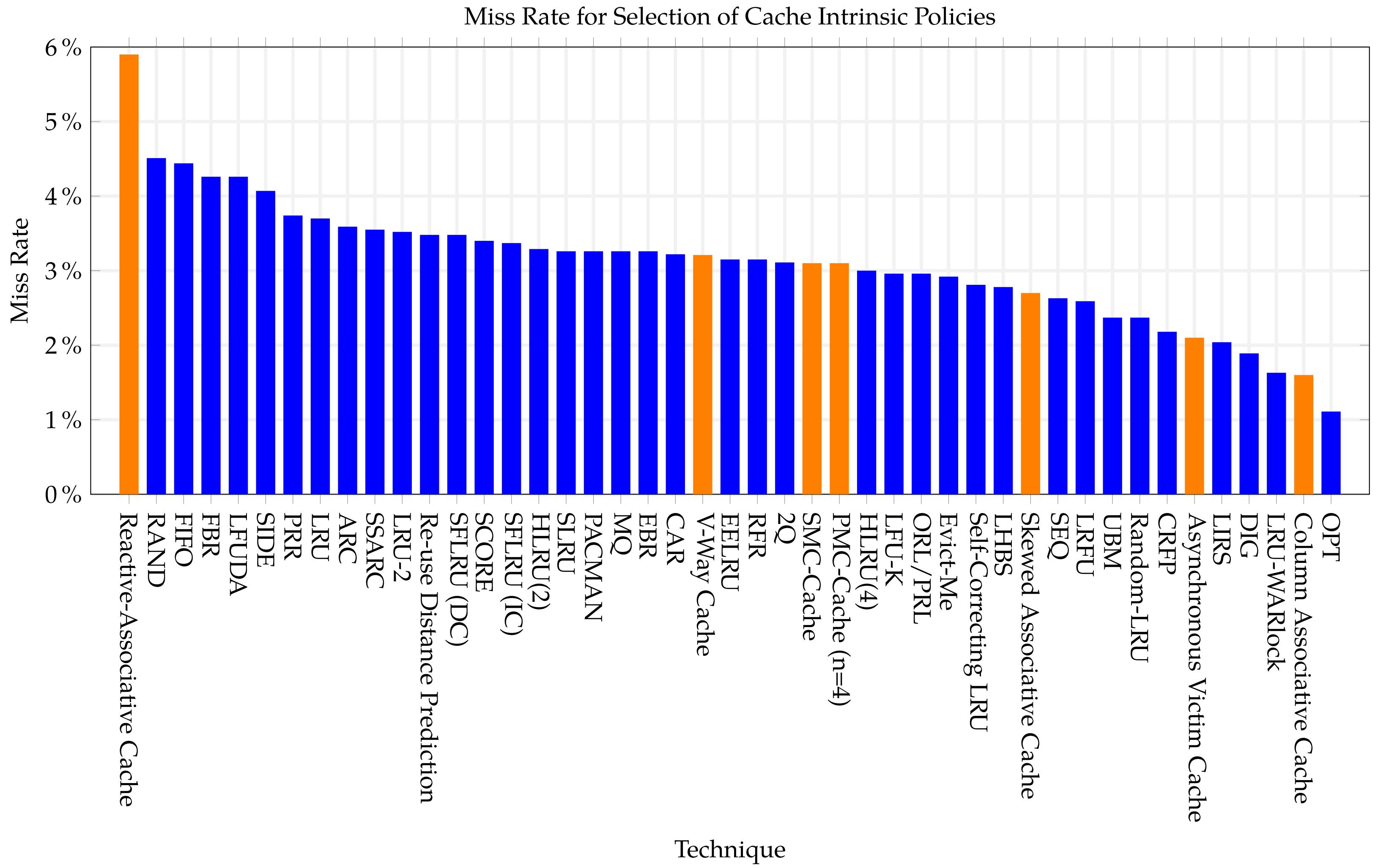


Figure 2.6: The graphs extends Figure 2.5 by adding the further cache augmentations which it was possible to compare. This was made difficult by a lack of comparable metrics but allows us to see that still `OPT` is the best strategy to reduce miss rate and therefore latency. There also maybe grounds to combine some of these techniques, as many of the papers on cache augmentations did not explicitly consider replacement techniques.

or require wholesale re-thinking of how caching works. This leads to the question of whether anything else could be changed to reduce latency outside of the cache, which our next section will discuss.

2.3 Cache Extrinsic Techniques

If we return to Figure 2.3, we can now consider the second branch of methods to reduce latency, which are not focused on the cache. First we will consider prefetching, a scheme where latency is reduced by trying to execute memory operations in advance of the need for them. That will be followed by a section on Memory Scheduling and how re-ordering memory accesses can be beneficial. Finally, there will be a short section on Code Transformations to aid latency reduction.

2.3.1 Prefetching

Prefetching, at its heart, is very simple. It means attempting to issue memory instructions before they are needed by the CPU, so that the long latency main memory access can be hidden. Obviously this is idealised, but many authors have constructed schemes that perform a function similar to this with varying degrees of success. Prefetching methods are loosely split into software and hardware types, but some combine elements of both. A crucial point to make here is that, as mentioned in Section 1.1.2, the difficulty of prefetching varies enormously with the kind of items that are being prefetched (instructions or data), due to the characteristics of each. On the whole, programs are structured in large blocks of sequentially executed code, as the success of the basic block abstraction of programming shows. Therefore, prefetching is much easier because the shape of programs in memory is more predictable. By contrast, data can have very little apparent structure in memory. Particularly for complex, pointer-laden, data structures, patterns of data can appear almost random, without any intuition as to the underlying structure. As a consequence, data prefetching is much more difficult and requires either an acceptance of a high rate of misprediction or a high degree of introspection in the program to achieve. Consequently, in this section we focus on data prefetching.

Software-based Prefetching

Software-based Prefetching tends to involve either the compiler or the programmer giving explicit instructions to the memory control system to perform a prefetch action. This is exactly the approach taken by Callahan, Kennedy and Porterfield [28] in that the compiler decides when a prefetch instruction should be executed. This is focused on loops and makes the assumption that the latency of a load in the loop will be covered by one iteration of the loop. With this in mind prefetches are inserted into the loop body, so that on the next iteration of the loop the necessary data will always be present. However, this is quite a primitive approach, so in work by Mowry, Lam and Gupta [133] the algorithm is refined, in an attempt to prefetch for only those accesses that are likely to be cache misses.

Zhang and Torrellas [235] takes the idea of explicitly inserting instructions further by marking groups of records that should be prefetched together in the code. This allows the compiler to bind these seemingly disparate pieces of data together, so that when they are accessed in the program, they can be prefetched. Another approach to software prefetching is proposed by Lipasti et al. [119], in the form of Speculatively Prefetching Anticipated Interprocedural Dereferences (SPAID). This is a heuristic that inserts prefetch instructions for the data referenced by pointers when they are used as arguments to a procedure. Luk and Mowry [122] takes a similar approach, but considers pointers in recursive data structures, proposing several different algorithms for prefetching them. The problem with both of these schemes is that one is only proposed and the other is implemented by hand, adding lots of complexity to the task of a programmer. They also leave open the question of how the recursive structure or pointer candidates are discovered, which is the key to the efficacy of approaches like this.

There are several problems with software prefetching that mean it's not worth exploring any further. The first is that the addition of prefetch instructions can as much as double the size of the executable code [113]. This a problem for two reasons. Firstly, it is uneconomical to double the code size when we're considering medium sized embedded systems due to the obvious memory constraints; secondly, there is no guarantee that all the extra instructions will be effective. The placement of software prefetch instructions must be made at the 'Goldilocks'[5] point, before the actual memory request is made. If the prefetch is made too close to the memory request, it will not have time to complete before it is required; so the processor will still stall. On the other hand, if it is made too far away from the access, then there is a chance it could be displaced from the cache by another memory operation, which would still cause a cache miss. If that were not challenging enough, to even calculate the 'Goldilocks' point would require knowledge of the system that is either very hard to attain or may not even be knowable statically. This includes the latencies of all the memory components involved or the state of each component at a particular code point. As a result, this section will focus on hardware-based prefetching.

[5]Not too close, but not too far away.

Hardware-based Prefetching

In comparison to software-based prefetching, hardware-based prefetching does not rely on the programmer or compiler to insert prefetch instructions. Instead, the prefetching happens entirely transparently to the programmer and is handled by structures in hardware that dictate when and how prefetches occur.

The earliest example of this is Smith [194], describing One Block Lookahead (OBL), which simply fetches the block next to the current one. This can be triggered either on every miss, on every memory access or using a tagging system. Hence, it prefetches on a miss or when a prefetch is good, i.e. it prevents a miss. Jouppi [96] develops this idea with a hardware structure known as a stream buffer. This uses Smith's ideas but places the prefetched data into an auxiliary structure, rather than straight into the cache. This avoids the issue of having prefetched data replaced by another cache operation. This is also effective at reducing the program runtime, as Farkas, Jouppi and Chow [64] demonstrates, with 26% reductions on average.

Regular Data Patterns

Fu and Patel [65] develops the ideas of OBL by defining it as a special case of sequential prefetching, where the memory unit is instructed to get the next p consecutive blocks after the desired block. However, as they point out, some data is stored in a pattern which is still regular, but has large gaps. Therefore, they introduce stride prefetching, whereby p blocks are still prefetched, but each block is separated by a given number of bytes. Baer and Chen [18] makes the stride dynamic by predicting it based on tracking good and bad prefetches and correcting the stride length accordingly. A similar approach is taken by Fu, Patel and Janssens [66], with the use of a stride prediction table, although this is limited to prediction in loops only. Dahlgren, Dubois and Stenstrom [43] takes on the idea of adaptation, but applies it to sequential prefetching instead, monitoring the number of successful prefetches and slowly increasing the degree of prefetching.[6] Meanwhile, Palacharla and Kessler [148] synthesises several ideas to extend stream buffers with filtering and non-unit strides. Chen and Baer [33] extends previous work [18] to create a correlated reference prediction table that solves the problem of miss-prediction when nested loops are involved. All of these approaches show reductions in execution time for the benchmarks they use, but they are still focused on predictable regular data patterns like loops or contiguous memory sections. The next step in the development of this method is accounting for irregular data patterns.

[6]how many elements are prefetched in each iteration.

Irregular Data Patterns

One of the first pieces of work to attempt to account for irregular data patterns is Alexander and Kedem [6]. In this approach stride prefetching is taken as a baseline, but instead of trying to predict individual addresses to prefetch this work attempts to predict blocks of addresses. This takes advantage of the fact that memory references tend to cluster. Therefore, if you can predict a block of addresses effectively, you will decrease latency through spatial locality. Lin, Reinhardt and Burger [117] tries to use spatial locality as well by fetching around misses, but also couples the prefetch to the memory controller, so that prefetches are only issued when the memory system is idle. Though this method sees large speed-ups in some benchmarks, the authors admit more work is still to be done to shape the prefetching to the application.

Solihin, Lee and Torrellas [196] attempts to solve the problem of prefetching complex structures by using pair-based correlation prefetching. Under this scheme chains of misses are detected by the system and stored so that action can be taken on multiple misses at once, after they have been observed the first time. Cooksey, Jourdan and Grunwald [39] develops the ideas of address identification and pointer chasing by attempting to identify virtual addresses as they are returned from memory systems, so they can be prefetched. This is combined with a standard stride prefetcher to give 12.6% speed-up on average throughout a suite of benchmarks. Finally, Yu et al. [232] also uses correlation prefetching for the specific case of accesses in high performance computing of the form, `A[B[i]]` where `A` and `B` are arrays.

Despite all this, prefetching the disparate memory accesses that often occur in general programs is still difficult to do through correlation as there is not enough information to predict some accesses. Many authors recognise this and as OoO processors developed, the next step in hardware based prefetching was to separate out the prefetch controller to allow it to be asynchronous from the processor. This effectively lets the processor define locality rather than the placement of objects in memory.

Asynchronous Prefetching

One of the first steps down the path to asynchronicity was by Veidenbaum and Gallivan [216], who took the drastic step of splitting memory and computation onto two entirely separate processors, with a specialised compiler. They reasoned that this would give the ultimate overlap, as the memory processor could run ahead of the computation processor. However, this approach was rendered less effective by the inability of the compiler and programmer to produce code amenable to this process. For example, if a program has lots of memory accesses with no, or very few, computation instructions between them, the ability to prefetch will be severely diminished as there will not be enough time to have completely performed a prefetch operation before the data is requested. Roth, Moshovos and Sohi [183] takes a more conservative approach and proposes a structure based on a correlation table and a prefetch buffer. This is combined with a method of pointer chasing that allows the prefetcher to remain ahead of the processor. However, there are issues with address identification and also problems with letting the prefetcher run too far ahead due to the 'Goldilocks point' mentioned in the previous section.

Vander Wiel and Lilja [215] takes a similar approach to Veidenbaum and Gallivan in letting the compiler control which prefetches are issued, but offloading the task of issuing them to a separate data-prefetch-controller. Work by Collins et al. [37], developed in 2002 [38], takes the idea of separate execution, but optimises it by precomputing a subset of the program to prefetch dynamically and then executing it in a separate thread context. This is augmented in the later paper by adding a pointer cache to break load dependencies, where memory instructions wait for pointers that are also stored in memory.

Mutlu et al. [134, 135] takes this concept the furthest with the run-ahead processor. Under this scheme when a long latency instruction is hit, the program state is checkpointed, as in a context switch, and the processor is left to run. This allows independent memory instructions to execute without needing to wait. None of the operations are actually committed to the processor's registers, as once the long latency instruction stops, processing is resumed and all interim results are erased, but this means that in the ideal case, a lot of prefetching has happened in the run-ahead period.

All these techniques show promise, but they rely on a lot of assumptions, particularly about the amount of independent instructions that a processor has between long latency instructions. Unfortunately, some programs are very prone to not exhibiting this behaviour. Therefore, although these techniques are useful, there are limitations on what they can achieve without substantial care on the programmer's part or an incredibly sophisticated compiler, with knowledge of the internal hardware of the processor.

Adaptive Prefetching

All these methods lead to increases in the performance of the system under test, but still suffer from the inherent problems seen in previous iterations of hardware-based prefetching. To move beyond those limitations, adaptive schemes are introduced so that the prefetching mechanism itself is not a-priori dictating the form of memory accesses, but is instead being driven by the compiler. This development is very similar to the development of cache policies that we saw in Section 2.2.1. Nesbit, Dhodapkar and Smith [138] were some of the first to try this approach. Building on their work on global history buffers [139] Nesbit, Dhodapkar and Smith use a system of calculating address deltas to find the next address to prefetch. The adaptive part comes by monitoring this algorithm and tuning the zone size and prefetch degree or turning off prefetching altogether when it is degrading performance. This idea is developed further by Arora, Banerjee and Davina [15] which takes the monitoring and optimisation idea and switches between multiple prefetching algorithms, as the prefetching degree changes in a predictable way. This is taken to its ultimate conclusion by Panda and Balachandran [150] which uses multiple expert predictors and weighted majority voting to decide the best element to prefetch.

2.3.2 Scheduling

Scheduling based techniques are often used to re-order accesses to memory to decrease latency. All are designed in some way to make the schedule of memory accesses more suited to the underlying memory technology and this is done in a number of ways.

New Scheduling Algorithms

Kiniwa and Kameda [111] were one of first authors to pursue this route, taking a similar approach to the work on non-blocking caches that we saw in Section 2.2.2. Under this scheme all memory requests are placed into a queue and this can be re-ordered to maximise the length of time elements spend in the cache. Unfortunately, this method requires some knowledge of future events, so is not a completely online process, and has a significant overhead to implement. Yang et al. [231] look at resolving `LOAD-LOAD` dependencies[7] faster, via a data forwarding mechanism to allow more overlapping. This has somewhat limited utility, but is the only place that explicitly considers data hazards in this context. Luo et al. [123] tries a similar approach but uses genetic algorithms to generate a scheduling algorithm offline. This gains a modest speed-up, but requires the software under discussion to display predictable patterns of execution.

The same is true of Kegley et al. [100] and Wei-Che Tseng et al. [222]. However, rather than genetic algorithms, they propose a slightly different optimisation approach by making tasks 'wander' around the schedule in the latter case and assigning a System Metric for Applciation Cache Knowledge (SMACK) score to various schedules in the former. Qazi et al. [164] describes a method of re-organising loop operations

[7]places where processors load an address from memory and then immediately access it.

to fill up empty slots and so make them more efficient, although the reduction in latency is quite small and its unclear how to apply the technique outside of the predictable context of loop execution. Modgil and Sehgal [132] take a more drastic approach and allow the memory controller to adapt its policy on when to act on write commands based on the level of memory traffic it observes. This is certainly a step in the right direction, but the performance increases are highly correlated with the memory configuration, showing this policy is much less general than necessary for our purposes.

Exploiting the Underlying Memory Structure

Approaching this from the point-of-view of a scheduling algorithm requires more information than the system can provide in the current paradigm. Other research has instead chosen to exploit the properties of the underlying memory technologies. Rixner et al. [179] increases the complexity of the memory controller, adding queues of pending references and an address arbitrator to produce a stream of DRAM instructions that (in more aggressive cases) take advantage of row and column locality to reduce the number of actual access the DRAM must make. Shao and Davis [188] takes the idea further in the context of OoO processors and groups memory transactions into bursts due to the non-uniform access latency of Synchronous Dynamic Random Access Memory (SDRAM).

These methods are all reasonable and achieve approximately a 30% performance increase [179] and 21% execution time decrease [188], but the real problem with reducing particularly DRAM latency is that the memory has preset timing parameters to ensure the data is valid when it is accessed. These parameters change incredibly slowly are now very long compared to processor cycle times. However, Hassan et al. [80] makes the observation that it is possible to reduce the timing parameters if the data has been accessed recently, due to the electrical properties of DRAM. Therefore, they propose a small cache of recently accessed rows that dictate the values of the timing parameters, allowing a performance increase for row local data. Shin et al. [189] pushes this idea even further, but rather than caching the rows, they track the time between refreshes, exploiting electrical properties of the DRAM to restore data that had previously been present more quickly. Kim et al. [109] brings a similar approach, describing a new kind of DRAM called Solar-DRAM. This new memory type has different regions characterised as strong or weak, allowing different timing parameters to be used for different regions. The idea here would be to place key data in the strong regions, so as to speed up latency when required.

Augmenting the Memory Controller

Rather than approach this from the point of view of the memory technology, other research has added new hardware to the memory controller, in general to integrate the controller more with the cache and memory systems. Stuecheli et al. [200] proposes a virtual write queue and scheduled write-backs so the memory controller directs the cache to transfer lines, and the writes are 'harvested' from the LRU part of the LLC, this section becoming the virtual write queue. Wasly and Pellizzoni [220] on the other hand splits the memory controller out of the processor completely and uses the existing Direct Memory Access (DMA) module to control memory behaviour, instead of delegating this responsibility to the CPU. This has a lot of similarities to Veidenbaum and Gallivan [216], though prefetching is not considered.

All these methods exhibit performance increases against the benchmarks they run but there are fundamental problems with attacking this problem from the point of view of scheduling, no matter which general approach is taken. The big problem is that in many cases there simply is not enough information available online to make the best decisions. This problem stems from a number of sources: the lack of integration between the cache, memory controller and memory hardware; the simplicity of memory controllers due to the lack of available space on a processor die; and the lack of knowledge in a compiler of the underlying memory system. This last point has been addressed in the literature, and in the last part of this section we will therefore consider how programs can be transformed and data can best be laid out in memory to reduce latency.

2.3.3 Program Transformation & Data Layout

The idea that there are better and worse ways of laying out data in memory is not new. So many processes (prefetching and caching among others) rely on the idea that data used together will be 'physically' close together, so it makes sense to use processes to increase the likelihood of this occurring, if at all possible. This is the approach taken by Panda, Dutt and Nicolau [152] which considers clustering of variables in the code and the insertion of dummy instructions to make sure that pieces of data accessed together are assigned to different cache lines, so that they do not displace each other. Other research considers loop fusion, a technique first demonstrated by Gao et al. [68] in 1993. Here multiple loops can be merged together into a single loop to reduce program runtime.

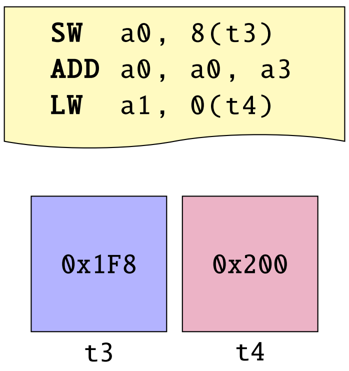
A myriad of techniques can be added on top of this to increase the technique's efficacy: Ding and Kennedy [56] proposes the use of hyper-graphs to discover the data sharing between loops, an improvement on the original technique as it means more than two loops can be fused if there is sufficient data sharing. Gomez et al. [72] pushes this even further and proposes loop morphing which, via loop splitting and fusion, can make even non-conformable loops fuse under some circumstances. Marchal, Catthoor and Gomez [125] in turn describes a technique where loop fusion is done incrementally and over several passes to gain the maximum speed up. On the data side there has been comparatively little work done from the point of view of latency reduction. Qazi et al. [164] describes a technique where you can search the array accesses for potential conflicts and place them in different parts of memory, akin to Panda, Dutt and Nicolau [152]. In recent times more significant work has been done in Wei et al. [224] which uses a mix of offline profiling and online monitoring to place data and code into an environment of different memory types. This leads to a reduction in average memory latency of 12.1% and reduced energy costs.

2.3.4 Summary

This marks the end of our consideration of cache extrinsic techniques, so what have we learned? Many of these techniques do produce appreciable reductions in both latency and overall program runtime which would seem to make them ideal candidates to answer some of the questions posed in Chapter 1. Prefetching in particular seems very promising, but all the examples seen are still only able to exploit fairly simple predictable patterns, rather than covering the diversity of expression that can be seen in programs, such as pointer-laden data structures and or non-deterministic system calls like `malloc`. Scheduling and program transformation are also a sensible ideal, but they have very little consideration for dynamic behaviour due to the disconnection between many of the hardware elements present. What is necessary to improve a lot of these techniques is the ability to consider dynamic behaviour alongside static. This would allow us, over time, to get a real picture of what operations actually cause latency, rather than accepting a lack of information on this point. If a way existed to feedback this dynamic behaviour to the cache and memory system, it could be incorporated with the static information and used to improve the quality of the predictions made by these techniques. One way to include this dynamic information is through the use of tracing and creating a feedback loop throughout the memory system, something the next section explores.

2.4 Incorporating Tracing to Reduce Latency

In the third portion of this literature review we consider tracing. Tracing is very simply a way of producing an ordered list of all the instructions a program has performed during its operation. This may not immediately seem useful, but the huge benefit of traces is that they include dynamic information about memory accesses that would not normally be accessible. For example, imagine there are two memory accesses to two addresses, one is stored in register `t3` and the other in register `t4`. If we see, in a program listing, that we store to the address in `t3` and read from the address in `t4`, is it impossible, without simulating the program and all the previous register interactions, to know if they are the same address. This prevents a lot of optimisations and insights that would be possible if this information was available, as Figure 2.7 demonstrates.



```
SW   a0, 8(t3)
ADD  a0, a0, a3
LW   a1, 0(t4)
```

Figure 2.7: This shows a programming code fragment of RISC-V assembly code and two registers `t3` and `t4`. Without conducting extensive simulation of all register accesses it would be impossible for a static analysis tool to know that at this point in the program that the memory address referred to by the first `STORE` instruction and the last `LOAD` instruction are the same. Consequently re-ordering these instructions, though it might look safe could lead to the program not functioning as the programmer intended.

There has been very little work in online trace processing, i.e. processing and reacting to the trace while the program is running, and even less work on applying those ideas to the problem of latency reduction. However, some work has been done to utilise tracing in various offline processes, some of which are detailed in the next few sections. To bring this consideration of the literature to a close we will also consider some examples of work done on using tracing as a method of on-chip debugging, as this forms the basis for the implementation that will be presented in future chapters.

2.4.1 Tracing as a Control Loop

Tracing is most often used as a tool to generate large amounts of data that can be explored offline, with engineers then making manual adjustments to the program as a result of the knowledge gained from the trace. Hu and Chen [86] is an example of this with the TraceDo System. This system, aggregates traces from DSPs in the System-on-Chip (SoC) and streams them into an emulator that adds timestamps to the various events. The data is then explored offline and is used to optimise long running functions in the program. Wang, Gao and Zhang [217] makes similar use of an emulator, but utilises the trace to drive cache emulation in a separate process, continuing a trend of making information useful to the programmer, but not to the system. This trend continues in Li and Mayer [114] and Mertz and Nunes [130]. In addition they both focus on trace collection and reducing the size of the traces collected, rather than on acting on the information the traces give.

The problem with all of these ideas is that they all consider trace analysis and action as something to be done offline, rather than as part of the system itself. However, some authors have tried to make this work. Singh et al. [191] describes a process of mapping applications to cores on a Multi-Processor System-on-Chip (MPSoC) while the applications are running. This is a step towards the kind of closed loop feedback that would be of benefit, but sadly the trace collection is done at design-time not runtime. In addition this model targets throughput rather than latency reduction, but there is no reason why the objective function used could not be changed. Shoukry et al. [190] proposes a system for scheduling content delivery on mobile devices, attempting, via the analysis of usage patterns, to predict what content users will want and fetching it via WiFi, rather than placing a large burden on Mobile Data Networks. Though this example is from an entirely separate field, this approach is exactly the one we should aim to emulate. There are questions to be answered about how accurate their statistical modelling is, but nevertheless this is an example of a tight feedback loop, using traces to predict activity to increase performance.

2.4.2 Tracing for In-Silicon Debugging

To finish this short section on tracing we will consider the use of tracing as a method of in-silicon debugging. While the relevance of this may not be immediately apparent, many of these papers formed the basis for the implementation of the solution that will be presented in Chapters 3 and 4. The first of these is Uzelac and Milenković [213], which presents an off-chip debug host that mirrors the internals of the system being traced. This idea in turn is picked up by Decker et al. [50], which also starts to do the analysis in this offline module, converting the trace stream to series of events that can be processed. This idea, of taking a large amount of trace information and filtering the crucial events, while also having to deal with instruction reconstruction, is crucial for the development of our solution in Chapter 3. A couple of ancillary issues are covered by other papers: Scheipel, Mauroner and Baunach [185] describes a method of tracking execution time in RISC-V processors without interfering with the system itself, but this is conceived as a set of performance counters on the chip. This is developed by Delshadtehrani et al. [52] who describe a programmable co-processor for tracking execution time which was crucial for the design of Gouram, which will be seen in Chapter 4 and Appendix A.

2.5 Review Summary

Having considered a large number of papers in the last three sections, we are forced to ask where this leaves us. The state of the art in each of the areas we've looked at appears to be as follows. When considering Cache Replacement Policies, the state of the art is very much as it has been for a long time, in fact much cache research has dried up in recent years as policies are "good enough" [158]. Consequently, especially in the field of medium-sized embedded systems we're looking at, the state of the art is still LRU and approximations to it [44], despite the fact that other policies will perform better, because space is at a premium and approximations of LRU are very easy to implement. Not only that but as many of the papers have shown it's more economical to increase the size of the cache instead of increasing it's complexity in terms of raw performance.

In terms of techniques that are external to the cache the state of the art here, considering the context within which we are working, is that of stride or unit-prefetching, again for reasons of simplicity of implementation, and there are further moves to shift complexity into the compiler and linker at compile-time rather than into the hardware at runtime. In other domains, particularly in servers or HPC other techniques are being employed that include multiple memory fabrics [149] and work like JENGA[207] to define caches in software, but it's difficult to apply that to the embedded context due to the need to apply as much resource as possible to the computation job in hand rather than to simply managing memory, despite the impact that might have on the runtime of the system.

So it would appear our goal of reducing latency appears to be blocked along the many paths we have considered. Solutions which attempt to improve the cache, via cache policy, are bounded above by `OPT` as shown in Figure 2.6, which does not itself reduce misses to 0, due to compulsory misses and a finite cache capacity. Solutions that target the cache architecture are fine in their way, but do not address the basic problem that they are trying to make decisions with a paltry amount of information because of the need to decouple all these systems from each other. Certain papers propose integrating the cache and memory controllers, for example [200], but there is a lack of integration across all components of the system, from the compiler to the memory hardware, that would give the information necessary to make better decisions.

Moving outside the cache, prefetching seems like a sensible way to solve these problems, and could be added on to almost any other solution, but its failures come down again to the lack of information available to the prefetching unit. As a result, most prefetching comes down to recognising patterns or attempting to derive the future from the past. Whilst this is good, it is not foolproof, especially as the diversity of applications increases. Scheduling and program transformation also look promising, but the inability to incorporate dynamic information limits their utility to what can be predicted statically, which is a relatively small amount of information.

2.5.1 Potential for the Application of Tracing

Hence, it would appear that there is little more that can be done, other than tinker at the edges with cache policies or prefetch units and hope that the processor-memory gap shrinks of its own accord. But what if there was a way to address the lack of information that is at the heart of the problems we have seen? What if we could fuse together prefetching and cache policy with dynamic information gathered from traces? Caches could act preemptively to fetch data they know might be required, utilising existing slack in programs to improve their overall runtime by better hiding the latency that exists. This is the basis of Trace Assisted Caching which will be explored in much more detail in the next chapter.

3 Trace Assisted Caching

Having now considered the literature we move on to a theoretical explanation of Trace Assisted Caching, the new technique this thesis proposes. We will start by exploring the motivation for this new technique. We follow that with a theoretical, processor agnostic, description of how it works and then conclude with a section that explains how this technique will bring benefit to hardware that implements this system.

3.1 Motivation

If we return to the central question of this thesis, we are trying to understand how memory latency can be reduced to reduce overall program runtime. However, the results of the literature review in the previous chapter appear to show that though progress has been made in multiple areas there is are still problems to be addressed. Cache policies definitely show some potential, however there is an upper limit on their ability to reduce latency, as demonstrated in Figure 2.6. Some of the gap between this ceiling and the state of the art has been closed with the research we have discussed, but a lot of researchers have effectively abandoned work on cache policies because they are now "good enough" [158, pg. 13 (386)] and the extra effort expended is not commensurate with the gains. To add to that, we cannot implement `OPT` efficiently, because it relies on clairvoyance to achieve its high performance, but even if we could there is a question as to whether we should. `OPT`, whilst the most effective cache policy from the point of view of miss rate, does not boast a 100% hit rate, because it is essentially reactionary. All the cache policies we have considered up to this point are in this vein; they all take action in response to a request for data rather than looking to find leading indicators and taking action on those, meaning that latency in waiting for memory to respond could be overlapped with computation when memory is inactive. Thus, all these reasons necessitate looking beyond cache policies if we want to reduce runtime in a non-piecemeal fashion.

3.1.1 Defining the Key Problems

If we look at the other key areas of the literature we see the same story repeatedly. So where does the root cause lie? It stems from two key problems: the first is that techniques are making decisions based upon limited information and have a limited choice of reactions when a point of decisions arises. In the case of caches, for example, which can only react to information once a miss has happened, the only option they have is to perform a costly memory access. With more information they could react earlier and have more scope to perform other actions, which when used correctly, should lead to lower program runtime. Even with prefetching, which is more pro-active, the problem recurs. Simpler prefetch schemes like OBL or stride are limited in the information they have, or can be undermined by memory padding or poor memory layout choices. Even in more advanced schemes, prefetching often lacks the information it needs, like effective addresses, to make very high quality decisions.

The second key problem is the abstraction presented by most programming languages, that memory and computation instructions take a similar amount of time. Maintaining this abstraction gives us pipeline stalls and the need for caching and memory hierarchies in the first place and leads to a co-assumption that because they take the same length of time they should operate synchronously. Some work, particularly in the realm of prefetching has tried to address this [134, 215, 216], by proposing systems where the memory system can work independently of the computation system. However, even these papers do not go as far as they could, because of a lack of information. In particular, if you cannot predict effective addresses, you are forced to incur lots of stalls in your memory execution, while you wait for non-obvious effective addresses to be calculated. This will increase the level of synchronicity between memory and computation which you are trying to avoid. Otherwise, you accept that you are predicting these memory accesses and have to allow for incorrect predictions and roll-backs. These two points together form the starting point from which the ideas that support Trace Assisted Caching arise.

To take this idea further, let us engage in a thought experiment by relaxing some of the aforementioned assumptions. Let us suppose we have a processor that can record and efficiently query information about any instruction that it has previously executed. It can ask how long a previous instruction execution took, what the effective address of that instruction was, which branch was taken by an instruction and so on. We are not assuming that this processor is perfectly clairvoyant, something that is clearly impossible in general, so it is possible that even with this level of recall the processor will make mistakes. For example, if a processor is in a loop with 50 iterations and it tries to use the information for branch prediction it would be unable to predict the break out of the loop using this method alone. By setting these parameters we keep the processor within the realms of possibility, but open ourselves to the potential for some interesting performance improvements.

Let us imagine a processor with a 4-stage pipeline, namely Fetch (`IF`), Decode (`ID`), Execute (`EX`) and Write-back (`WB`). Let us further suppose that there are no data forwarding mechanisms and for the sake of argument let us assume that fetches from main memory take 5 clock cycles from the submission of the request to the data being available. Now, let us imagine the processor is executing the code fragment in Listing 3.1. The execution in terms of pipeline stages will probably look something like the diagram in 3.1, where it takes 14 clock cycles to execute these 5 relatively simple instructions.

Listing 3.1: This code executes a few computational instructions that have interdependencies.

```
1  addi    t3, 100
2  add     t2, t2, t3
3  lw      t4, -440(s0)
4  mul     t2, t2, t4
5  mv      a0, t2
```

Because of the data dependencies in the processor, this is the limit that we can reach in terms of effectiveness, but now imagine we ran the same program on a processor that had all the advantages previously described. It has a store of information pertinent to all the previous runs of this particular program, can query it efficiently and its memory system can act asynchronously from the computation side of the CPU. With this system we could very easily have the memory query for the most likely memory location and execute it early, in order to avoid a lot of the stalls that we see. With a situation like that we would see something like the diagram in Figure 3.2.

As the caption states this has reduced the runtime of the program by almost 30% and while this experiment is somewhat artificial, it does lead to the question: what allows that to happen? The answer is the availability of effective addresses much earlier than they would usually be available. In Figure 3.1 the effective address of the `LOAD` instruction in Line 3 of Listing 3.1 is only known at clock cycle 4, so a number of stalls are introduced, because of the data dependencies in this part of the program. If the address were known earlier, then the request could also be made earlier, eliminating those stalls and their knock-on effects. However, it is not just a large number of effective addresses that are needed. There also needs to be a way to

Instruction	**Pipeline Stage**													
ADDI	IF	ID	EX	WB										
ADD		IF	ID	EX	WB									
LW			IF	ID	EX	EX	EX	EX	EX	WB				
MUL				IF	ID	STALL	STALL	STALL	STALL	STALL	EX	WB		
MV					IF	STALL	STALL	STALL	STALL	STALL	ID	STALL	EX	WB
Clock Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Memory System					PR	PR	PR	PR	AV					

Figure 3.1: The pipeline execution continues as normal until clock cycle 4 where the `LW` instruction enters its Execution phase. Because loading a value from main memory takes 5 clock cycles this extra latency is not only added to the `LW`, but also to the `MUL` and `MV` instructions. Overall this leads to a program runtime of 14 clock cycles, when the optimal time, assuming no pipeline stalls, would be 8. To further clarify, the memory system row contains `PR` and `AV` standing for Processing and Available respectively.

Instruction	**Pipeline Stage**									
ADDI	IF	ID	EX	WB						
ADD		IF	ID	EX	WB					
LW			IF	ID	EX	WB				
MUL				IF	ID	STALL	EX	WB		
MV					IF	STALL	ID	STALL	EX	WB
Clock Cycle	0	1	2	3	4	5	6	7	8	9
Memory System	PR	PR	PR	PR	AV					

Figure 3.2: If we assume the preemptive action to take is the one at line 3 in Listing 3.1, then because the effective address can be known at clock cycle 1, the memory system can begin this load much earlier. As a consequence, the stalls associated with performing the memory operation are eliminated from the program's runtime, reducing the overall runtime by 4 clock cycles. A decrease of nearly 30%.

tie those effective addresses to the instructions that create them, because we need to be able to understand what is safe and unsafe to do preemptively. For example, if there was a preemptive `LOAD` into an address that was expecting a `STORE` then the loaded value will be overwritten by the preemptive `LOAD`. Therefore, when the processor actually executes the `LOAD`, it will read an incorrect value. Consequently, we need both pieces of information, effective address and instruction, in order to understand how and whether preemptive action is possible.

3.1.2 Exploring the Design Space

Now we have an idea of the information we require to make the solution viable, several decisions have to be made before we can start exploring exactly how this system is designed. For example, if we need to capture this data when should it be captured? We have effectively three choices, design time, compile time or runtime.

When Should Information Be Captured?

If we first consider design time we almost immediately start to run into problems. Firstly there are limits to the amount of information that can be known statically about a program in this context. For example, if there is an instruction of the form `LW t4, -300(t1)`. To calculate this effective address we need the data that is in `t1`, but of course that cannot become available until the previous instruction has completed its execution and `t1` could hold any value. Therefore, predicting it is next to impossible without simulating the entire program. Now on the surface this doesn't seem to be a problem, but in order for that simulation to be entirely accurate it would be necessary to have accurate simulation models for every component in the system that's involved in deciding the effective address. For example, we would need accurate models of the CPU, the memory controller, the memory hardware, the OS and many more components besides. These models are often not available and sometimes have non-deterministic elements. Consequently getting to the dynamic information that we need to move this approach beyond the application of static analysis is very difficult or even impossible to get in this context.

Moving to compile time, we have a similar problem in that it's possible to write programs that are very difficult to analyse without also turning the compiler into a simulator. Furthermore, if we attempt to make this approach work at compile time, we risk this approach becoming very implementation specific. It may indeed be possible to augment compilers with information about the platform they are compiling for beyond that which they currently have. Say that it was known that variables were going to be allocated sequentially from address `0x200`, then this would be information that could be encoded into the compiler to work out more of the effective addresses than is possible statically. However this would then force the use of a particular compiler that had this knowledge and would mean changing any other component in the toolchain would require changes to the compiler as well. It also could limit the applicability of this research to one very specific toolchain which is not desirable.

On a related note, a trade-off we would have to make with any system like this is that we will be sacrificing space to store the extra information that we need to make this approach work. We could try and achieve this by adding extra markers to the compiled assembly code, but this could have the same problem as Software-based Prefetching where the code-size may well increase beyond the capacity of the system. In that case we have to accept that the data will need to be stored elsewhere which would mean we could not take a Commercial Off-The-Shelf (COTS) compiler and start using it, we would again be forcing the use of a particular compiler which had the ability to extract and store this extra data, further constraining any potential users and the applicability of this research.

This leads us naturally to considering how we might do this at runtime, here there are none of the problems of what we can know statically because we can capture dynamic information as it happens if we can record the program somehow. In addition we don't need accurate models because we have the actual system. Now of course this would mean we would require some kind of 'training' phase where the program simply executes and we record information before it can be used, but it may be possible to circumvent this by priming this store of information at design or compile-time so this can be eliminated.

Of course there are further trade-offs we have to make, we're going to have to expend more resources within the system itself to make this happen at run-time, as many systems do not have the capability to capture information at the level of granularity that we require. In addition we need somewhere to store this information, however that would be required in any event if we chose compile-time or design-time. The key trade-off we're going to make is how much of an overhead, in terms of time, is this system going to add? Now clearly the ambition of this system is to convert cache misses into cache hits by overlapping the cache misses as shown in Figures 3.1 and 3.2, and if that is the case it's hoped that this trade-off will be balanced by the commensurate gains we make. This is key to the entire project, if we cannot find a way to ensure that the overheads do not exceed the gains then this solution is simply not feasible.

So it seems as though runtime gives us the most capacity for improvement over static techniques while also presenting us with trade-offs that we will have to keep a close eye on if we want to ensure this method is a success. The next question is therefore exactly where this system should be integrated, can we keep the chip architecture the same and simply observe external signals or do we need to augment the CPU itself to gain access to the information we need?

Where should this system be integrated?

As we have now decided on runtime as the best time to capture this information where is it best to site the mechanisms we're planning to use? Can we simply add this component off-chip with respect to the CPU or are we going to need to update the CPU structure in order to make this work? Initial experiments focused on trying to place a component outside of the CPU that had knowledge of the instructions the processor was receiving, and the data requests the processor was producing by monitoring the instruction and data lines the CPU exposed. However this ran into two problems very quickly. The first was that it was very difficult to match instructions that were sent to the processor with the memory addresses that were requested. This is explained further in Appendix A, but briefly there are situations where the processor will fetch an instruction from memory, but not actually execute it. If this instruction is an instruction with memory consequences it can lead to a situation where each of the inferences as to which instruction matches which effective address are off-by-one or more if this happens repeatedly. This has no solution because there is no external signal given, per instruction, as to whether it is actually executed or not, at least not within the processor models considered.

With that in mind this approach has fundamental problems, as we are relying on the fact that we can successfully associate memory addresses with instructions in an accurate way. As a result we are forced to the conclusion that we cannot treat the processor as a black-box that we can simply measure inputs and outputs from. We need to expose signals that were previously hidden in order to be able to successfully associate memory instructions with effective addresses in order for this approach to successfully work.

With all this in mind now that we are now looking at a system that works at run-time and will have to update an existing CPU architecture in order to work successfully. So where does this leave us? We know we are now working at runtime and need a way to record program interactions so we can extract the required information from them. This naturally leads us towards tracing as a method of getting the information we require. The next section explores tracing as a solution and show a high-level design for a system that implements it.

3.1.3 Tracing

A program trace is an ordered list of events that track the execution of a particular program from start to finish. What constitutes an event in a trace is very much defined by the granularity of the trace itself. If we are tracing at the level of clock cycles then an event might be a change of signal value, but if we are tracing at the level of instructions then it might only be code branches that constitute an event.

With that in mind, what might we need to record to realise the ambitions in the previous section? We would certainly need to record at the level of instructions in the program, but we would also need to trace the data concerned with the program as well, because it is the effective addresses we need to record for the technique to work. This means that we will have to record at the level of pipeline stages and then aggregate the recordings up to instruction level. If we did not do this, we would have no way to guarantee that the correct instruction matched the correct effective address, as mentioned in the previous section.

However, tracing is not the whole story, we also need some method to store the traces and to query them efficiently. It may be possible to do this in the same memory as the program data, but that will be explored as we design the solution. Further, we also need some way of adding agency to the existing cache, so that it is able to action the memory requests from the trace. This is almost akin to a small processor within the cache, though one that is very limited. There will also need to be some degree of co-ordination between this system and the CPU, otherwise it would be very easy for the state of memory to become incorrect with respect to the program's semantics. In the next section we discuss a high-level design for the system and how it might be implemented.

3.2 High Level Design

Now that we have seen a theoretical description of how Trace Assisted Caching works let us consider a high-level design for the system. Chapter 4 will focus on the implementation, but let us consider how the key pieces in this scheme would work. First, let us imagine a single-core, Harvard Architecture, CPU with an L1 data cache, akin to the medium sized embedded system we presented in Figure 1.2. This cache can be Direct-Mapped or Set-Associative and implements a replacement policy of some kind as appropriate. This can be seen in Figure 3.3.

As alluded to in the previous section, there are two key pieces that form the architecture for Trace Assisted Caching: the trace recorder and the intelligent cache. Each will be

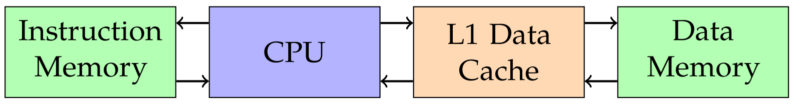


Figure 3.3: The basic design begins with a processor with a Harvard Architecture. This allows the effect of any caching on the data side to be isolated and quantified more easily.

described in turn. We will also describe any changes that would need to be made to the processor to support these new pieces of hardware.

3.2.1 Trace Recorder

The trace recorder would sit attached to the processor and would monitor the internal control signals emitted by the processor. These internal signals would not only be those that manipulated memory, but would initially be every pipeline phase of every instruction the processor executed. This would require changes to the processor, externalising previously internal signals to allow us to track the execution of instructions, as explained in the previous section. A diagram of this can be seen in Figure 3.4. Perhaps it would be simpler to rely on dedicated, preexisting tracing hardware, such as CoreSight[41], but this does not provide the level of introspection to track every pipeline stage. At present CoreSight only provides branch-tracing for programs[41] which does not help us as we're interested in data. Further, we do not want this system to introduce a performance penalty to the running system. Therefore, anything that means the processor has to perform more actions than it would on a normal execution, like outputting to a trace port or adding extra instructions, must be avoided.

The amount of data a trace like this would generate is potentially gargantuan, multiplying the amount which already exists in instruction traces by the length of the pipeline, but there are several reasons why this level of detail is required. The first is that the goal is to link together instructions in the program with the effective addresses they generate. If we record at a coarser granularity, it can be very difficult to associate memory accesses with the instructions that generated them. This is particularly true in very complex programs, unless an Instruction Set Simulator (ISS) is implemented, which can be co-simulated with the actual processor. Recording at the level of pipeline stages and aggregating to instructions later means there is no ambiguity. This also means that we have to take into account branching behaviour and stalls in our recorder, and this will be explored further in Appendix A. This would give us a complete trace, at the level of pipeline stages, for any program that runs on the processor. In addition tracing at the level of pipeline stages means that we are recording a serialised data

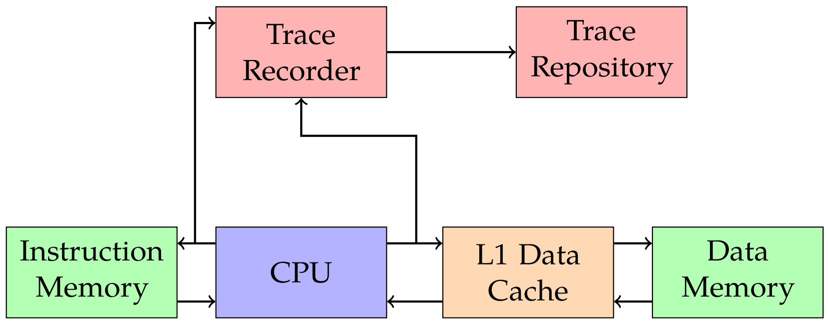
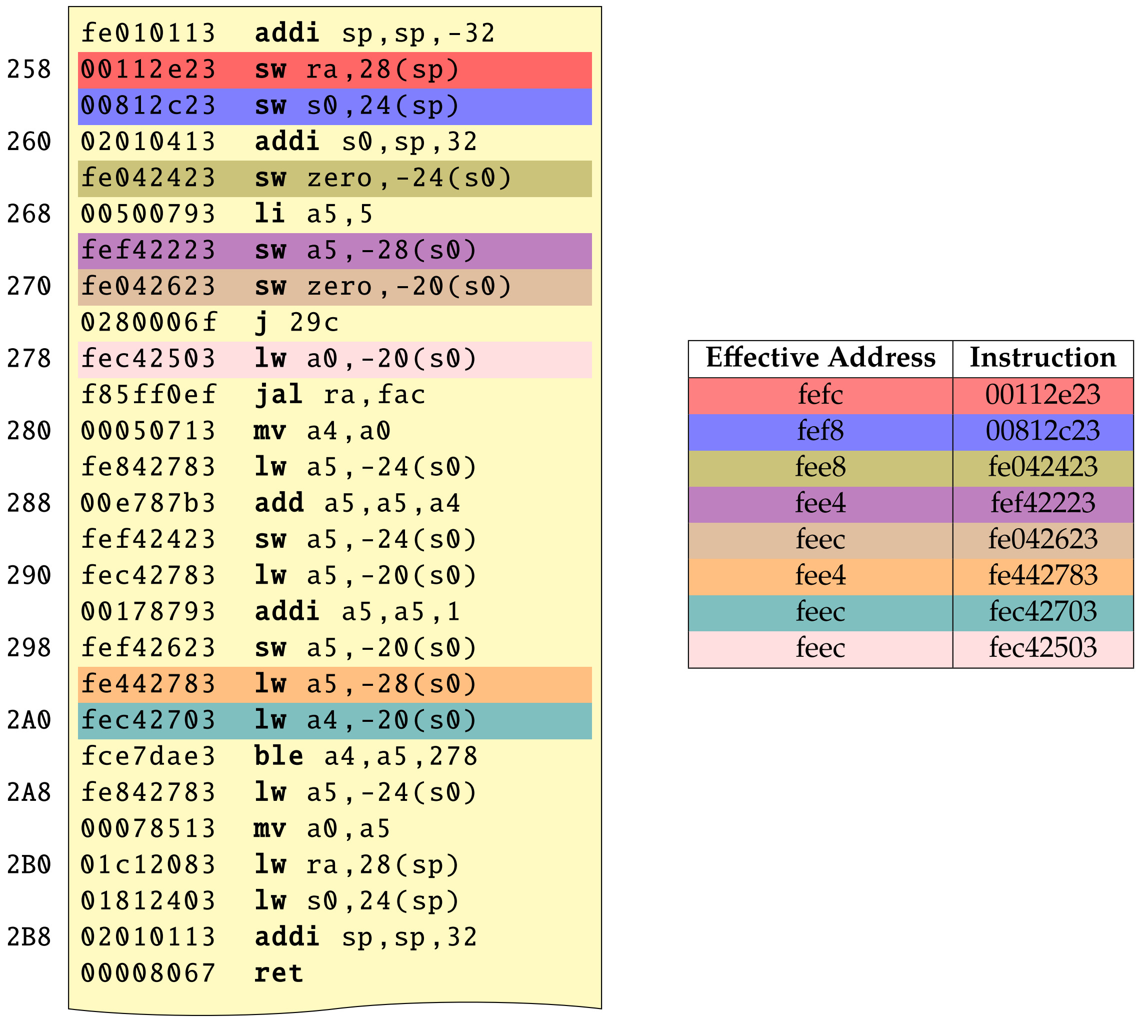


Figure 3.4: The trace recorder takes in information from both the instruction stream and the data stream to produce a stream of pipeline stages bundled into instructions. It will also consider signals that were previously internal to the processor so it can track the execution of pipeline stages. This is stored in the trace repository after filtering to remove all non-memory instructions.

stream. Thus, there is no need to do any de-multiplexing or complex analysis to associate memory operations with instructions, at the level of pipeline stages there can only be one instruction per stage at a time.

Now we have this data it is imperative that we filter it down in some way. Even for a small processor running a short program, gigabytes of data could be generated and this is simply not viable to store or query efficiently, especially in the context of an embedded processor. The first thing we can do is to throw away the details of each individual pipeline stage. These are not needed to instruct the memory system directly. We use them as a way to ensure the correct ordering of the memory instructions and the correct association of effective addresses. The second piece of filtering is to remove any instructions that do not have memory implications. This could be as much as 60% of the instructions captured. We are fortunate when using a processor like the `RI5CY` that only `LOAD` and `STORE` can access memory and these are easily identified by their opcodes. In other architectures this may not be the case, especially if the architecture is more Complex Instruction Set Computer (CISC)-like. By filtering and aggregating using these two criteria we can produce a trace that is an ordered list of instructions linked to effective memory addresses. An example of this is shown in Figure 3.5.

A further point to add here regards the trade-off between space and accuracy. Under the scheme as currently described we veer very much towards the accuracy end of the spectrum with regards to the data that is stored. The trace we store is a complete record of all the memory accesses made by the processor during the execution of the program, so it is completely accurate in the information it records. There is certainly an argument to say that all this data is not necessary and there are probably refinements and simplifications to be made, such that only the most pertinent data is retained, as this will give the largest return on investment. This is not something we have actively explored in this thesis. However, the topic will be returned to in Chapter 6.



```
    fe010113  addi sp,sp,-32
258 00112e23  sw ra,28(sp)
    00812c23  sw s0,24(sp)
260 02010413  addi s0,sp,32
    fe042423  sw zero,-24(s0)
268 00500793  li a5,5
    fef42223  sw a5,-28(s0)
270 fe042623  sw zero,-20(s0)
    0280006f  j 29c
278 fec42503  lw a0,-20(s0)
    f85ff0ef  jal ra,fac
280 00050713  mv a4,a0
    fe842783  lw a5,-24(s0)
288 00e787b3  add a5,a5,a4
    fef42423  sw a5,-24(s0)
290 fec42783  lw a5,-20(s0)
    00178793  addi a5,a5,1
298 fef42623  sw a5,-20(s0)
    fe442783  lw a5,-28(s0)
2A0 fec42703  lw a4,-20(s0)
    fce7dae3  ble a4,a5,278
2A8 fe842783  lw a5,-24(s0)
    00078513  mv a0,a5
2B0 01c12083  lw ra,28(sp)
    01812403  lw s0,24(sp)
2B8 02010113  addi sp,sp,32
    00008067  ret
```

Effective Address	**Instruction**
fefc	00112e23
fef8	00812c23
fee8	fe042423
fee4	fef42223
feec	fe042623
fee4	fe442783
feec	fec42703
feec	fec42503

Figure 3.5: On the left a fragment of the disassembled version of the benchmark `fac.c` can be seen, which is part of the Mälardalen Benchmark Suite [76]. On the right the trace that relates to this part of the file can be seen, with effective addresses next to each memory instruction. The colours indicate which trace element maps to each instruction in the disassembled file and it can be seen that the trace follows the execution of the program, following jumps and branches as they arise.

We now need to turn our attention to a second problem, how should this data be stored? We cannot employ the same data memory as the running processor, because it could affect the retrieval of actual program data. Therefore, we need a separate memory store to keep these traces accessible to the cache. This takes the form of a trace repository that is stored in a physically separate memory, so it can be queried by the cache on each clock cycle when the program is run for a second time. It is also possible to seed this trace repository with data from a previous run if we did not want to have to run the program once in a 'training phase'. Issues such as where this repository is to be stored, how it might be fetched from efficiently and how it might be built up iteratively over multiple runs will be dealt with in future sections.

So how might all this work in practice? The program would execute as normal, though the cache would be bypassed during the recording phase. This is so that we record the 'worst case' behaviour, rather than that caused by the cache. While the CPU executes, the control signals it generates are tracked and recorded by the trace recorder, filtered as the recording occurs, and sent to the trace repository. Once the program reaches its end, marked by it entering the processor's trap state, the trace recorder will stop recording. The data held in the repository is then available for the second and further runs of the program.

3.2.2 Intelligent Cache & Memory System

Now that we have a highly filtered version of the trace stored we can start to use it to allow the cache and memory system to work asynchronously from the processor. This works in the following way: on every clock cycle, instead of sitting waiting for the processor to issue a command to it, the cache polls the trace repository. The trace repository then sends to the cache the next trace in the stream that it had recorded in the previous run of the program. The cache now has potentially two requests it could service, a preemptive request or an on-demand request. In all cases the on-demand request would take priority, because we do not want to introduce further latency by delaying a request from the processor. If we did allow this, it might be possible for the program to exhibit performance worse than the training run and this is something we wish to avoid at all costs.

If there is no on-demand request, the cache seeks to service the preemptive request and there are four scenarios that could occur, depending on the content of the instruction to be executed. These four scenarios are illustrated in Figure 3.6.

Scenario 1: If the preemptive instruction is a `LOAD` and there is space within the cache then simply execute the instruction.

Scenario 2: If the preemptive instruction is a `STORE` and there is space in the cache then you cannot execute it, because you do not know what data is to be stored there. However, you can reserve the space in the cache to make sure it is not overwritten by another action.

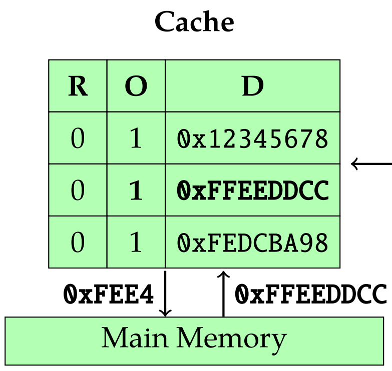
Scenario 3: If it is a `LOAD`, but there is no space in the cache, then perform a write-back and then do the same as in Scenario 1.

Scenario 4: If it is a `STORE` and there is no space, again perform the write-back and then reserve the space.

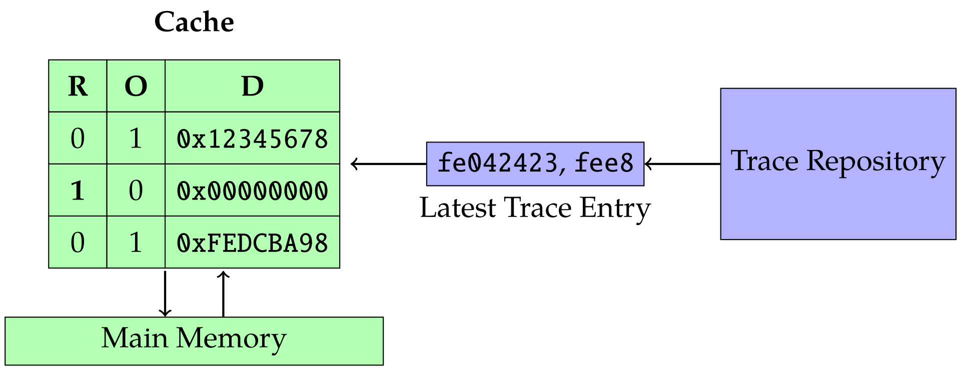
All this means that when the processor catches up to the preemptive action it will either, have the data already, in the case of scenarios 1 or 3, or it will be guaranteed to not have to perform a write-back in the case of scenarios 2 or 4. As a result, latency will decrease, because all these operations will have been done ahead of time.

There are some limitations to this approach of course, most of which relate to how far ahead of the processor the memory system can go. For one thing no attempt is made to re-order the memory operations in the way that an OoO processor would. This is mostly to simplify the implementation and to avoid the complex problems of ensuring consistency over multiple interleavings of the instructions. That being said if an OoO processor was presented with this information it certainly possible that it could make less conservative re-orderings but this purely speculative and potentially something to be explored as future work. As a result of this, there are situations where no more preemptive actions can be taken until the processor has performed some critical action. For example, should a processor issue preemptive instructions, such that it tries to perform a `LOAD` to a location that is being reserved for a `STORE` where the memory addresses are not the same, then it will not take any action until the processor has completed the `STORE`. This is because the cache cannot know in advance what data will be part of the `STORE` operation and therefore cannot writeback that data until it is known. This is shown in Figure 3.7. In this situation the processor will keep polling to discover if the operation has happened and can then act accordingly. There is a potential for future work to consider continuing to execute independent instructions at this point, but that was not considered in this thesis.

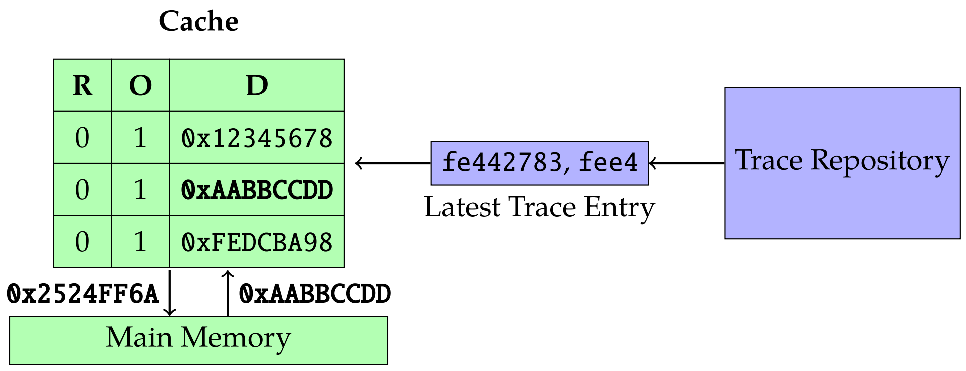
A final point to address is that this scheme is agnostic to the implementation details of the cache that it works with. In this thesis both an 8-way Set-Associative and a Direct-Mapped cache were used, but there is nothing special about these choices. It would be very easy to implement higher associativity caches or larger caches with very few changes to the Trace Assisted mechanism. This also brings up the possibility of varying the replacement policy for the cache but again that was not considered in this thesis.



(a) Scenario 1 - A load where the cache has capacity, the load is simply executed by the cache and stored in the correct location.

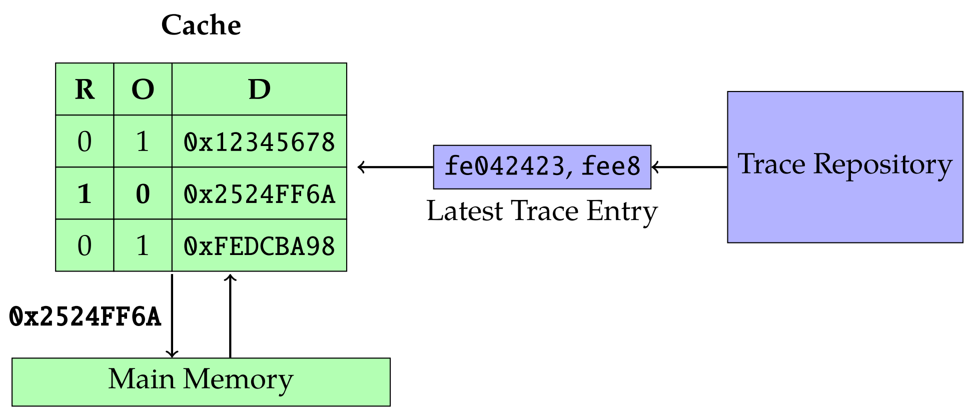


(b) Scenario 2 - A store where the cache has capacity, the store cannot be actioned because the data to store is not known in advance, but the cache element is reserved.



(c) Scenario 3 - If the cache does not have capacity for a load then perform the write-back to memory first and then act in accordance with Scenario 1

Figure 3.6: Illustrations of the four scenarios described. The R, O and D columns above the cache in each diagram stand for Reserved, Occupied and Data respectively. The bold text indicates elements that change throughout the scenario. The cache may also have other elements, such as validity bits, but these are omitted for the sake of clarity and are mostly implementation details. Scenario 4 is printed on the next page.



(d) Scenario 4 - If the cache does not have capacity for a store, then again perform a write-back and proceed as Scenario 2.

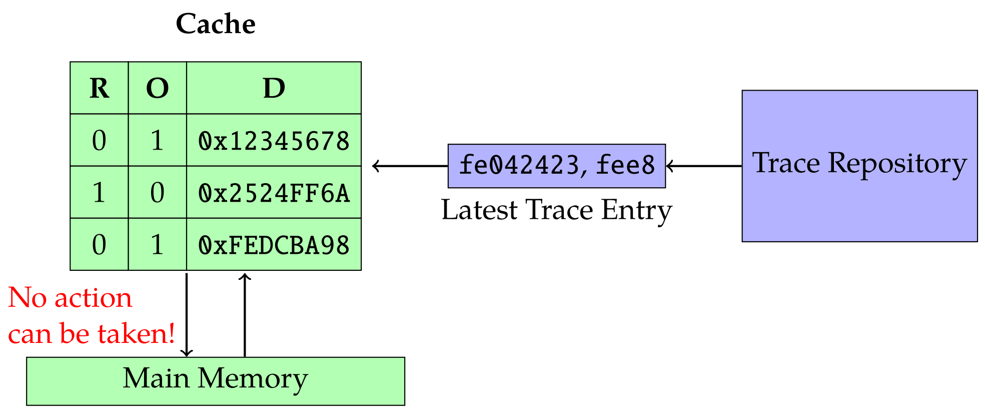


Figure 3.7: In this situation the latest trace entry is attempting a `LOAD` into a location that is reserved for a store. At this point the preemptive execution cannot continue because the `STORE` has not yet happened. The only option is for the preemptive instruction to poll the cache until the `STORE` has happened and then processing can continue.

This is Trace Assisted Caching, recording the traces of programs, filtering them and then allowing the cache to run-ahead of the processor, leveraging the trace information to preemptively perform memory operations. How do we know whether this is going to be a success? And, how do we know that we are not going to introduce performance regressions from this technique? The next section answers these questions.

3.3 Justification of Success

Now we understand how Trace Assisted Caching works at a high level we need to look more closely into how we expect to benefit from this system. It is potentially a high investment in extra hardware, as a trace recorder is required, as well as the changes to the cache and the trace repository. Hence, there needs to be a commensurate benefit in runtime reduction to justify the expense. We have already seen in Figures 3.1 and 3.2 the method by which we should benefit, by overlapping long and short latency instructions to remove potential pipeline stalls. This has benefits through the whole program and is additive to any other techniques that might be used to reduce runtime, but what about performance degradation? Without careful thought it can be easy to get into the situation where trying to improve performance is worse than not having used the technique at all. How can we be sure that this technique will not introduce a performance penalty that will eliminate all the gains it might otherwise have made?

3.3.1 Protections against Performance Degradation

Let us consider the three scenarios that might arise when we try to execute a preemptive instruction before the processor requires the data it references. There are three scenarios, detailed below:

Situation 1 The memory instruction is predicted far enough ahead of time such that the data is ready in the cache before the processor requests it. This is the situation depicted in Figure 3.2.

Situation 2 The memory instruction is predicted ahead of its execution, but not far enough to be ready when the processor requests it. In this situation, there will still be some benefit, but the pipeline will still stall. This is depicted in Figure 3.8.

Situation 3 There is not enough time to perform the memory instruction ahead of it being needed. The processor reverts to the original behaviour or the address predicted is incorrect and the time spent preemptively accessing memory is wasted. Here no benefit will be experienced. This is depicted in Figure 3.9.

It's worth pointing out here that due to the constraints mentioned in Section 3.2.2 we can never get into a situation where the preemptive fetch happens 'too early' and is therefore replaced by the cache before it can of use. To take an example suppose we have three `LOAD` instructions `A`, `B` and `C` where `A` and `C` map to the same cache block but `B` doesn't. To get into the situation described above the cache would have to fetch the data at `C` before the processor had completed the load from `A` from its perspective. However that's impossible because when the cache loads into the cache block that could hold `A` or `C` that block is marked as 'in use' so the cache will stall and wait until the processor has completed the `LOAD` from `A`, which will have no memory consequences as the data is already present, before it continues. So there is no need to worry about data becoming invalid once it has been preemptively fetched.

In all of these situations the effect on the length of time taken for the memory operation ranges from nothing, where it is as effective as in the first run of the program, to a memory instruction that has virtually no latency. Because we are not attempting to re-order the memory operations, we are not allowing the processor to overtake the preemptive memory system. Therefore, the only thing it will experience is much faster memory accesses in a transparent way. There is, of course, the chance that no benefit is gained from any of these enhancements, which admittedly questions the case for the economics of this technique, but at the very least this technique is not actively harmful to the program running on it in terms of its runtime.

A further point that must be considered is about programs that are not well-suited for this technique. Specifically programs that repeatedly cause situations as in Figure 3.7, or programs that are heavily computational and so access memory very little. In these cases, we will repeatedly see the kinds of behaviour described in Situation 3, and the technique will have little to no impact on latency. This technique relies on exposing more potential for concurrent memory and processor execution, but this has to be present in the first place in order to be exposed. If there is no scope to allow this interleaving in the first place, there is no improvement to be had. One of the research questions is around identifying sets of circumstances where this technique will perform better, and one of the outcomes of this thesis should be the ability to quantify exactly which types programs will benefit most from this technique. If this is achieved it will be much easier to appropriately target the technique, rather than wasting time trying to speed up fundamentally ill-suited programs.

Instruction	**Pipeline Stage**											
ADDI	IF	ID	EX	WB								
ADD		IF	ID	EX	WB							
LW			IF	ID	EX	EX	EX	WB				
MUL				IF	ID	STALL	STALL	STALL	EX	WB		
MV					IF	STALL	STALL	STALL	ID	STALL	EX	WB
Clock Cycle	0	1	2	3	4	5	6	7	8	9	10	11
Memory System			PR	PR	PR	PR	AV					

Figure 3.8: In this scenario we assume that for some reason the request to memory does not begin until Clock Cycle 2. In that instance we still introduce some stalling, but runtime is still decreased, so there is still a net positive effect. This shows that any advantage gained should be beneficial, accepting that there may be cases where knock-on effects could occur.

Instruction	**Pipeline Stage**													
ADDI	IF	ID	EX	WB										
ADD		IF	ID	EX	WB									
LW			IF	ID	EX	EX	EX	EX	EX	WB				
MUL				IF	ID	STALL	STALL	STALL	STALL	STALL	EX	WB		
MV					IF	STALL	STALL	STALL	STALL	STALL	ID	STALL	EX	WB
Clock Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Memory System PR	PR	PR	PR	AV										

Figure 3.9: In this scenario we start a preemptive memory access in Clock Cycle 0, but when the CPU starts to perform the access in Clock Cycle 3, this is actually not the address the CPU is requesting. As a result execution continues as though no preemptive behaviour had happened. As a consequence, apart from an extra piece of data in the cache. This is no different from a cache miss, so the runtime is made no worse by the preemptive action having taken place. Of course the extra data in the cache could have positive or negative consequences, but these are difficult to quantify and this situation is quite rare in the programs studied.

It would be remiss of us, when considering performance degradation, not to think about what happens in the case of a miss-prediction. An example of this might be if there are memory instructions that rely on data that must be input to the program at the start. Consequently it would be impossible to predict those addresses before the program starts and running the program repeatedly will not generate any useful patterns, because the addresses are input dependent. It's possible to make an argument that extra work at compile-time could resolve this problem, but as explained in Section 3.1.2, even if we used compile time to construct some kind of dependency graph where we could relate each memory access to the input data somehow this still doesn't resolve the problem of methods of accessing memory that obscure the true values, we still have the static analysis problem, see Figure 2.7. Further it's not just the effective address values that could be input dependent but also the whole structure of the program, and this will have an impact on the collected traces. In this work we do not deal explicitly with this problem, however, Chapter 7 lays out the work that would need to done ot begin to address this. For the purposes of this thesis we are going to assume that repeated runs of a program work with the same input data each time for the purposes of simplicity.

This chapter has described a design for what a TAC would look like and examines some of the theoretical hurdles that will have to be considered in order to implement such a scheme. The next chapter deals with the implementation of this scheme in the context of a Field Programmable Gate Array (FPGA) running a soft-core `RISC-V` processor.

Part III

Experiments

4 Implementing the Platform

So far in this thesis we have seen examples of what a system that implements Trace Assisted Caching might look like from a very high level. For the purposes of context, our ambition for this experiment is to design the hardware that would be necessary to implement Trace Assisted Caching in a medium-sized embedded system. To evaluate the efficacy of the design we will then synthesise this design so that it can be realised inside the Xilinx VC707 FPGA [211]. We will then perform measurements on the newly designed hardware, as well as on other simpler designs, to demonstrate the performance increase that is seen as a result of Trace Assisted Caching.

In this chapter we are going to explore the actual implementation of the scheme that was undertaken as part of this thesis. We will first explore some of the pre-existing components that were used. We then follow that by discussing the development of each of the new elements that were necessary to implement the scheme. To conclude we will discuss the resource implications of the chosen implementation.

4.1 Pre-existing Components

When beginning the construction of the hardware necessary to implement Trace Assisted Caching there are several modules that we can re-purpose from other projects. Starting with the processor as a whole, for these experiments we are going to use the `RI5CY` [70] processor from the PULP Foundation which implements the `RISC-V` ISA[221]. The `RI5CY` is a 32-bit, in order processor with a 4-stage pipeline. It supports the RV32I, RV32C and RV32M standards found within the `RISC-V` standard, which allows for integer computation, compressed instructions and integer multiplication and division. There is an optional floating point module, but that will not be included in our instance of the `RI5CY`, because it simplifies tracking the computation of the instructions. A big benefit of the `RI5CY` is that it has a Harvard Architecture, making it ideal for these experiments, because we can isolate the data memory to more accurately quantify the benefit from Trace Assisted Caching. In addition, because the `RISC-V` ISA is based on classic Reduced Instruction Set Computer (RISC) we do not have to be concerned about multiple esoteric addressing modes or microcode, as we might have to with a CISC architecture. To be precise, only `LOAD` and `STORE` are allowed to access memory, which simplifies the task of tracking memory operations significantly.

As the `RI5CY` processor is released under the OpenHardware [144] initiative, all of the source code is publicly available to be scrutinised and adapted if necessary. We will need to make certain changes to the `RI5CY` processor to expose some internal signals, but because we have source code access, and because Xilinx's Vivado Design Suite can synthesise this code into a hardware description for an FPGA, this is relatively straightforward. This will be dealt with more fully in the next section, where we discuss the Trace Recorder. A further benefit of using the `RI5CY` processor is that it has already been integrated into the PULPino [206] SoC which was built using the Xilinx Vivado toolchain. This means there are several ancillary hardware blocks available online that allow the `RI5CY`'s native memory protocol to interface with the Xilinx `AXI` protocol, used to communicate easily with the various hardware blocks on the FPGA. This means we can re-use lots of different `AXI`-based memory implementations, without having to write our own adapter. This adapter is known as a `core2axi` block.

Another important factor is the memory technology we will use to support the processor. Xilinx provides a feature as part of newer versions of Vivado known as XPMs [212]. These XPMs allow us to specify the parameters of memory implementation (size, address width, latency etc.) and then delegate the problem of constructing such a memory system to Vivado's in-built tool set. As memory system design can be one of the more complex areas of design when dealing with FPGAs, and it is not something we are that interested in for these experiments, it makes sense to make use of this facility rather than manually crafting our own memory implementation. We can communicate with this memory system using the `AXI` protocol [10], for which all the interfaces are generated by Vivado. This allows us to specify the instruction memory and data memory separately, in a much simpler way than having to write all the code to implement the `AXI` protocol and manage individual physical memory elements.

Therefore, at this point we have a fully functioning processor, connected to two physically separate memory implementations, generated by the XPMs. Between the processor and the `AXI` ports on the XPM generated memory are two `core2axi` blocks that convert the processors native memory protocol to a set of signals which conform to the `AXI4` standard. A diagram of the current architecture can be seen in Figure 4.1.

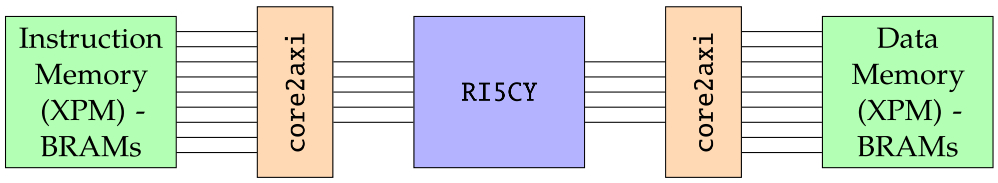


Figure 4.1: The resulting architecture that is used as a starting point for implementing the more complex parts of Trace Assisted Caching. The Memory protocols used are the `RI5CY`'s built in protocol, before the `core2axi` adapter, where the protocol is converted to `AXI4` for communication with the XPMs. The XPMs are implemented as collections of Block Ram Access Memories (BRAMs). The use of BRAMs over the onboard DRAM and the mitigations around latency are explored in Section 5.1.3.

4.2 Trace Recorder (Gouram)

The next stop on the journey to building the hardware platform we require will be to construct the Trace Recorder, which shall be called Gouram, so that that the TAC has a source of trace data. There was some consideration initially of using an existing system, particularly considering the RISC-V debug and trace standard was recently published [153], and a version has been created for the `RI5CY` [163]. However, the implementation offered by the PULP project does not offer data tracing at present, which we require. Therefore, we were forced to construct our own Trace Recorder to work in concert with the TAC. The exact details of the implementation can be found in Appendix A. We will not focus on the details in this section, suffice to say that we can assume we have a reliable method of producing the required trace data. Great effort has been taken to decouple the Trace Recorder from the TAC, such that if an implementation is developed that is more integrated with the CPU or uses fewer resources, it can be used instead, as long as there is interface compatibility.

4.2.1 A Note on Names

As stated in the previous section the Trace Recorder is referred to interchangeably in this thesis as *Gouram*, and it is labelled as such in the source code for the hardware. The Trace Assisted Cache is named *Enokida* and encompasses the traditional cache implementation module and the extra hardware required to add the trace-assistance. Due to implementation requirements a name was also required for the non-trace-assisted version of the cache (in reality a very thin wrapper around a traditional cache implementation) and this is called *Saruyu*. Finally the altered `RI5CY` CPU is referred to as *Godai*. The full hardware platform developed for this solution, including Godai, Enokida and Gouram is referred to collectively as *Kuuga*.

For ease of reference these are all contained within the following table.

Name	**Description**
Kuuga	The entire hardware platform implemented for these experiments, containing a modified CPU (Godai), a trace assisted or standard cache (Enokia/Saruyu) and a Trace Recorder (Gouram).
Gouram	The Trace Recorder, (see Appendix A)
Godai	A modified version of the `RI5CY` CPU that exposes internal signals
Enokida	The trace assisted cache, consisting of a standard cache implementation plus the extra hardware associated with co-ordinating communication with the Trace Repository and CPU memory requests.
Saruyu	The standard cache, a very thin wrapper around the same cache implementations used in Enokida, but without any of the trace assistance hardware.

Table 4.1: This table lists the names of the components created for this thesis and gives a brief description of their function.

4.3 Trace Assisted Cache (Enokida)

Now that we have a process that can produce a large amount of trace data for a standard run of a program, we need a mechanism to act on the information collected, the TAC itself. This section divides into two, because we utilise two different cache architectures to demonstrate the benefits this technique derives, even with very different underlying cache implementations.

The overall functioning of the TAC consists of two modules that are intimately linked, but serve separate functions: the Trace Repository, which is responsible for tracking the state of the trace entries; and, a cache implementation. This architecture is laid out in Figure 4.2, with the connections between them also shown. We will explain each module in turn and then follow with an example that tracks how they link together to give the functionality we require.

4.3.1 Trace Repository

In keeping track of the state of the traces, the Trace Repository operates in one of two states, read or write mode. In write mode the Trace Repository sits and listens to the trace events that are output from Gouram and stores them in BRAM memory in the

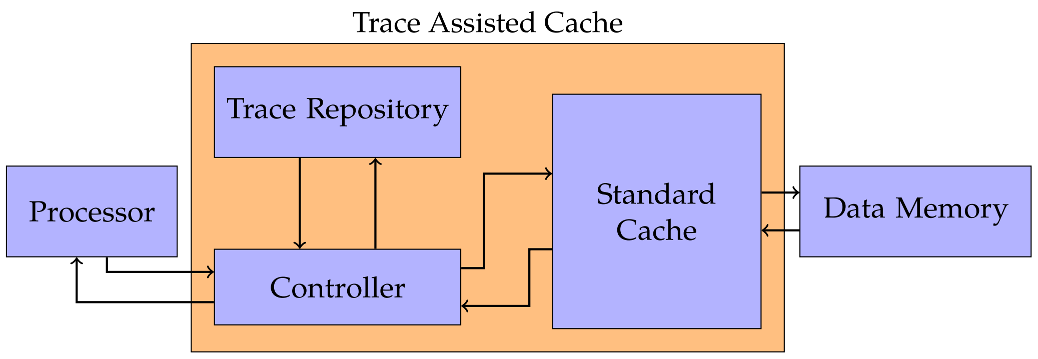


Figure 4.2: A very high level block diagram of the structure of the TAC. The logic for the controller is contained within the same SystemVerilog module as the container, labelled Trace Assisted Cache in the diagram.

order they are received. This means that a non-insignificant amount of memory is required to store the necessary traces.

As will be demonstrated in Section 4.3.2, each trace entry consists of a 48-bits split into a 16-bit address and a 32-bit instruction so we will require 48-bits per memory instruction executed. With this in mind, and due to resource limitations on the FPGA the Trace Repository is limited to $2^{17} = 131072$ entries, which comes to 768KiB. This is much larger than the capacity of the cache itself and in a medium sized embedded system this could be prohibitively large. It's important to remember however that the Trace Repository is not storing the data associated with the memory requests. The fact that this much memory is required by the Trace Repository is one of the limiting factors in the wider applicability of this technique. Chapter 7 discuss some of the methods by which this might be overcome, however these were not implemented in this thesis due to time constraints and the proof-of-concept nature of this solution.

Read Mode

When the Trace Repository is in read mode it follows a very simple state machine, shown in Figure 4.3. As a result, when the Trace Repository is in the `LISTEN_FOR_REQ` state, it is trying to ensure that it is always ready to respond to the request for a trace, should it arise. To that end, every clock cycle, it checks if a trace has been requested and whether it has a trace to return. If the answer to the second part of that question is no, it proceeds to query the underlying memory structure (Transition G) to get a trace back from memory and to be in a state where it would be actionable if a request were called for.

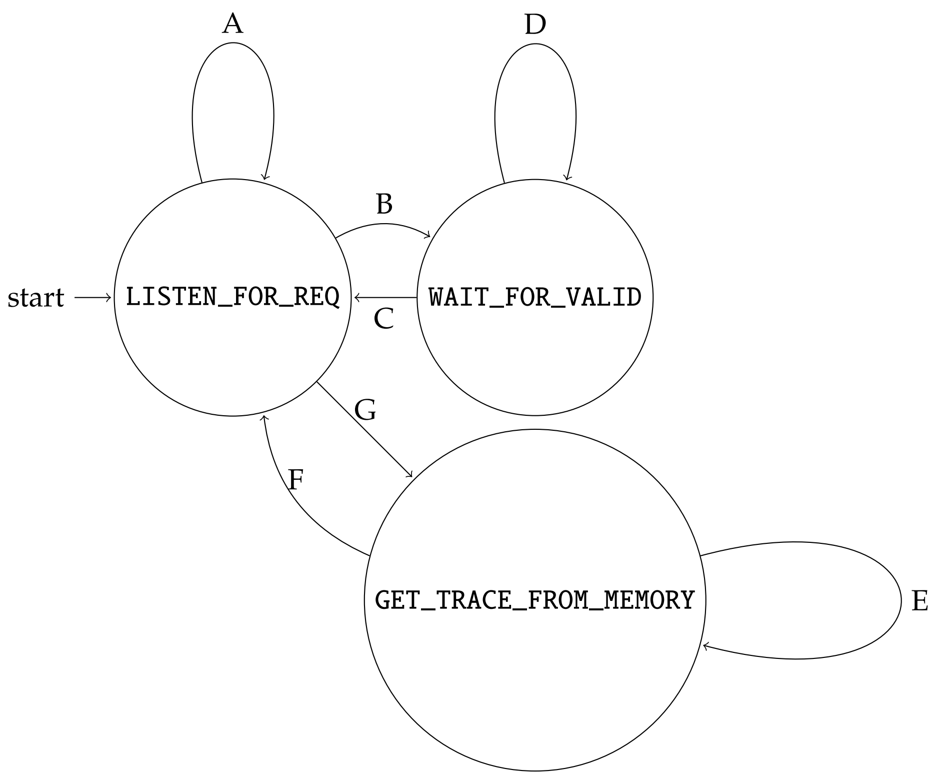


Figure 4.3: This state machine describes the behaviour of the Trace Repository when in read mode. It sits waiting in the `LISTEN_FOR_REQ` state until a request is made to the Trace Repository and then acts accordingly.

Whenever the TAC is idle, i.e. when it is not processing a memory request either from the processor or from the Trace Repository, the control module signals the repository to get the next trace that is not currently being processed. As the process above has been running constantly in the background, let us assume that when the request comes the repository is ready to respond. At this point, the repository checks if the trace it has recalled from memory is executable at the current time, by taking transition B to the `WAIT_FOR_VALID` state. When in this state the Trace Repository queries it's own representation of the state of the cache to check the status of the slot that this piece of data would be mapped to, were it actioned. This will resolve in one of two ways, either the slot is available (it is empty or the entry that is present is not the subject of a pending memory operation) or the slot is not. If the slot is available then obviously we can proceed, because there are no dependencies. If the data has already been processed, then it must have already been used and we are not re-ordering, which means it can be replaced with no ill effects. So in either case the trace entry is returned to the TAC to be acted upon.

However if this is not the case, then there are two further options. If a memory instruction has arrived at the cache from the processor, then this supersedes any trace memory operations, so we set the cancelled signal high to acknowledge this to the top-level of the cache. Alternatively if we cannot execute the instruction because of what is in the cache, we sit in the `WAIT_FOR_VALID` state (transition D), where we repeatedly ask the cache if we can now act on the trace that has been retrieved from memory. This is not optimal because it may be the case that other independent instructions could execute while the dependency is resolved. However as we are not implementing re-ordering this is not a situation we can take advantage of. As the processor is still executing while this happens, every time the question is asked the state of the cache could be quite different, so this is a legitimate question to ask on each clock cycle. Once it is appropriate to proceed, the underlying cache implementation is signaled with the address and the instruction type (`LOAD` or `STORE`) so the cache can act upon the information in the trace. After this, the repository returns to the `LISTEN_FOR_REQ` state (transition C) and the process starts again.

Tracking the Active Set of Memory Instructions

There is a second important function that the repository also fulfils, which is tracking the traces as the memory instructions they denote are enacted in the cache, so that it is clear which are in-flight and which are not. It also has to keep track of the memory instructions executed directly by the processor, so that traces are not returned from the repository if they have already been acted upon. In order to do this, the repository maintains a small circular buffer of in-flight memory operations, known as the active set of memory operations.

Consequently, whenever the cache completes a memory transaction and stores it, it fires a signal called `mark_done` and sets several corresponding flags. There are three separate situations that might arise at this point and they are all handled slightly differently by the repository[1]:

1. The cache has executed a memory instruction from the Trace Repository, but there is still some work to be done by the system for this instruction to be considered as complete. For example, if the repository returns a store operation, it is impossible to predict the value that would need to be stored, so we can reserve the space, but we cannot action the store until the data is available from the processor.

2. This is a counter-point to the previous point, whereby the processor is now completing an instruction that was set up by the repository, for example a

[1]We are making several assumptions here that the order of memory operations is consistent between the trace and the running program. These are addressed in Chapter 7 with some strategies to remedy this dependency, however for the purposes of our experiments, as is explained in Chapter 5, these assumptions hold.

STORE instruction. It should be noted that this also includes the effective NO-OP situation, where the processor requests a LOAD, but the cache has already done this LOAD ahead of time.

3. A memory operation has been completed by the cache in the standard way, as the processor requested the data before the cache had time to request the trace to action it.

In the first situation we add the trace in question to the in-flight set of memory instructions and designate it as processing by advancing a processing pointer. In the second case we designate the trace as complete by advancing a retired pointer and add to the counter of committed memory operations that lets us know when processing is complete. In the final case the repository does both things, and advances both pointers. We then update the `cache_tracker`, which is a data structure designed to preserve the link between the index of the trace in memory that is executing and the cache index that the trace is mapped to. This is very important when asking whether a new trace instruction can be executed or not without having to query the cache directly. Furthermore, in the Set-Associative cache implementation of the Trace Repository, there are extra steps to track the replacement order of the cache sets through a `fifo_tracker` data structure. This is because in the case of the Direct-Mapped cache, the index can be predicted from the effective address of the memory instruction. However, in a Set-Associative world we need to track the state of each of the sets, which in this case is modelled as a series of FIFO queues, so extra tracking data is required here.

A final function the repository serves is to return trace indexes or the address of the trace in the memory that backs the Trace Repository. This is done by linearly searching the active set, which is very small, and the importance of this will become clearer once we explain how the trace assisted cache functions, in the next section.

Liveness & Correctness

A very reasonable question at this point is: how can we guarantee the correct functioning of this state machine? Further, how can we guarantee it will function without deadlocking? In an idealised scenario, we could produce formal proofs for each component, but that was not undertaken as part of this thesis due to the large amount of time spent on the implementation of the platform. However, taking each state in turn, there are several arguments we can make that give very strong indications that the state machine in Figure 4.3 possesses the properties of liveness and correctness.

Listing 4.1: This SystemVerilog code details the `LISTEN_FOR_REQ` state's operation inside of the Trace Repository's state machine.

```
LISTEN_FOR_REQ:
begin
    automatic bit next_trace_ready = (action_pointer + 1
       == last_addr) && trace_valid;
    if (next_trace_ready && trace_req)
    begin
        if (committed_counter == capture_pointer+1)
           processing_complete <= 1'b1;
        else
        begin
            next_available <= action_pointer+1;
            if (can_next_available_be_executed(trace.
               mem_addr))
            begin
                trace_out <= trace;
                trace_index_o <= action_pointer+1;
                entry_valid <= 1'b1;
                state <= LISTEN_FOR_REQ;
            end
            else if (cancel)  cancelled <= 1'b1;
            else state <= WAIT_FOR_VALID;
        end
    end
    else
    begin
        entry_valid <= 1'b0;
        cancelled <= 1'b0;
        if (!next_trace_ready)
        begin
            trace_valid <= 1'b0;
            trace_entries_addr_i <= action_pointer+1;
            trace_entries_ena_i <= 1'b1;
            last_addr <= action_pointer+1;
            latency_counter <= READ_LATENCY;
            state <= GET_TRACE_FROM_MEMORY;
        end
        end
    end
```

Initially there will be no data retrieved from memory so `next_trace_ready` will be 0. As a result, it will take the `else` branch of the `if` statement and its state will change to `GET_TRACE_FROM_MEMORY`. There is no chance of deadlock at this stage, because if `next_trace_ready` is 0, and `next_trace_ready` is declared as a `bit` value, so can only be 0 or 1, it forces the state machine into the `GET_TRACE_FROM_MEMORY` state.

Listing 4.2: This SystemVerilog code details the `GET_TRACE_FROM_MEMORY` state's operation inside the Trace Repository's state machine.

```
GET_TRACE_FROM_MEMORY:
begin
    if (latency_counter > 0) latency_counter <= latency_counter-1;
    else
    begin
        state <= LISTEN_FOR_REQ;
        trace.mem_addr <= trace_entries_data_o[DATA_ADDR_WIDTH+DATA_DATA_WIDTH-1:DATA_DATA_WIDTH];
        trace.instruction <= trace_entries_data_o[DATA_DATA_WIDTH:0];
        trace_valid <= 1'b1;
    end
end
```

Listing 4.3: This SystemVerilog code details the parameters passed to the XPM that backs the Trace Repository showing how the latency is controlled by the `localparam` on line 64 being passed to the parameter in line 78.

```
64  localparam READ_LATENCY = 1;
65
66  xpm_memory_spram #(
67    .ADDR_WIDTH_A($clog2(TRACE_ENTRIES)),              // DECIMAL
68    .AUTO_SLEEP_TIME(0),            // DECIMAL
```

```
    .BYTE_WRITE_WIDTH_A((DATA_ADDR_WIDTH + DATA_DATA_WIDTH)),        // DECIMAL
    .ECC_MODE("no_ecc"),            // String
    .MEMORY_INIT_FILE("none"),      // String
    .MEMORY_INIT_PARAM("0"),        // String
    .MEMORY_OPTIMIZATION("true"),   // String
    .MEMORY_PRIMITIVE("block"),      // String
    .MEMORY_SIZE((DATA_ADDR_WIDTH + DATA_DATA_WIDTH)*TRACE_ENTRIES),             // DECIMAL
    .MESSAGE_CONTROL(0),            // DECIMAL
    .READ_DATA_WIDTH_A(DATA_ADDR_WIDTH + DATA_DATA_WIDTH),          // DECIMAL
    .READ_LATENCY_A(READ_LATENCY),            // DECIMAL
    .READ_RESET_VALUE_A("FF"),       // String
    .USE_MEM_INIT(0),               // DECIMAL
    .WAKEUP_TIME("disable_sleep"),  // String
    .WRITE_DATA_WIDTH_A(DATA_ADDR_WIDTH + DATA_DATA_WIDTH),         // DECIMAL
    .WRITE_MODE_A("no_change")     // String
  )
```

Once in the GET_TRACE_FROM_MEMORY state (Listing 4.2) transition E will be taken multiple times until the latency_counter variable is 0. This must occur as the READ_LATENCY parameter is set to a positive value, as can be seen in Listing 4.3. It would also be impossible for this to be set to a negative value, because the XPM declaration that can also be seen in Listing 4.3 will throw an exception if READ_LATENCY is set to a negative value and the code will not synthesise, so we will not deadlock here. Once in the else branch we can guarantee that the information will be ready to read from memory, because we are targeting BRAMs which require [212] a latency of 1 to be set for READ_LATENCY_A. This will cause us to take transition E once; after that the data will be ready, because we have spent the 1 cycle required to retrieve it. Again, as we are assuming the correct functioning of the XPM. Therefore, we can assume we will receive the correct data once we read it out of the output ports of the XPM. Once that is complete, we will return to the LISTEN_FOR_REQ state via transition F, but with entry_valid now set to 1.

So far we have not observed a situation where a deadlock is possible and we can guarantee that we will have correct data at each phase. Now back into LISTEN_FOR_REQ, we can guarantee that last_addr will equal action_pointer + 1, because it will have been set as such the last time we were in this state. Consequently, next_trace_ready will become 1. Now, if `trace_req` is not high, then the `else` branch of the `if` statement will be taken, but because `next_trace_ready` is set to 1, transition A will be taken rather than G.

At this point one of two things will happen: either the processor will continue to process computation instructions, so the TAC will be allowed to send the `trace_req` signal, as previously described; or, the processor will complete the memory operation referred to by the trace that has been fetched. If the latter happens then the `action_pointer` will be advanced, so `next_trace_ready` will become 0 and the same process we have just seen will continue.

However, in the former case, execution will advance to the first branch of the `if` statement. Let us assume that it is not the last trace to be executed, otherwise `processing_complete` would be set to 1. This would lead to `trace_req` never being set to 1 again. Thus, transition A would always be taken, and the state machine would have entered a trap state. In the case that it is not the last trace, the function `can_next_available_be_executed` will be computed. This function has no loops, so must return a result. Assuming the TAC has reported correctly thus far, it will have an accurate model of the cache to query. Consequently, an answer will be provided. If the answer is positive, the trace will be output as expected, if not transition B will be taken because the trace cannot be actioned without potentially changing the semantics of the program.

Listing 4.4: This SystemVerilog code details the `WAIT_FOR_VALID` state's operation inside of the Trace Repository's state machine

```
WAIT_FOR_VALID:
begin
    if (can_next_available_be_executed(trace.mem_addr))
    begin
         trace_out <= trace;
         trace_index_o <= next_available;
         entry_valid <= 1'b1;
         state <= LISTEN_FOR_REQ;
    end
    else if (cancel)
    begin
         cancelled <= 1'b1;
         state <= LISTEN_FOR_REQ;
    end
end
```

At this point, it is natural to ask the question, could we get stuck in the `WAIT_FOR_VALID` state? If we consider the code in Listing 4.4, then we see this is impossible, because on every transition it asks whether the instruction can be executed. Again, assuming the TAC is responding correctly, then the cache model must be accurate. If we now consider the processor, it must be executing an instruction that is before the one that has been retrieved from the repository or in fact is the instruction retrieved from the repository. This is because the `action_pointer` that governs where the next trace is to be retrieved from is updated on every memory access, be it preemptive or from the processor. Therefore, either the trace retrieved is the current instruction being executed or is ahead of the processor by some distance.

If it is the case that the processor is executing an instruction that is behind the one retrieved, then we know that it must action all the memory instructions between the one it is currently executing and the one that is waiting to be valid. Consequently, the instruction we are waiting for will either eventually be executable, because the memory instruction that blocks it must be executed by the processor, or the cancel signal will be fired, which will mean transition C will be taken and further progress can be made. Once we are back in the `LISTEN_FOR_REQ` state, `trace_req` will be 0, because the TAC sets it as such once it has received the trace from the Trace Repository. Therefore, transition A or G will be taken and the process begins again.

Consequently, there does not appear to be a way for the machine to deadlock. It would also appear to be correct from the point of view of returning the data we require as we require it. Clearly this is not a formal proof, but it should suffice for the Proof-of-Concept nature of this thesis. Any further work that expands the role of the Trace Repository would be well advised to invest time in a formal proof. One final point is that this discussion is based on the assumption that the memory instructions are the same between the first and second run of the program, which is true for the benchmarks studied in this thesis.

4.3.2 Trace Assisted Cache (Enokida)

So now we understand the Trace Repository, our attention can turn to the TAC. Here all of the work we have done comes together and we see our goal of a cache that can act preemptively, backed by the knowledge from the Trace Repository, has been realised. Again, there are small differences between the way the cache acts if it is backed by a Direct-Mapped, as opposed to a Set-Associative, but these are minor and will be flagged as we explore the implementation.

Operation of the Cache

In a similar manner to the Trace Repository the cache can effectively be in two modes. These are either a pass-through mode, where it delegates the memory calls through directly to the memory system or an active mode where it uses the trace information to make preemptive decisions. The first mode is used when the Trace Repository is filling up with data, typically in the first execution of a program. This is encompassed by the `ID`, `CPG` and `CPR` states in Figure 4.4. The second mode is then used to act on that data.

In the second mode the cache again acts as a relatively complex state machine, as in Figure 4.4. We will explore each state in turn and then present an example which demonstrates how all of these pieces fit together to give the complete implementation we are looking for. To begin, Enokida starts in the `IDLE/ID` state, where it resets various variables and sends a request to the Trace Repository, enabling a trace to be ready in the next state, should it be possible to act on it. This is dealt with as in Section 4.3.1. After that, it transitions (G) into the `MAKE_REQ_TO_CACHE/MRC` state, where the address that is being requested is checked against the cache to see if a write-back is necessary and if the entry is valid. This has to be checked at this point, because the CPU memory protocol only guarantees the correctness of the address at this point in time. Thus, requesting this data early is important.

Once that has been achieved, in the next clock cycle, Enokida makes a decision as to whether to action the trace it recalled from the Trace Repository or whether to act on a request coming from the CPU directly. As our intended result is not to introduce more latency to existing memory operations, we cause the CPU operations to take priority and so cancel the preemptive operation if both are available at the same time. This is communicated back to the Trace Repository as described in Section 4.3.1. Once this has been decided, the relevant data is bundled into a request to the cache implementation and this is sent to the cache. We then transition (I) into the `CACHE_HIT_GNT/CHG` state, where several different things might happen.

In the first case if we are acting on a trace from the Trace Repository, it may be the case that the data that is already present in the cache at the location we would need is not yet ready to be written back. For example, it may be the subject of a `STORE` or a `LOAD` may be in progress. If that is the case we have no choice but to wait for the CPU to finish with the data. Consequently the cache goes into a `SLEEP` state (transition K) until such time as another memory operation from the CPU arrives and unblocks the preemptive operation (transition ζ). If that is not the case then we also need to check if we are in the position where there is nothing to be done, so that we can move on. This can sometimes happen if we are dealing with a `STORE` operation from the Trace

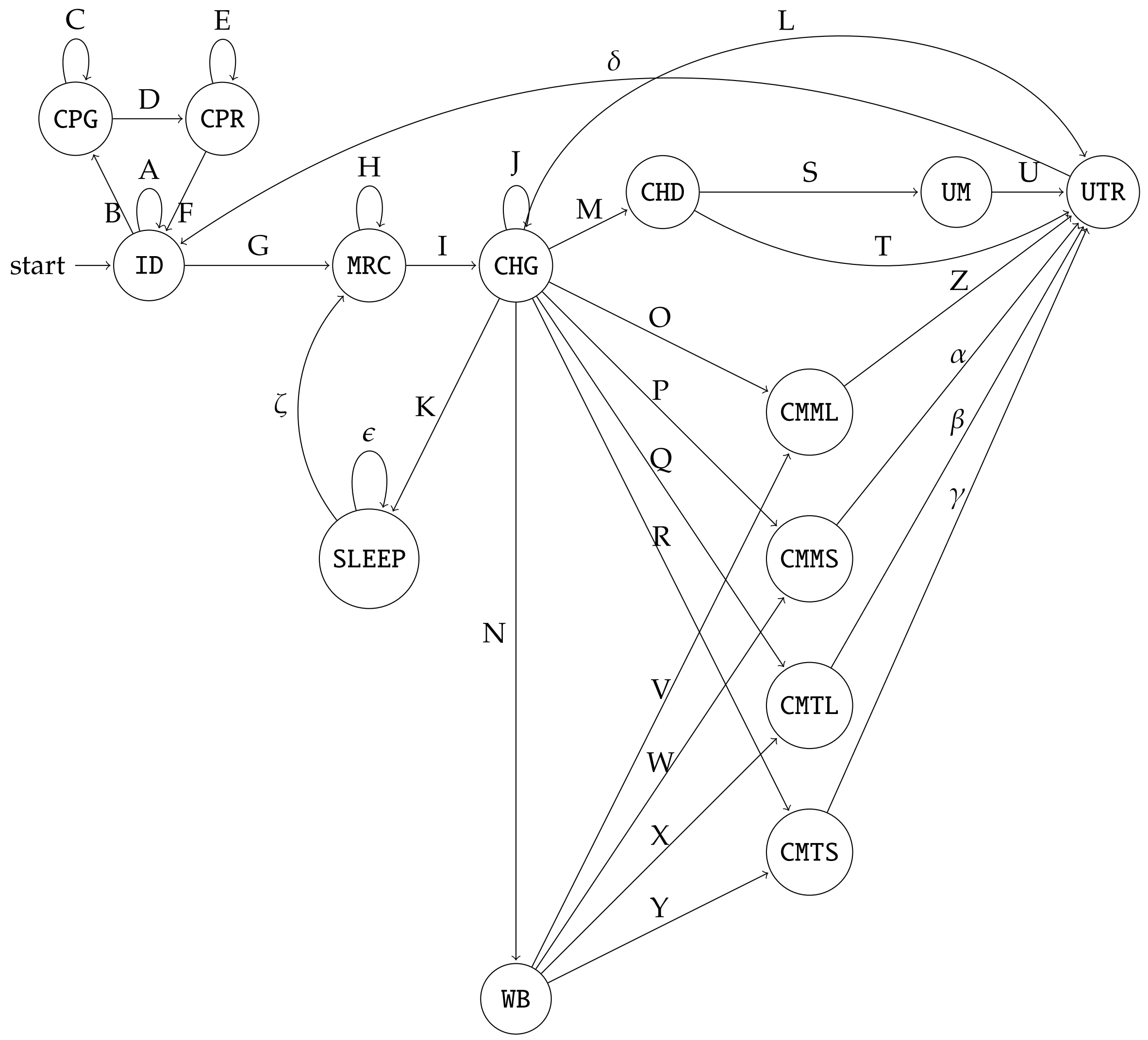


Figure 4.4: This state machine details the operation of the trace assisted cache. The names of the states are abbreviated to allow a more concise presentation of the diagram and some states have been merged for similar reasons.

Repository and there is no need to write anything back from the cache to memory. In that case, we set the `processing` flag high and advance to `UPDATE_TRACE_REPO/UTR` via transition L. Finally, if neither of those situations is in play, the machine wait for the cache to respond to our request (transition J).

Once it has responded, we will either see a cache hit or a miss. In the case of a hit, if the instruction was a `LOAD` we can transition to `CACHE_HIT_DATA` (via M), where we manipulate the memory protocol responses to make it appear as though a memory transaction has occurred very quickly and then proceed to update `UPDATE_MAPPING/UM` via transition S. In this state the Trace Repository is queried to find the index of the trace that has just executed. This is then placed into a map that is indexed by the cache index, so Enokida has knowledge of where the data related to each trace is actually stored in the cache. After this, the Trace Repository is updated in `UPDATE_TRACE_REPO`, via transition U. Alternatively if it is a `STORE`, we mark the cache location as reserved and then transition to `UPDATE_TRACE_REPO`.

Alternatively, if our request to the cache comes back a miss, then we need to write-back the data that is already present. Therefore, we transition to `SERVICE_WB_WAIT_GNT` and then `SERVICE_WB_WAIT_RVALID` (all encapsulated in the `WB` state and reached via transition N), which stimulates the memory to writeback what is presently in the cache to memory. After that or once the cache miss is detected, we then need to deal with the substance of the memory operation. There are four possible transitions here, and the correct one is decided as a combination of the source of the memory instruction and the type of instruction it is. For example, we could transition to `SERVICE_CACHE_MISS_TRACE_LOAD/CMTL` if the memory instruction came from the Trace Repository and it was a `LOAD`, and so on for the other combinations.

Let us deal with `STORE` operations first. If they come from a trace entry, then we transition to `SERVICE_CACHE_MISS_TRACE_STORE/CMTS` via transition R. This state then either reserves the entry in the cache once the data is available, or in the case where we are storing something that is not full-word length, performs a write-back first to preserve the data that is already present. Once this happens, we transition to `UPDATE_TRACE_REPO` via γ. If the operation comes from the CPU, however, then we perform the operation by passing the signals directly through to memory and then updating the necessary indexes as the data enters the cache, passing through state `CMMS` and transition P. Finally, then we can transition to `UPDATE_TRACE_REPO` via transition α. If the operation is a `LOAD` and has come from the CPU, then we again pass through the signals and update the cache indexes as a result. If it is a `LOAD` that comes from a trace instruction, we instrument the memory as though we were the CPU and update the cache indexes accordingly. This passes through analogous states and transitions to the STORE operation.

Once all this is complete, we transition to the UPDATE_TRACE_REPO state, whereby the trace index that has been actioned is sent off to the Trace Repository to be marked as done and then Enokida returns to the IDLE state via transition δ for the process to begin again. The operation of the Set-Associative cache, as opposed to the Direct-Mapped, is very similar in this case. However, there is some more book keeping to be done in the Set-Associative version, because the cache index cannot easily be obtained from the memory address in the same way that the Direct-Mapped cache can.

Liveness & Correctness

In a similar manner to our discussions on liveness and correctness in the previous section, we need to undertake a similar analysis for the TAC. Again, let us assume that the Trace Repository, the underlying cache implementation and the processor are correctly implemented and deadlock free. To begin, we can exclude any thoughts of deadlock from the states that operate in the pass-through mode previously described. Because of the assumptions we have made, these states enact the memory side of the protocol that is detailed in the RI5CY manual [12]. If we assume that the processor is correct and deadlock free, this cannot deadlock and will produce the correct results, because it conforms to the contracts laid out in the user manual.

Listing 4.5: This SystemVerilog code details the MAKE_REQ_TO_CACHE state's operation inside of the TAC state machine.

```
MAKE_REQ_TO_CACHE:
begin
    mem_data.ready <= 1'b0;
    proc_cache_data_rvalid_o <= 1'b0;
    proc_cache_data_rdata_o <= 32'b0;
    if ((entry_valid || proc_cache_data_req_i) && !
       prev_signals_saught)
    begin
        addr_to_check <= (proc_cache_data_req_i) ?
           proc_cache_data_addr_i : trace_out.mem_addr;
        prev_signals_saught <= 1'b1;
    end
    else if (prev_signals_saught)
    begin
        req <= 1'b0;
        // If it's the case that a memory request is waiting as well then give that priority
        if (proc_cache_data_req_i)
        begin
            // Cancel a request to the Trace Repo if there is one.
            cancel <= 1'b1;
            cpu_req.addr <= proc_cache_data_addr_i;
            cpu_req.rw <= proc_cache_data_we_i;
            cpu_req.data <= (proc_cache_data_we_i) ? proc_cache_data_wdata_i : 0;
            cpu_req.valid <= 1'b1;
            mem_trace_flag <= 1'b0;
            load_store_in <= proc_cache_data_we_i;
        end
        else
        begin
            cpu_req.addr <= trace_out.mem_addr;
            cpu_req.rw <= check_store(trace_out.instruction);
            cpu_req.data <= 32'b0;
            cpu_req.valid <= 1'b1;
            mem_trace_flag <= 1'b1;
            load_store_in <= check_store(trace_out.instruction);
        end
        state <= CACHE_HIT_GNT;
    end
end
```

Thus, if we now consider the `MRC` state in Listing 4.5,[2] when this state is entered for the first time we know that either `entry_valid` is high or will become high eventually. This is because we have assumed the correct functioning of the Trace Repository, or that `proc_cache_data_req_i` could be high now or must become high at some point in the future, as the processor is assumed to be functioning correctly. Consequently, when the first `if` statement is reached, though it may take a few cycles of waiting, `prev_signals_saught` must become high, which will allow, after the next

[2]In a similar fashion to the previous section, Listings 4.5 - 4.7 are based on the single SystemVerilog file that governs the TAC; hence, the discontinuous line numbers. They are also taken from the Direct-Mapped version of the TAC, as for these states, there is no difference in their operation.

cycle sets up the `cpu_req` variable, transition M to be taken. Consequently, there cannot be deadlock here, as there is no way that `prev_signals_saught` could remain unset and thus keep the machine locked in this state. In terms of correctness, if `proc_cache_data_req_i` becomes high at any point during this state, it will override the trace entry, which is the desired behaviour. Therefore, we can be sure that the correct request to the cache will be made, so that both properties are satisfied thus far.

Listing 4.6: This SystemVerilog code details the `CACHE_HIT_GNT` state's operation inside of the TAC state machine.

```
CACHE_HIT_GNT:
begin
    if (cancelled) cancel <= 1'b0;
    if (wb_necessary && !proc_cache_data_req_i)
    begin
        addr_to_check <= cpu_req.addr;
        state <= SLEEP;
    end
    if (!cpu_req.rw && indexed_cache_entry_valid &&
       mem_trace_flag && !wb_necessary)
    begin
        processing_flag <= 1'b1;
        if (cpu_res.checked)
        begin
            state <= UPDATE_TRACE_REPO;
        end
    end
    else
    begin
        if (cpu_res.checked)
        begin
            cpu_req.valid <= 1'b0;
            if(cpu_res.ready)
            begin
                if (!mem_trace_flag)
                begin
                    proc_cache_data_gnt_o <= 1'b1;
                    cache_hit_count <= cache_hit_count +
                       1;
                    cache_trans_count <= cache_trans_count + 1;
                    state <= CACHE_HIT_DATA;
                end
                else
                begin
                    processing_flag = 1'b1;
                    mapping_cache_to_trace_index[cpu_req.addr[INDEXMSB:INDEXLSB]] <= trace_index_o;
                    state <= UPDATE_TRACE_REPO;
                end
            end
            else if (mem_req.rw)
            begin
                cached_addr <= mem_req.addr;
                cached_data <= mem_req.data;
                state <= SERVICE_WRITE_BACK_WAIT_GNT;
            end
            else
            begin
                if (mem_trace_flag) state <= (check_store(trace_out.instruction)) ? SERVICE_CACHE_MISS_TRACE_STORE : SERVICE_CACHE_MISS_TRACE_LOAD_WAIT_GNT;
                else state <= (proc_cache_data_we_i) ? SERVICE_CACHE_MISS_MEM_STORE : SERVICE_CACHE_MISS_MEM_LOAD_WAIT_GNT;
            end
        end
    end
end
```

When we reach the `CHG` state (Listing 4.6), again we are not going to deadlock because there are three situations, one of which will occur with certainty. Either the first `if` statement will be taken, indicating that even though a preemptive action should occur, we are blocked from doing so. Therefore, the cache should take transition K; or, we can already calculate that there is no work to be done, so transition L can be taken to the `UTR` state; finally, since we have assumed the cache will respond eventually, the `else` branch of the second `if` statement can be taken, which then causes one of potentially many actions to occur. At the end of each of these actions is a different transition, dependent first on if there is a cache hit or miss and then on whether the source of the memory operation is a trace or comes from the processor. Therefore, again at this point, there cannot be deadlock, because there is no fourth alternative allowing transition J to be taken forever.

Once the `CHG` state has been left, we enter a series of states that have no loops and either implement the memory protocol from the `RI5CY` manual or perform book-keeping. With that in mind, both the correctness and liveness of the machine is therefore dependent on the contracts set in other components that we consider to be deadlock-free and correct. This is true of the `CHD`, `CMML`, `CMMS`, `CMTL`, `CMTS` and `WB` states. The only states that are left which are of concern are the `SLEEP` state, and the `UM` and `UTR` states.

Listing 4.7: This SystemVerilog code details the `SLEEP` state's operation inside the TAC state machine.

```
SLEEP:
begin
    // Continue to sleep unless it's the case that the
       blocking entry has been retired, or that a memory
       request starts
    if (proc_cache_data_req_i) state <= MAKE_REQ_TO_CACHE;
end
```

Turning to the last few states, in the case of the `SLEEP` state (Listing 4.7), we know that the processor cannot be ahead of the Trace Repository in the instructions it executes. Consequently, if the `SLEEP` state is entered it must be the case that the Trace Repository is ahead of the processor. Therefore, either the next memory instruction it executes will be the one referred to by the entry in the Trace Repository or it will occur some time later. Consequently, we know the `proc_cache_data_req_i` signal will go high at least once, allowing transition ζ to be taken. As any actions initiated by the processor will override any preemptive actions, a material change to the state of the cache will take place, meaning that when the check for the preemptive instruction is made again, it will be considering a different state and there is further chance for progress.

Considering the last two states, `UM` and `UTR`, if we assume the correct functioning of the other components, then neither of these could cause deadlock and will update the correct parts of the state to be considered correct. As a result, we can guarantee that transition δ will always be reached. Thus, the process can begin again. On a final point, if we have reached the end of processing, i.e. the processor has moved to the trap state, then the TAC will lock up as well in the `IDLE` state, but this is expected behaviour to make our experiments easier to measure.

Again, the discussion that is presented above is by no means a rigorous proof of the correctness of this machine. For complete confidence a full proof and the application of model-checking would need to be used, not only for this machine, but for all the components that make up the experimental platform. An alternative would be to start from scratch with a formal specification and refine the state machine and implementation from that. However, in both these cases the complexity involved was time-prohibitive, but this informal proof provides enough confidence to proceed in engineering the system. Any future work should seek to add the theoretical guarantees of correctness presented here.

Cache Implementations

The previous section describes the operation of Enokida, but it relies on a standard cache implementation that acts as a submodule of the overall cache. As already stated in this chapter, we use two different cache architectures, the first one being a 128 entry Direct-Mapped cache write-back cache that is taken from Hennessy and Patterson [82] directly. The Set-Associative cache is based on the same implementation as the Direct-Mapped, but is further adapted to be Set-Associative. After the adaptation it becomes an 8-way associative cache, so we have 16 sets of 8 cache blocks and each set is ordered in a FIFO fashion for simplicity. It is worth pointing out that there is no compulsion for this to be the case and it would be very easy to change the cache architecture to a full associative or even a non-standard architecture without changing Trace Assisted Caching at all. Any other number of techniques could also be used alongside Trace Assisted Caching including dead-block prediction and others, if it were so desired. However, this is not an avenue that has been explored in this thesis.

Example

Now we have fully described the hardware that implements the ideas from Chapter 3, we can see an end to end example of exactly how this technique works. For this purpose, we will look at the execution of a small part of `nsichneu.c`, the decompiled `RISC-V` assembly, which is shown in Listing 4.8. We will only focus on the first few memory operations that are executed, as these are the most illustrative. We also assume that this program has already run to completion once, so that the trace cache is filled with useful information to reduce the runtime on a subsequent run.

Listing 4.8: This is a snippet from the execution of `nsichneu.c`, address `0x200` is the start of `main` and so execution begins from there.

```
 200  a0010113    addi sp,sp,-1536
 204  5e812e23    sw   s0,1532(sp)
 208  60010413    addi s0,sp,1536
 20C  00200793    li   a5,2
 210  fef42623    sw   a5,-20(s0)
 214  4580906f    j    966c <main+0x946c>
 218  fec42783    lw   a5,-20(s0)
 21C  fff78793    addi a5,a5,-1
 ...  ...
966C  fec42783    lw   a5,-20(s0)
9670  00f05463    blez a5,9678 <main+0x9478>
9674  ba5f606f    j    218 <main+0x18>
```

After some boot up code that is omitted for brevity, execution of the example begins at address `0x200`, which the processor will execute. The Trace Repository is already primed with data as soon as the fetch begins for `0xA0010113`, so the Trace Repository will return the pair (`5E812E23`, `0xFEFC`) to the cache. This pair is the raw trace entry that is stored inside the Trace Repository from a previous run of this program, the first element of the pair stores the instruction, as can be seen on line `204` in Listing 4.8. The second element contains the effective address that instruction generated in the previous run. With this information the cache can begin it's work, as this instruction is a `STORE` it will look at the contents of the cache, discover that the appropriate entry is empty, mark it as reserved and then signal the TAC to update the state of that particular trace. This will update the trace as stored in the Trace Repository's active set, as it is still technically 'in-flight'. The movement of data in the architecture described can be seen in Figure 4.5.

While this is happening the processor is still executing `0xA0010113`, so the Trace Repository will return a new pair (`0xFEF42623`, `0xFEEC`), corresponding to `0x210` in Listing 4.8. The sequence of events will then be very similar to before, the cache will look into the space it has available, discover the entry required is empty, reserve it and then return control to the Trace Repository. At this point the Trace Repository will add this `STORE` trace to its active set, as a second in flight instruction.

At this point the processor is processing `0x5E812E23`, which means that the Trace Repository, despite returning the pair (`0xFEC42783`, `0xFEEC`) will be ignored in favour

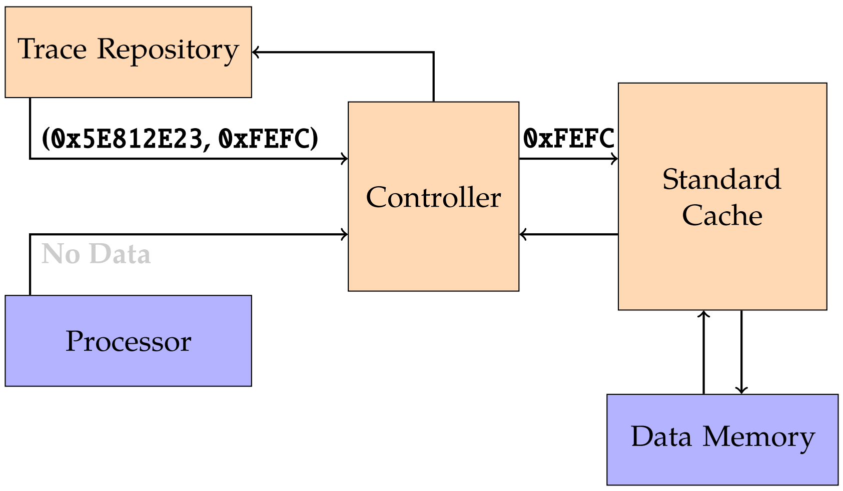


Figure 4.5: The first step of this example shows the Trace Repository signalling an intent to store data at `0xFEFC`. This is communicated to the cache, which then reserves the space and then the controller communicates with the Trace Repository to mark that trace as in-flight. On an additional point the diagram is very similar to the one in Figure 4.2, however modules have been split apart to better illustrate the passing of signals between the modules. As a result they have been recoloured to reinforce the fact they are part of the same system, even though they appear separate in the diagram.

of servicing the processor's request. This request will store the data from register `s0` at address `0xFEFC`, which in reality fills up the already reserved space in the cache from the previous trace execution of the `STORE` instruction. At this point, this is exactly what would have happened in any case, because the cache is cold. The real power of this solution comes in the next action. Once this happens the Trace Repository is again signalled to update the state of its active set. This update causes the object that represents the trace element (`0x5E812E23`, `0xFEFC`) to be marked as complete, so it can be safely overwritten, as it is no longer in flight. This can be seen in Figure 4.6.

Now the processor moves on to executing `0x00200793`, but the Trace Repository has returned the same pair as previously (`FEC42783`, `FEEC`). This pair can now be actioned, so the cache performs a `LOAD` to the correct place in the cache for address `FEFC`. This `LOAD` happens much earlier than it otherwise would have done, not taking place from the processor's point of view for another four instructions. This allows more overlapping, so that when the memory bus would have been idle, it is now performing useful work. This `LOAD` will execute, be stored in the cache and then the Trace Repository will be signalled to mark this trace element as done, because it has been completely performed, unlike the `STORE` operations where we need to wait for

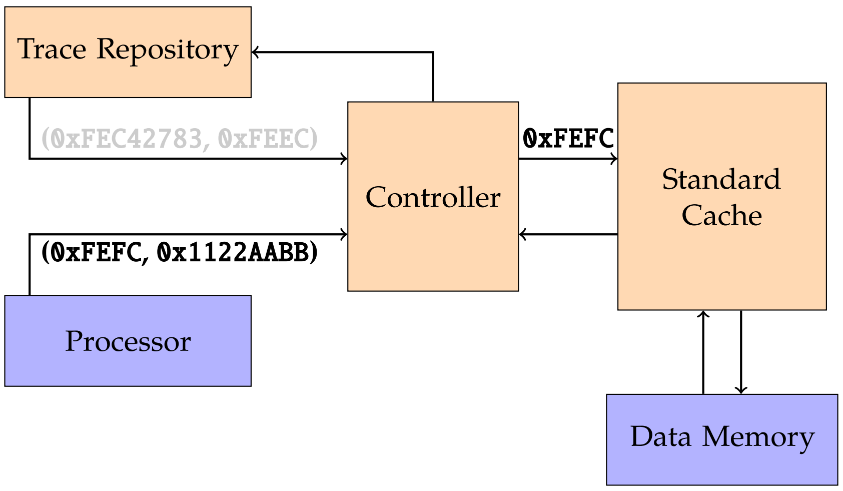


Figure 4.6: In this step the processor overrides the pair from the Trace Repository by requesting to store `0x1122AABB` at address `0x1122AABB`. This is communicated to the cache, which writes the data, as the memory has not been used before.

the data to become available. This can be seen in Figure 4.7.

The Trace Repository then returns a new pair (`0xFEC42783`, `0xFEEC`). However, again the processor wants to execute the instruction at `0x210`, so that supersedes the preemptive call. This `STORE` takes very little time, because again the data can very quickly be stored in the cache. Then, the processor will take the `JUMP` instruction to `0x966C`, as per the instruction at `0x214` and will execute the load instruction at `0x966C` to load data from `0xFEEC`. Now at this point, under normal circumstances the processor would have to stall and wait for this `LOAD` to complete, because the data cannot be in the cache and the caches are cold, so a long wait would ensure. However, because of Trace Assisted Caching, this is not the case and the data is in fact already available in the cache. This means that the instruction takes relatively little time and the processor can move on to the next branch instruction instead. This can be seen in Figure 4.8.

This pattern is repeated throughout the execution of this code, with the cache getting as far ahead of the processor as it can without violating the semantics of the program being executed and then the processor catching up as it actually executes the program. This means the process is transparent as far as the processor is concerned; from its point of view, it seems as though every memory access that is preemptively resolved is a cache hit, so all of the modifications are confined to the cache not to the processor.

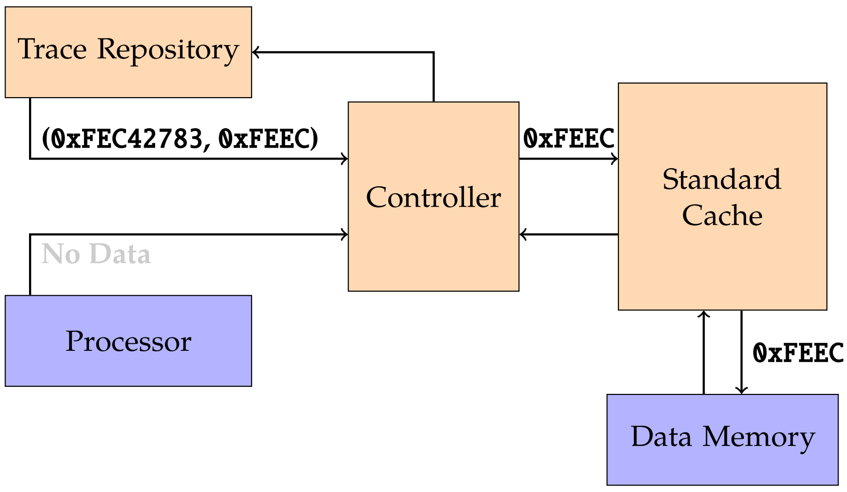


Figure 4.7: Here the processor is not requesting any memory operations, so the Trace Repository's request for a `LOAD` can be actioned. This misses in the cache and is then passed to the data memory to be recalled. Once this is completed, the Trace Repository can be updated to mark this trace as complete.

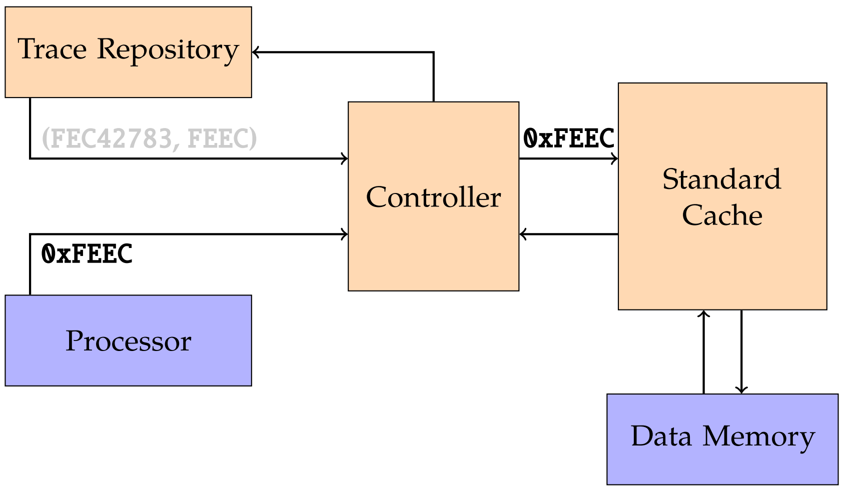


Figure 4.8: Now that the cache is primed with the information from `0xFEEC`, when the processor requests the data it becomes a cache hit, when otherwise it might have been a cache miss. This then reduces the overall runtime of the program by having overlapped this fetch with the processor executing other instructions.

4.4 Experimental Hardware

In this final section on experimental design, we will consider what our expectations might be of a platform that follows the design we have already laid out. This includes not only the Trace Recorder, but also the TAC and its associated Trace Repository. We will then conclude by thinking about the very practical concerns of implementing the platform, before moving on to conducting the experiments in the next chapter.

4.4.1 Expectations of the Platform

A fair question to ask before we consider the concrete implementation of the experimental hardware platform is how we expect it to behave with regards to some key parameters. What should the time and space requirements be for a system like this? What kind of expectations can we set around its behaviour? And, how effective do we think it will be?

Time & Space Requirements

Taking the questions of time and space first, the TAC is essentially implemented as two large modules, a cache implementation and the Trace Repository. These are composited by a higher level SystemVerilog wrapper that co-ordinates their interactions and communicates with memory. Taking these three components in turn, the cache implementation (either Direct-Mapped or Set-Associative) is relatively fixed and also does not have a complicated implementation. We would expect it to have a fairly low usage of both memory[3] and computation resources [4] as a result, and contribute only a small amount to the overall hardware footprint of the platform.

Turning now to the Trace Repository, we need a mechanism to communicate the traces that have been captured back to the processor as it is running; as a result this component consumes a large amount of hardware. In our current design the Trace Repository stores 48-bit traces (32 bits for the instruction that generated the memory access being tracked and 16 bits for the effective address). This means that if we wanted a Trace Repository to take up the same amount of memory resource as the cache, we could only have 85 trace entries stored at once. This is not enough for any reasonable speed ups. Therefore, the decision was made to include the whole

[3]BRAMs and LookUp Table (LUT) memory.
[4]LUTs, Flip-Flops (FFs) and Digital Signal Processor (DSP) resources.

trace for this implementation to demonstrate the technique, accepting that there will certainly be more efficient ways to filter this data to gain the maximum advantage for a minimal resource spend.

The Trace Repository also features several data structures that are needed in order to track the execution of traces. These include a set of in-flight memory operations, held in a circular buffer and known as the 'active-set', as well as a tracker that matches cache indexes with trace indexes, so we know how the two relate. This is further compounded in the Set-Associative case, because the state of the FIFO queues have to be tracked as well. These data structures scale as the cache does, with more cache entries prompting larger trackers. The active set may need to scale as well, but the size is determined much more by program dynamics, as it relates to the amount of instructions that can be executed before the preemptive execution is blocked.

Putting all this together, we would expect the Trace Repository to contribute a large amount of hardware to that used by the TAC, especially in the Set-Associative case, where more needs to be tracked. Looking at the SystemVerilog wrapper that co-ordinates these two smaller components, we would expect this to have negligible extra effect on memory resources. The only data structure it contains that is not a signal or a simple variable is a mapping of cache to trace indexes, which is a copy of that held by the Trace Repository. Consequently we would expect the Trace Repository to dominate these areas in terms of resource usage, as this is where a large amount of hardware will be needed to realise the design.

In terms of computation resource requirements, we would expect the Trace Repository to score very low, as it is essentially a glorified memory representation. The Trace Repository does not need to perform calculations or reformat the data it presents, so we would expect this measure to be very low when compared to a standard cache. This is particularly true in the Set-Associative case as it features a lot of extra querying hardware to find the correct placement of data elements in a set. Similarly, we would expect the wrapper to contribute very little. Even though it contains a large state machine, each of the phases does very little in terms of computation, setting some signals here and there, but mostly waiting and acting as a coordinator, rather than trying to compute results. As a result we would expect the usage in these categories to be very low across the board, with the exception of the Set-Associative cache.

Time requirements are a slightly different story. It has to be possible for the Trace Repository to respond on every clock cycle to allow trace requests to execute as soon as possible. However, there is very little the Trace Repository does other than co-ordinate querying an underlying memory implementation and presenting the results, so its time requirement in terms of clock cycles will be very low. The cache, of course, has to co-ordinate with main memory, so requires a lot more time to be spent, which will dwarf any extra cycles the repository makes by a large amount.

Moving to the question of the overarching SystemVerilog wrapper, this contains a very large state machine, which adds multiple clock cycles on top of every memory request, regardless of whether it is preemptive or not. This is because, in this implementation, a lot of co-ordination with the Trace Repository is required, even in the case that a memory access is made from the CPU, so there can be no preemptive element. We will explore this further in Chapter 6, but the overhead of adding the TAC is quite high in terms of clock cycles. Therefore, we would expect that it would not perform as well as the cache in the case of there being no preemptive actions at all. This is because an overhead is being incurred on every memory access, with no chance for the required overlapping to occur to reduce or outweigh this penalty. The other elements should introduce negligible time constraints, so we would expect the wrapper to be the largest contributor to any time overheads that we may see in the data. It is hoped that this can be accounted for and will not force worse performance than a standard cache, but this has to be accepted as a possibility.

Behaviour & Effectiveness

In terms of settings expectations for behaviour and effectiveness, it seems unlikely that the TAC will outperform a standard cache in all situations that could be presented. For a start, it would be exceptional if the TAC outperformed a standard cache in a situation where the active set of a program fitted entirely into the cache. Because the TAC relies on overlapping computation and memory instructions to achieve a runtime decrease, if there are no, or very few, memory interactions, then there will no spare capacity for the overlapping, and thus no performance increase. It would be reasonable to expect there to be parity between the two implementations, since in this situation they would be performing exactly the same operations. However, as mentioned in the previous section, there is quite a high clock cycle overhead on the TAC, this seems less likely in practice.

Secondly, we would not expect the TAC to perform particularly well for programs where there is very little gap between the memory instructions executed, i.e. there is very little ‘slack’ in the memory utilisation figures. We will define this more precisely in Chapter 6, but for now, if we consider that the TAC exists to re-order memory operations in such a way as to decrease runtime, if there are no gaps into which memory operations can be moved, it will be impossible to re-order them. In addition, due to the way that the TAC will be engineered, any CPU requests will automatically trump any preemptive requests. Therefore, it may well be the case that we could incur a large amount of overhead with regards clock cycles for no benefit, because all the TAC does is act like the cache implementation at its heart, rather than in a more intelligent way.

All that being said, it should be expected that if we are in a situation where the cache hit rate is fairly low and the memory accesses in the program are fairly spread out, then the TAC should perform much better than a standard cache, because there is more potential for the overlapping that is crucial to its success. Exactly how much overlapping of the memory instructions is required in order to perform better than a standard cache, while outweighing the overhead the TAC adds, is a question that the experimental results should help us answer. We will return to this in Chapter 6.

4.4.2 Implementing the Platform

Now we have considered what we might expect from the experimental hardware platform comes the task of actually constructing it. All the source code and automation scripts that power the experiments can be seen in Zenodo [171, 172]. This section will discuss some of the finer details of the implementation of the platform, how it was tested and some of the post-implementation results from the Xilinx Vivado suite. This will allow us to compare our expectations with the actual implementation.

The hardware was created using the Xilinx Vivado suite of tools, version 2018.2. The source code for the hardware is written in SystemVerilog and synthesised using Vivado's internal synthesiser and bit-file generator. The hardware platform is composed of several custom designed hardware blocks, including:

- A delay element (which emulates the large delays seen when consulting main memory, this will be discussed further in Chapter 5)
- The TAC (Enokida)
- The Trace Recorder (Gouram)
- The chosen cache implementation (Direct-Mapped/Set-Associative)

The designs also feature a System Integrated Logic Analyser (ILA)[155], a block that Xilinx themselves provide, so that signals can be monitored in a way that is akin to their simulation environment, but without the flexibility of software simulation. This will be used to measure runtime and extract various counter values from the system-under-test once the program has completed its execution. More about this process is discussed in Chapter 5, as well as the automation framework designed to extract this data.

Testing

As Vivado allows us to create a block design describing our hardware platform and then re-use that design in multiple projects, we took this approach when testing the platform. Several testbenches were designed, each of which could utilise the same block design and input a memory file as the experimental hardware, but in an environment that allowed much more introspection than after the design has been synthesised and a bitfile created. The simulation environments also made use of a proprietary Xilinx AXI Verification IP (VIP)[156] block that effectively mocked out the memory implementation to allow us to focus on confirming the effectiveness of the hardware that had been designed.

For purposes of clarity, it's worth mentioning at this point that a lot of this testing was not done using the real hardware itself. Vivado offers a comprehensive simulation suite that allows you to do functional and some non-functional testing inside of the tool itself. This was very useful when testing, not only because it allows a global view of all the signals that are being communicated in a design, but also because it allowed single step debugging over some of the complex synchronisation and state machine mechanics that were present. In addition Vivado bundles a lot of functionally accurate models of the components you can synthesise in the tool, however it makes very few guarantees about the timing accuracy of some of these models. It would have been very convenient to simply use the more accurate simulations to take measures of latency and other non-functional properties. This would have given us the ability to better analyse any delays or overheads introduced by the process and also would give us richer data to analyse, rather than being constrained by the data limits of the ILA that Xilinx provides. Unfortunately there were several issues with doing this, and these are explored in the opening sections of Chapter 5.

Very little, if any automated testing was done in these environments. This was not because it was not possible, but because the nature of the hardware that was being written was iterative, and it was very obvious when mistakes existed as the hardware would get stuck and programs would fail to complete. As a result, a lot of end-to-end testing was used to confirm that the caches performed as expected. As the final state of the program could be known in advance, it was easy to capture this in testbenches and then use those to quickly iterate on multiple hardware designs. In future, or if this work were to be continued, some extra time investing in a more comprehensive, module level set of testcases would be a welcome addition to the platform. SystemVerilog has had a rich assertion language for a long time versions of Vivado have been published since this work was done that support those features. Therefore, in future versions of the platform this would be worthwhile pursuing.

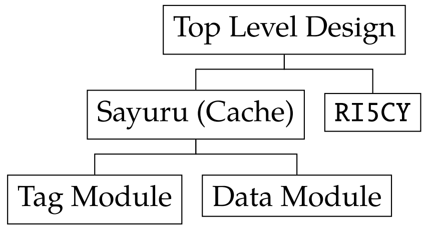
Actual Implementation Figures & Implications

Once the system was tested, the code could then be synthesised and a bitfile created, with appropriate ILA triggers applied. This allows us to inspect the utilisation figures to see how they compare to our expectations in sections above. In Tables 4.2 to 4.5, we can see the number of resources that are consumed by each of the components of interest, broken down by sub-module where appropriate. It should be noted that in Tables, 4.2 and 4.3, Gouram (the Trace Recorder) is instantiated in the Top Level design, for debugging purposes, even though it is not shown in this table. Diagrams showing how those submodules are related are included in Figures 4.9a and 4.9b.

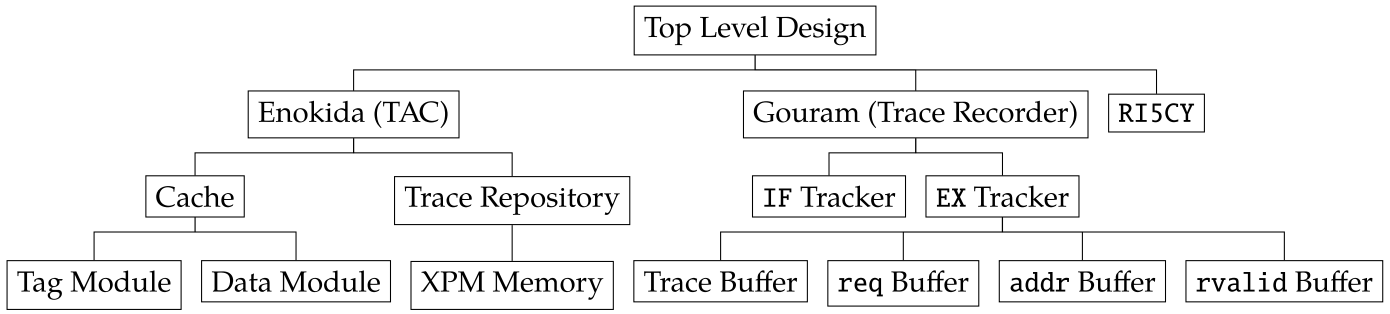
From looking at these tables we can see that the Trace Recorder Gouram is the largest consumer of resources, taking up the vast majority of the resources of the whole platform, with the worst offenders being the `EX` Tracker and the various buffers that power it. This is very much an artifact of the Proof-of-Concept nature of this implementation, but is also due to the fact that architecturally we chose an asynchronous model for tracking the memory requests. As can be seen from the tables, the `IF` tracker takes a synchronous approach and this is much more efficient in terms of resources.

Looking at the caches specifically, we see that the Direct-Mapped cache takes many fewer resources across multiple categories than a Set-Associative cache of the same size. This is not surprising, because a Direct-Mapped cache does not have any of the search hardware required in the Set-Associative cache. Specifically on this search hardware, this was implemented using a `for` loop construct in SystemVerilog. Since implementing this, it has come to light that when this synthesises, it trades space for time and synthesises parallel hardware to satisfy the specification. Unfortunately this only came to light during the data analysis and due to the length of time required to gather the data there was no opportunity to explore alternatives.

Regarding the Trace Repository in the TAC, as we predicted the Trace Repository is the largest element by an order of magnitude and contains a large amount of BRAMs to implement the XPM that backs the Trace Repository. This is a further topic we will return to in later chapters. This is something we could also further optimise and



(a) This tree diagrams shows how the modules that make up our hardware platform (Kuuga) are arranged hierarchically when instantiated with a standard cache.



(b) This tree diagrams shows how the modules that make up our hardware platform (Kuuga) are arranged hierarchically when instantiated with a TAC.

Figure 4.9: Tree Diagrams of module inter-relationships in Simple and Complex Hardware Platforms.

Name	Slice LUTs	Slice Registers	F7 Muxes	F8 Muxes	Slice	LUT as Logic	LUT as Memory	LUT Flip Flop Pairs	Block RAM Tile	DSPs
Top Level Design	141939	27736	18598	4985	39679	140364	1575	16229	279.5	6
— RI5CY	6234	2535	502	77	1875	6234	0	1364	0	6
— Sayuru (Direct Mapped)	1048	2250	224	112	836	1016	32	269	0	0
—— Direct Mapped Cache Implementation	848	1794	224	112	692	816	32	1	0	0
——— Tag Store	742	1792	224	112	653	742	0	0	0	0
——— Data Store	96	0	0	0	40	64	32	0	0	0

Table 4.2: This table shows the hardware utilisation breakdown for our hardware platform, using a Direct-Mapped cache with no trace assistance.

Name	**Slice LUTs**	**Slice Registers**	**F7 Muxes**	**F8 Muxes**	**Slice**	**LUT as Logic**	**LUT as Memory**	**LUT Flip Flop Pairs**	**Block RAM Tile**	**DSPs**
Top Level Design	145894	30694	19768	5545	42519	144283	1611	17179	279.5	6
— RI5CY	6233	2535	502	77	1788	6233	0	1369	0	6
— Sayuru (Set-Associative)	5016	5208	1394	672	3044	4948	68	1460	0	0
—— Set-Associative Cache Implementation	4816	4738	1394	672	2901	4748	68	1171	0	0
——— Tag Store	4679	4736	1362	672	2857	4675	4	1169	0	0
——— Data Store	128	0	32	0	45	64	64	0	0	0

Table 4.3: This table shows the hardware utilisation breakdown for our hardware platform, using a Set-Associative cache with no trace assistance.

Name	Slice LUTs	Slice Registers	F7 Muxes	F8 Muxes	Slice	LUT as Logic	LUT as Memory	LUT Flip Flop Pairs	Block RAM Tile	DSPs
Top Level Design	158675	35606	19766	5250	45136	156338	2337	21342	664	6
— RI5CY	6233	2535	502	77	1845	6233	0	1362	0	6
— Gouram	128452	15430	17720	4762	35800	128452	0	10665	0	0
—— IF Module	27770	7606	4936	98	7611	27770	0	7450	0	0
—— EX Module	100669	7790	12784	4664	28227	100669	0	3168	0	0
——— Trace Buffer	2176	1771	192	0	669	2176	0	1763	0	0
——— Memory `rvalid` Buffer	44045	620	6024	2204	12010	44045	0	560	0	0
——— Memory `req` Buffer	44208	624	6024	2204	12058	44208	0	532	0	0
——— Memory `addr` Buffer	1246	4136	544	256	1705	1246	0	38	0	0
— Enokida (TAC)	15965	8023	1238	360	5253	15885	80	4578	256	0
—— Trace Repository	14411	5544	790	142	4273	14411	0	4320	256	0
——— Trace Repository Memory Implementation	254	13	0	0	201	254	0	5	256	0
—— Direct Mapped Cache Implementation	1333	1794	448	218	1024	1301	32	3	0	0
——— Tag Store	1231	1792	448	218	983	1231	0	2	0	0
——— Data Store	96	0	0	0	40	64	32	0	0	0

Table 4.4: This table shows the hardware utilisation breakdown for our hardware platform, using a Direct-Mapped TAC.

Name	Slice LUTs	Slice Registers	F7 Muxes	F8 Muxes	Slice	LUT as Logic	LUT as Memory	LUT Flip Flop Pairs	Block RAM Tile	DSPs
Top Level Design	183448	43555	28249	9022	53788	181025	2423	23678	664	6
— RI5CY	6232	2535	502	77	1816	6232	0	1364	0	6
— Gouram	128434	15430	17720	4762	36198	128434	0	10651	0	0
—— IF Module	27743	7606	4936	98	7609	27743	0	7449	0	0
—— EX Module	100678	7790	12784	4664	28628	100678	0	3156	0	0
——— Trace Buffer	2176	1771	192	0	670	2176	0	1768	0	0
——— Memory `rvalid` Buffer	44046	620	6024	2204	12076	44046	0	551	0	0
——— Memory `req` Buffer	44217	624	6024	2204	12141	44217	0	522	0	0
——— Memory `addr` Buffer	1246	4136	544	256	1790	1246	0	39	0	0
— Enokida (TAC)	40709	15972	9721	4132	14087	40543	166	6857	256	0
—— Trace Repository	33527	10458	7384	3007	10677	33527	0	5419	256	0
——— Trace Repository Memory Implementation	254	13	0	0	225	254	0	7	256	0
—— Set-Associative Cache Implementation	6839	4740	2337	1125	3271	6769	70	1165	0	0
——— Tag Store	6670	4738	2305	1125	3197	6664	6	1163	0	0
——— Data Store	160	0	32	0	72	96	64	0	0	0

Table 4.5: This table shows the hardware utilisation breakdown for our hardware platform, using a Set-Associative TAC.

potentially implement further caching to both reduce hardware requirements and also reduce the overhead it introduces.

Overall regarding resources it is obvious that the TAC requires a higher level of resource spend than a standard cache of the same size. Comparing Table 4.2 and Table 4.4, specifically Enokida to Saruyu, the latter takes 15965 LUTs while the former is only 1048, and similar drops are seen throughout the resources measured. A similar story is seen in the other tables (4.3 and 4.5 where the former takes 5016 LUTs and the latter 15965. This is reasonable, not only because it is much larger than the bare cache implementations that are recorded in the tables, but also because if we consider other sources we see similar implementations fall along the same lines. If we consider the `VexRiscv` [81], a processor that implements the same `RISC-V` standard as the `RI5CY`, we can see from Table 4.6 that the caches are much more efficiently implemented. Our implementations only include a 0.5Kib cache, while this implementation includes a 4Kib Instruction Cache and a 4Kib Data Cache. Clearly many more resources are being used in our designs than necessary, even when comparing raw implementations. Furthermore, the figures for our implementation have not been engineered to reduce resource consumption, whilst the `VexRiscv` is a commercial offering. However, it demonstrates the potential for optimisation that exists, which gives us hope that even though this Proof-of-Concept implementation is very resource hungry, this will not be the case indefinitely, if more time is spent on the implementation.

Despite the high level of hardware spend the tables document, it is hoped that the high level of resource usage will lead to a commensurate high level of improvement to justify the large hardware spend, compared to having a non Trace Assisted Cache. This could be lower were it to be implemented differently and this is certainly a target for optimisation in future versions. However, this broadly falls into line with what we would expect to see for the TAC, in that it takes up more resources than a standard cache, but hopefully produces results which a standard cache would never be able to.

4.5 Summary

This chapter has shown how the experimental platform we are going to use was implemented. From high level descriptions down to discussions of the correctness of the hardware designed, we now have a high level of confidence that the hardware will perform as required in order to conduct the experiments and answer our research questions. We have also discussed the time and space usage of this new technique, which provides another axis on which to compare our implementation with that of

Name	Slice LUTs	Slice Registers	F7 Muxes	F8 Muxes	Slice	LUT as Logic	LUT as Memory	LUT Flip Flop Pairs	Block RAM Tile	DSPs
Top Level Design	1952	1489	2	0	644	1951	1	582	5.5	4
— VexRiscV (Caches Included)	1937	1456	2	0	635	1937	0	569	5.5	4
—— VexRiscV	1937	1456	2	0	635	1937	0	569	5.5	4
——— Data Cache	718	163	0	0	282	718	0	35	2.5	0
——— Instruction Cache	490	75	0	0	183	490	0	43	1.5	0

Table 4.6: This table shows the hardware utilisation breakdown in a competitor to `RI5CY`, the `VexRiscv`. This is an in order 5-stage design that includes an instruction and data cache bundled with the processor. These results were measured on the same FPGA as the `RI5CY` to allow a direct comparison.

a standard cache. We have seen the very high level of resource usage that the TAC consumes and hope that it will be justified by the performance increase the TAC allows in the programs we test.

The next step is to design an experiment such that we can verify whether the addition of this new technique is in fact worthwhile. Using these experiments we can more robustly quantify the improvement and also begin to identify and generalise a pattern for the types of program that are most amenable to this technique. The next chapter describes how this experiment is set up and then documents the results.

5 Experiments & Results

Now that we have an experimental platform we need to define a series of experiments to verify the claims made about the reduction in runtime the TAC should enable. This chapter covers those experiments and presents the results before also engaging in a discussion of those results.

5.1 Experimental Setup

By way of an experiment, we need a series of tests that can verify whether we see reductions in the runtime of programs when a TAC is used. To achieve this we have taken a benchmark suite that will serve as a stimulus, in this case the Mälardalen Worst-Case Execution Time (WCET) [76] benchmark. This benchmark was chosen because it is well established and well used. In addition it does not rely on particular hardware implementations, as all the benchmarks are provided as compilable C code. There are discussions to be had around how much the choice of benchmark will influence the results, for example Mälardalen is very focussed on code that is difficult to predict a WCET for so you often find unstructured code or code with lots of nested loops. Clearly that has an impact on the situations that arise during the execution of the code and its conceivable that there are some situations that will more favour Trace Assisted Caching. We will discuss this further in Chapter 6 once the data has been analysed as it would be premature to speculate about this before the results were known.

With a benchmark decided, we need to further decide on the hardware variations we will exercise with the benchmarks. We use five different configurations that take into account a control implementation (so simply a processor with no data caching whatsoever), and the four combinations that arise from choosing a Direct-Mapped or Set-Associative cache and including the Trace Assisted hardware or not. This leads to the following five hardware variations:

1. A `RI5CY` processor directly connected to data memory with no caching (No Cache)

2. A `RI5CY` processor with a standard Direct-Mapped cache connected to data memory (Saruyu - Direct-Mapped)
3. A `RI5CY` processor with an 8-way Set-Associative cache connected to data memory (Saruyu - Set-Associative)
4. A `RI5CY` processor with a Direct-Mapped TAC connected to data memory (Enokida - Direct-Mapped)
5. A `RI5CY` processor with an 8-way Set-Associative TAC connected to data memory (Enokida - Set-Associative)

These variations were chosen because they allow us to answer the research questions posed in Chapter 1. The first question around runtime reduction can be answered by comparing the No Cache Hardware Variant to the other 4, while the second question can be answered by pairwise comparing Saruyu and Enokida for the same underlying cache implementation. By tracking not only runtime but also the number of situations each cache experiences as the benchmarks execute we will also be in a position to answer the third question around overheads; and the final question can be answered by considering data from the two Enokida variants.

Not only this but by varying the cache implementations, it allows a view onto the impact of associativity on the performance of the TAC. By including a Set-Associative and a Direct-Mapped cache it is possible to see how increasing associativity helps or hinders the TAC. Of course this could be further expanded by including a fully-associative cache, so that the full continuum was covered. However, because fully-associative caches of a large size are rarely implemented, it was difficult to find an implementation that could be synthesised, so this was not included in this thesis.

Each of these hardware variations will be loaded with the executable code of one of the benchmark tests, loaded onto a Xilinx VC707 FPGA and then monitored using the ILA provided by Xilinx to track the execution time of the benchmark in clock cycles. This will be then be stored and can be compared against the other hardware variations.

5.1.1 Use of the Experimental Approach

It's worth exploring at this point the use of this experimental approach at all. Much research in the area of caching and architecture uses simulation as it allows a much deeper level of introspection into the underlying dynamics at play. In addition it would allow a better separation of cost and benefit, which might better allow us to answer the third research question from Chapter 1. This was originally how the experiments were planned to be run, for the reasons previously stated, however as mentioned in Section 4.4.2 there were several issues that prevented this from working in the manner originally envisioned.

The first problem was that had we used a simulator like `gem5`[25] or a similar tool we would have lost access to a lot of preexisting code that allowed us to focus on only implementing the new pieces of the architecture rather than every other part of the system. As `RI5CY` and the Pulpino SoC are synthesised using Vivado there are already a lot of components written and maintained by the project that are written with the Vivado toolchain in mind. Furthermore Vivado has access to a lot of proprietary Xilinx hardware models which again reduce the burden of work from having to implement our own interconnects or memory modules had we used something like `gem5`.

The second problem was much more of an issue in that Vivado has limited simulation options in that its most accurate simulations require you to have already completed the 'Implementation' phase of design. However even then there were many situations where designs worked in simulation but actually failed on the real hardware. This was mostly due to a lack of specificity in the SystemVerilog specification around default values and required several complex workarounds to reproduce in simulation, common cases in the actual hardware. Not only this but to get the hardware descriptions to the point of being able to use this level of simulation it was necessary to expend nearly as much time it took to generate the bitfile. Specifically, in Vivado the process of generating a bitfile goes through 4 phases, each component is synthesised in isolation, the whole system design is synthesised, the synthesised design is then implemented on the chosen hardware and finally a bitfile is created. To pass through the first three stages took around 50 minutes per design, and the generation of the bitfile around 30 seconds. So with the combination of the questions around accuracy and the fact that it confers no time advantage made experimentation more attractive, despite the fact that the level of information that could be attained was lower.

That being said, a lack of information was sometimes a benefit, particularly for the largest traces, `adpcm` being a prime example, trying to simulate the program even simply in a functional simulator, proved to be impossible on multiple occasions due to the length of the benchmark and the amount of data it produced. Even though actually implementing the design did cause a lot of information to be untracked, using the ILA made more benchmarks tractable than some alternatives because it focused the information captured.

So for these reasons, although it trades off a lot of the capabilities of introspection that would have been useful, taking the experimental approach allowed us to re-use many components that cut down the implementation time, allowed us more accurate results and also allowed us a broader range of benchmarks to compare. That being said the introspection was not lost as with some work the functional simulations could be correlated with the hardware behaviour, allowing more introspection than the hardware alone provided.

5.1.2 Process for Each Experiment

Each experiment follows a predefined set of steps:

1. Compile the benchmark into an executable using the `riscv32-unknown-elf-gcc` compiler. This requires a custom linker script as the Harvard memory layout is very different to the von Neumann architecture that `gcc` is expecting.
2. Take the compiled binary and decompile it to convert it into Xilinx's proprietary `mem` format, which is essentially a textual representation of the binary contents of the memory we want to load. As part of this process, the instruction and data segments of memory will be split into separate `mem` files. This will be passed into the hardware synthesis process in order to load the program into the BRAM, because although Xilinx has tools to do this post-synthesis, they do not support custom processor designs.
3. Generate a top-level design that references the generated `mem` files and the pre-existing Block Designs for the hardware variation that is required. These block designs also include monitoring tools in order to track the runtime effectively from within the FPGA.
4. Synthesise and implement this design as a bitfile that can be loaded onto the FPGA. There is some extra complexity here, because triggers for the logic analyser have to be loaded so that they run at boot. Thus, there is a small amount of back and forth between the FPGA and the host machine to arrange this.
5. Load the bitfile onto the FPGA and then connect to the ILA to extract the data generated while the program was running.
    a) In cases where the TAC was not included the program was run once and the results collected. In the other cases, the program was run twice, the first time as though it had no cache run to prime the trace repository and the second time with the Trace Assisted Caching enabled. The second time end-to-end is reported and the first run through of the program is ignored.

Great care was taken to ensure that the same event markers were used to measure the end-to-end time, so that neither hardware variation had any advantage or disadvantage.

b) This required small changes to Gouram and Enokida in order to recognise when the program had turned from a first run into a second. This was accomplished by injecting an artificial store into the zero register. As this is not something anyone could want to do, it was considered to be different enough that it would not be confused for a genuine instruction. In future implementations, this could become a custom `RISCV` instruction, but as that is relatively complex to achieve, it was deemed out of scope for this thesis.

6. Export this data into a script that calculates the runtime of the program and the values of various counters for cache misses, cache hits and so on. The runtime specifically was calculated by triggering the ILA on the `JUMP` instruction, before the first instruction of the `main` function. This trigger was set to be in the middle of the capture window for the ILA, so that it would also capture the first instruction of `main`. It was then possible to scan through the address values recorded and work backwards to the first clock cycle of a `data_req` signal that related to the first instruction of `main`. This calculated the start time. As the end time was dictated by the program hitting the trap state, the decision was made to designate the end time as the start of the third repetition of the `JUMP` instruction that formed the trap state. This was consistent for each hardware variant.
7. Store these values and then begin again with a new hardware configuration or benchmark as appropriate.

All these steps were automated via a automation framework, known as Ichijou [171].

5.1.3 Specific Experimental Concerns

In discussing the experiment, there are a few important issues that should be raised at this point to give a full and accurate account of exactly how the experiments work. The first point to make is that not all the Mälardalen benchmarks were run for these experiments. It was decided that only a subset of 25 out of the full set would be run, leaving out the following benchmarks:

- `compress`
- `crc`

- `edn`
- `fir`
- `lms`
- `ndes`
- `sqrt`
- `st`
- `whet`

They were largely discounted for three key reasons, with each presenting one or more of the following problems:

1. The Benchmarks required access to memory directly - In certain benchmarks, particularly `compress`, it is assumed you can know `a-priori` how much memory will be available to the program. However, because of the way Vivado generates memory, this introduced a circular dependency into the automation script. Consequently it was very difficult to tune the parameter effectively, so the decision was made to exclude benchmarks that exhibited this behaviour.
2. Intrinsic use of Floating Point Operations - The `RI5CY` processor that was used for these experiments, whilst having the option to be configured with a floating point unit, was not configured as such for these experiments due to the complicated pipeline behaviour that would introduce. This is not to say that our approach wouldn't work but would have required the repetition of work already completed to track the stages of the floating point pipeline. As a result, for reasons of simplicity, this particular avenue was not explored. Consequently, any benchmarks that relied on the use of floating point numbers were excluded.
3. Traces Too Large for trace repository - The trace repository was limited in size to 131,072 trace entries. Consequently, any benchmarks that had more than that number of memory operations could not be used. This eliminated some of the largest benchmarks due to their length and the profligate way they utilised memory instructions.

It is also worth mentioning that some of the other benchmarks had to be slightly modified in order to remove their reliance on floating point calculations. The three benchmarks that suffered from this were `minver`, `select` and `qsort-exam`, consequently they are postfixed by `-int` when they are presented in the results. There were also a few small fixes to the logic in some benchmarks, as some were hitting segmentation faults in the processor and reading undefined data.

In addition, benchmarks like `bsort`, that rely on arbitrary data, were changed to use a fixed known value before the execution of the benchmarks. For example, in `bsort` there was a fixed array of 100 numbers given to the function, rather than generating them randomly on each run. This was done because we had made a simplifying assumption that the two runs of a program would be consistently parameterised so as to focus on the implementation of the TAC, rather than focusing on other issues. Relaxing these restrictions is considered future work.

A further point to make is that extra hardware was also introduced to artificially delay calls to memory. Though this was not ideal, it was necessary for two reasons. The first was that the FPGA we used, the VC707, did not have an easy way to communicate with its on-board DRAM when the definition of memory that was expected by the `RI5CY` was out of sync with what was provided by the on-board DRAM. This meant it was very difficult to store the programs we wanted in memory and then measure the interactions with the DRAM. BRAMs, on the other hand, could be reconfigured into any non-standard configuration and is compatible with the Xilinx XPM system. Therefore we arrived at a compromise, that we would use BRAMs, which have very low latency, and delay them by introducing artificial delay modules. This effectively simulates a very slow main memory compared to a CPU, which can execute instructions much faster. To be specific the delay module introduced a delay of 50 cycles to the round-trip latency of a request to memory. As seen in Figure 2.1 this is roughly in line with what would be expected from main memory in a medium-sized embedded system.

With all this in hand, we can now progress to the results of these experiments and attempting to understand what they tell us about runtime decreases in these circumstances.

5.2 Results

The results of the experiments are presented below in several forms, the first of which is a graph that shows the runtime, in clock cycles, of each of the benchmarks executed on the different hardware variants. The benchmarks are ordered by increasing number of memory instructions recorded, a very rough approximation of complexity.

Turning back to our research questions from Chapter 1 we can immediately pick out some interesting details from this graph. The first is that the blue lines in each bar group are never exceeded by any of the other hardware variants. That being the case it's fairly easy to conclude that the adding the TAC does reduce runtime,

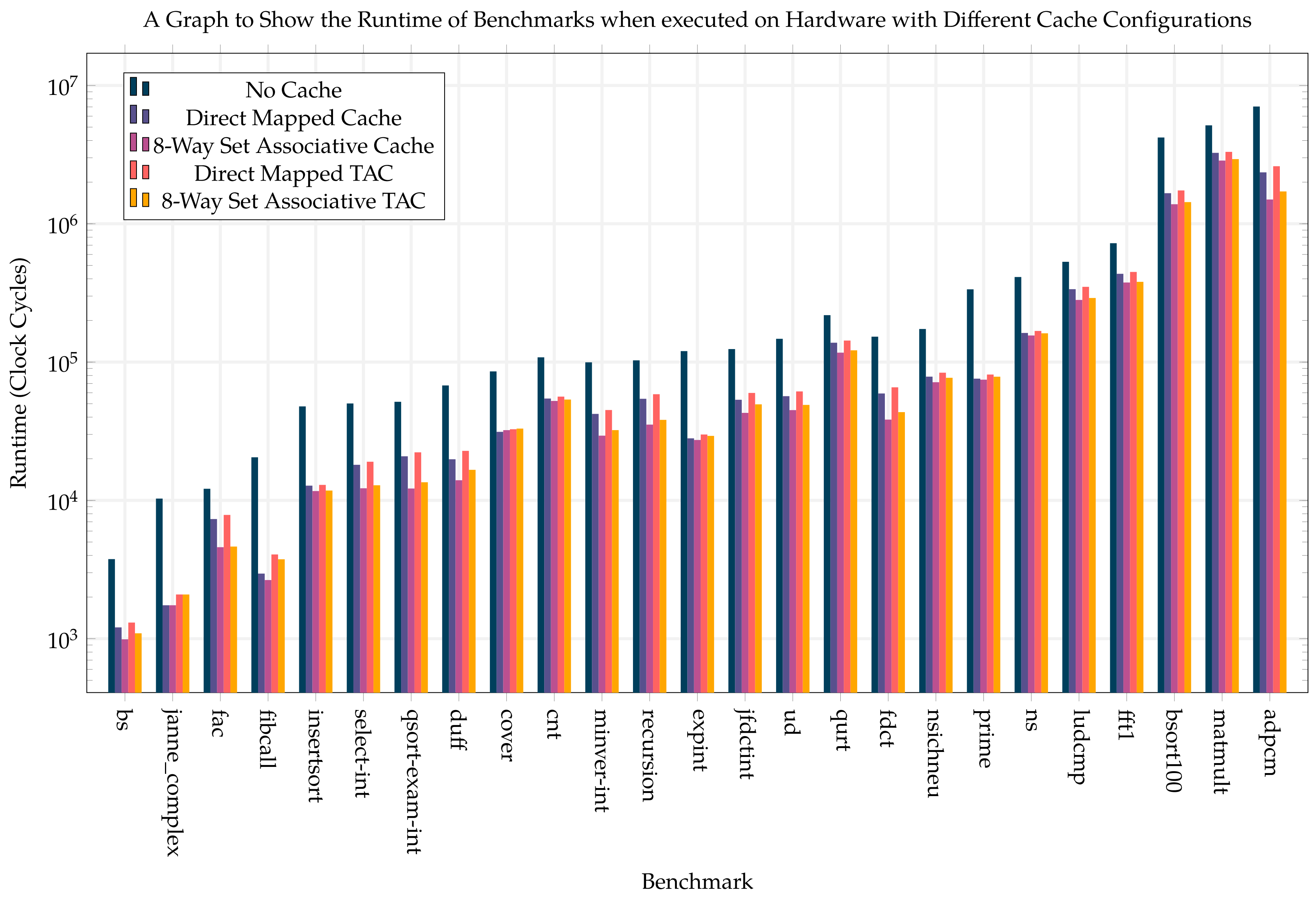


Figure 5.1: This graph shows the runtime of each of the benchmarks as it executed on each hardware variant. The raw data used to plot this graph can be seen in Appendix B.

however as it also clear from the graph it does not reduce runtime as much as the standard caches in any of the benchmarks that were tested. There are several particular occasions where the performance is very close, such as `cnt`, `prime` and `expint` but other occasions where performance is barely comparable between the solutions, such as `fac` and `recursion`. In Chapter 6 we'll analyse these particularly poorly performing benchmarks as they have several interesting features that give insight into why there is such a performance disparity. To make clear the difference in performance Figure 5.2, will now plot the percentage improvement of each hardware variant over the No Cache variant. This will allow us to normalise the performance differences so we can see how much improvement or degradation is experienced in each case.

Figure 5.2 shows in even more clarity exactly where the performance differences are in the various benchmarks. In general the results are somewhat disappointing as, even though we expended much more hardware, as we saw at the end of Chapter 4, the performance of the TAC never exceeded the performance of the standard caches. From these results it would therefore appear as though we have to answer our second research question in the negative, as there are no examples of the TAC outperforming the standard caches for any of the benchmarks we tested. The outliers mentioned in the previous paragraph also become more pronounced when the data is viewed in this format, particularly in the cases of `fac`, `recursion` and `minver-int`.

A further point to be made is that when we consider associativity the Set Associative cache, without trace assistance, is consistently the best across every benchmark. Specifically, it improves runtime over not having a cache by around 67% on average and by 18% over a Direct-Mapped cache of the same size, as can be seen in Table 5.1. This is despite the usual trade off between higher associativity and higher access times, as mentioned in Chapter 2. Though no concrete analysis has been done as to exactly the reasoning for this, since this thesis is concerned with the effects of adding the TAC not the changes in associativity, it is conceivable that due to the relatively small size of the cache the searching overhead is not comparable to the time it takes to writeback an element to main memory. Since this phenomena happens much more frequently in Direct-Mapped caches due to their design this could explain the consistently better performance. However of course this is speculative and would need measurements of each of the phenomena in question to quantify exactly.

To look more specifically at the exact performance differences, Table 5.1 shows the percentage improvement or degradation when comparing the different caching hardware variants to each other and from this we can see that the TAC always exhibits a degradation, in one case of over 40%, when compared to a standard

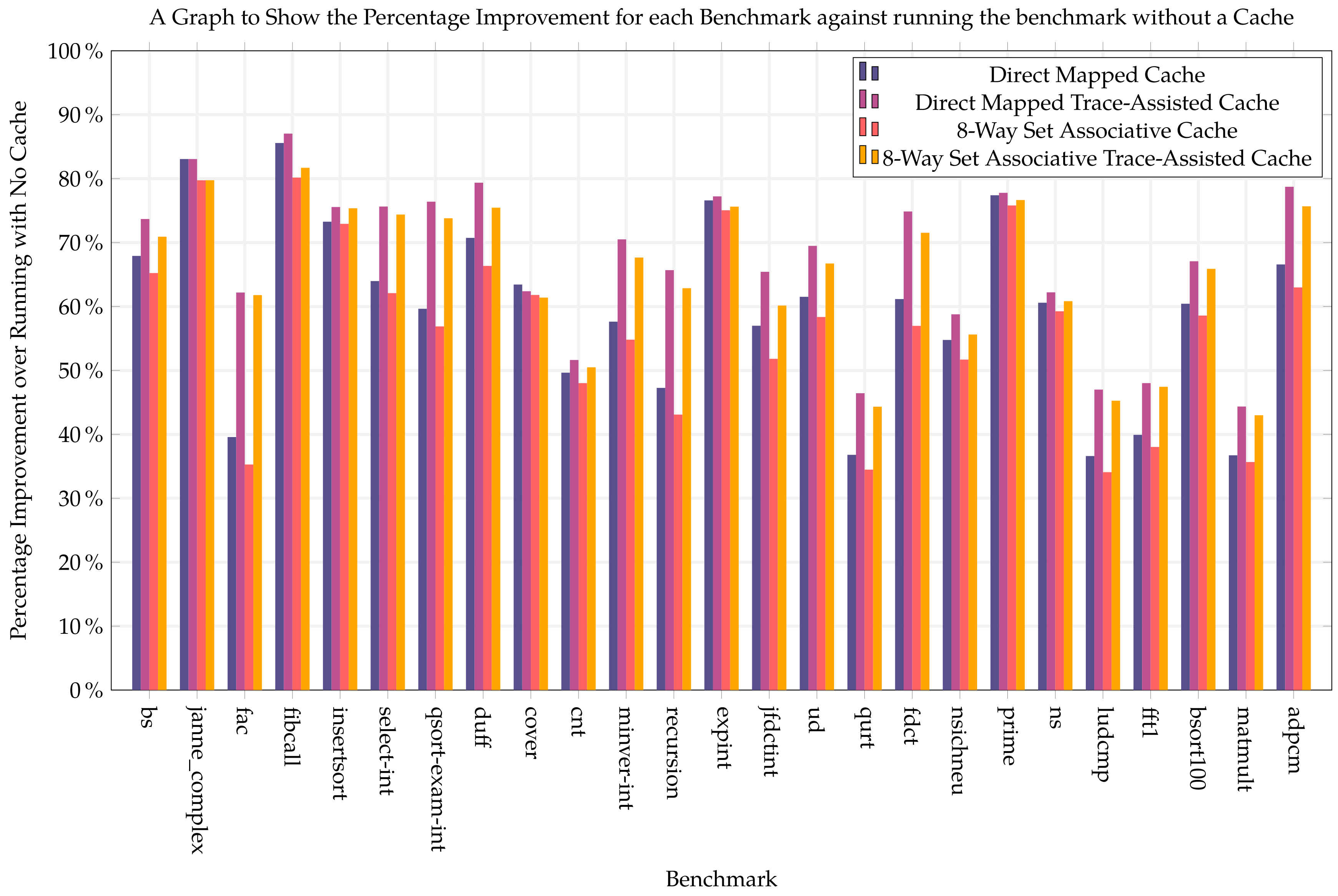


Figure 5.2: This graph shows the percentage improvement for each hardware variant over our control hardware, which has no caching at all and is directly connected to memory. This allows us to show how much adding the TAC is of benefit and sometimes of detriment to different benchmarks on different pieces of hardware.The raw data used to plot this graph can be seen in Appendix B.

cache.

This adds further weight to our negative answer to our second research question, and in the next chapter we will explore exactly the reasons for some of these very large degredations in performance, particularly focusing on the largest outliers, `duff`, `fibcall`, `fac` and `janne_complex` among others. At the other end of the spectrum, some of the degradations are around 1%, such as `insertsort`, `fac` and `fft1`. What makes these benchmarks so amenable to this technique, while some seem almost allergic to it? This will also be explored in the next chapter. The key figure to bear in mind from this table specifically is that the average degradation in performance is 8%, which considering the large hardware spend identified in Chapter 4 is very disappointing.

5.3 Exploring Cache Metrics

To begin that process of making sense of these results we can also consider some of the ancillary metrics that were captured during the experiments, particularly those around cache behaviour. Therefore, in the next set of tables we show increasingly more detailed data about each cache implementation.

5.3.1 Standard Caches

For the simple caches we are more limited in the information we can capture, because the number of situations that can arise is not as great as in a TAC. Consequently, every memory access is either a cache hit or a cache miss and may involve a write-back to memory if the desired location is full. This can be split further into hits and misses that are in response to `LOAD` or `STORE` instructions. Tables 5.2 and 5.3 display this information for a standard Direct-Mapped cache and a standard 8-way Set-Associative cache respectively.

If we consider these tables at face value we start to see some indications as to the performance differences. We will expand upon these in the next chapter, but if we consider that the central idea of Trace Assisted Caching is that cache misses can be converted into cache hits through overlapping if we look at benchmarks like `expint`, `insertsort` and `prime` the number of misses present is very low as a proportion of the total number of memory interactions. For example in Table 5.2 `expint` records only 20 out of over 2000 memory interactions that were cache misses, while for `insertsort` it's 33 out of around 800, which gives limited space for any improvement from the TAC. In the Set-Associative case it's even more pronounced, with only 15

Benchmark	Improvement (Set-Associative vs. No Cache)	Improvement (Set-Associative vs. Direct-Mapped)	Degradation (Set-Associative TAC vs. Standard Set-Associative)	Degradation (Direct-Mapped TAC vs. Standard Direct-Mapped)
bs	73.68%	17.98%	10.50%	8.37%
janne_complex	83.07%	0.00%	19.60%	19.66%
fac	62.19%	37.42%	1.07%	7.11%
fibcall	87.04%	10.18%	41.28%	37.52%
insertsort	75.56%	8.58%	0.80%	1.26%
select-int	75.64%	32.37%	5.14%	5.25%
qsort-exam-int	76.40%	41.51%	10.98%	6.84%
duff	79.36%	29.49%	18.93%	14.98%
cover	62.39%	−2.86%	2.65%	4.45%
cnt	51.63%	3.94%	2.36%	3.25%
minver-int	70.50%	30.38%	9.65%	6.65%
recursion	65.69%	34.93%	8.22%	7.92%
expint	77.22%	2.68%	7.06%	6.60%
jfdctint	65.42%	19.61%	15.26%	12.03%
ud	69.49%	20.71%	9.04%	8.21%
qurt	46.43%	15.25%	3.94%	3.66%
fdct	74.86%	35.27%	13.24%	10.78%
nsichneu	58.78%	8.86%	7.66%	6.79%
prime	77.78%	1.70%	5.07%	7.06%
ns	62.22%	4.13%	3.67%	3.40%
ludcmp	47.00%	16.40%	3.29%	3.99%
fft1	48.02%	13.47%	1.10%	3.16%
bsort100	67.08%	16.80%	3.61%	4.68%
adpcm	78.73%	36.35%	14.35%	10.76%
matmult	44.35%	12.05%	2.45%	1.67%

Table 5.1: This table details the performance increase/decrease when comparing the runtime of each of the benchmarks in the situations given in the headings. For example, the Improvement (Set-Associative vs. No Cache) gives the percentage improvement of the runtime when a Set-Associative cache is added compared to not having a cache at all.

Benchmark	Cache Hits (Load)	Cache Hits (Store)	Cache Misses (Load)	Cache Misses (Store)	Writebacks (Proceeding Load)	Writebacks (Proceeding Store)
bs	42	8	7	8	2	0
janne_complex	128	43	0	8	0	0
fac	91	35	37	45	23	37
fibcall	238	115	2	10	2	2
duff	794	196	56	150	51	80
qsort-exam-int	564	147	107	76	57	36
select-int	598	155	62	56	44	33
insertsort	594	189	11	22	7	5
expint	1504	497	5	15	5	5
cover	889	366	206	9	29	1
recursion	812	431	217	280	190	235
jfdctint	1253	337	272	342	159	245
minver-int	1092	282	190	151	124	88
fdct	1556	555	340	254	199	153
ud	1657	425	286	115	115	69
cnt	861	399	158	126	45	101
nsichneu	2202	3	269	490	211	271
prime	3899	1727	9	23	8	15
qurt	1214	645	257	454	203	292
ns	4646	932	823	8	112	0
ludcmp	3208	1338	981	1165	626	617
fft1	3897	2707	687	926	574	488
bsort100	48170	13233	8294	1657	4944	692
adpcm	76672	32876	6592	6220	6209	6037
matmult	43550	6984	19392	4722	4075	793

Table 5.2: This table lists the number of each type of event that occurs during the run of each benchmark with regards to cache behaviour when run using a Direct-Mapped Cache. Hits and misses are broken up into `LOAD` and `STORE` operations, as are cache misses. Writebacks are split up similarly, but the split is made based on the instruction they preceded.

Benchmark	Cache Hits (Load)	Cache Hits (Store)	Cache Misses (Load)	Cache Misses (Store)	Writebacks (Proceeding Load)	Writebacks (Proceeding Store)
bs	44	8	5	8	0	0
janne_complex	128	43	0	8	0	0
fac	113	48	15	32	14	15
fibcall	240	115	0	10	0	0
duff	801	236	49	110	24	3
qsort-exam-int	661	187	10	36	10	5
select-int	651	177	9	34	8	4
insertsort	605	196	0	15	0	0
expint	1509	497	0	15	0	0
cover	879	367	216	8	38	0
recursion	936	565	93	146	81	111
jfdctint	1371	456	154	223	134	169
minver-int	1232	346	50	87	39	46
fdct	1767	720	129	89	71	52
ud	1827	463	116	77	71	35
cnt	894	398	125	127	46	86
nsichneu	2331	3	140	490	122	346
prime	3906	1733	2	17	2	2
qurt	1339	916	132	183	83	122
ns	4741	932	728	8	67	0
ludcmp	3610	2085	579	418	304	241
fft1	4327	3364	257	269	159	168
bsort100	52035	14524	4429	366	3332	277
adpcm	82571	38699	693	397	418	282
matmult	49985	9172	12957	2534	1745	871

Table 5.3: A this table features the same categories of presentation as Table 5.2, but focuses on the Set-Associative cache.

misses for both `expint` and `insertsort` for the same number of memory accesses overall.

If we compare that with the performance measures we've seen, particularly in Figure 5.2, we would expect to see parity in the ideal case but this is consistently not the case. In fact `expint` and `prime` experience non-trivial performance degradations compared to their equivalents running with standard caches. `expint` has a degradation of 7.06% when comparing Set-Associative Caches and 6.6% in the Direct-Mapped case, whereas `prime` has a 5.07% degradation in the Set-Associative case and 7.06% in the Direct-Mapped case. This suggests that there are overheads being introduced by the TAC which are being incurred even in the cases where there is little potential for it to act. This goes someway to explaining the average performance penalty that is incurred in the transition to using a TAC and this will be explored more fully in Chapter 6.

5.3.2 Trace-Assisted Caches

If we now move on to the TAC we can track many more events, because we introduce more possibilities through the addition of the preemptive behaviour. Cache hits are split into three variants for both `LOAD` hits and `STORE` hits. The first of these is the simplest, which is a straight cache hit. Here, the CPU requests some data and it is already in the cache with no preemptive action having been taken to place it there directly. Of course it may be the case that the set of previous instructions that caused this piece of data to occupy this place in the cache had preemptive elements, but we only track at the granularity of individual memory instructions, not chains of instructions that cause an outcome.

The second type of cache hit has a preemptive element, but that preemptive element was a cache hit. This means that when the trace was recalled from the trace repository it was found that the data was already in the cache, so no action was required. The third type is a cache hit with a preemptive miss, this is the same as the previous type, but a cache miss occurred, so there is the potential for a real memory operation to have occurred also. The measurements relating to events of this type can be seen in Table 5.4 for the Direct-Mapped cache and Table 5.6 for the Set-Associative Cache.

Cache misses are still fairly simple, because if data is not present in the cache there cannot have been any preemptive action, otherwise the cache query would have hit. As a result, they are only split into `LOAD` and `STORE`. Writebacks are also tracked separately, but are further subdivided by whether they occurred as part of a preemptive action or in response to a CPU request. The measurements for these events are shown in Table 5.5 for the Direct-Mapped Cache and Table 5.7 for the Complex Cache.

Considering this new data there are two key points that will inform some of our discussions in the later chapters. The first is that we had been assuming that the a high number of preemptive actions would be a positive indicator of higher performance relative to the standard caches but this is not the case. Taking `bsort100` in the Direct-Mapped case, 11478 preemptive actions out of a total number of 71354 memory interactions. However, according to Table 5.1 there is still a degradation in performance of 4.68%. This implies there is not a direct correlation between the number of preemptive actions taken and the corresponding performance increase and adds further weight to the idea that overheads are being introduced somewhere in the process. A further interesting example of this phenomena is `matmult` with 31409 preemptive actions but still a 1.67% degradation. While this is better it still underscores that the standard caches are more economic as they produce better performance and utilise fewer resources as shown in Chapter 4.

To dig into this slightly further it's not only the number of preemptive actions that occur its also the type of action that has a bearing, for example in the case of `fibcall` all the preemptive actions are `STORE` operations. When we combine that with the fact that there are few if any writebacks recorded this shows that even though these `STORE` operations are being made preemptively they're not reducing runtime very much because they are mostly storing data in the cache rather than in main memory. This gives the impression that for workloads that can store their working set effectively in the cache, the TAC is not going to give any advantage because there are relatively few memory operations that actually access main memory and so very few operations to overlap.

On a related note the fact that the type of preemptive instruction is very important is also demonstrated by `adpcm`. If we consider in Table 5.3 that it records 418 Writeback proceeding a load but then in the comparative table (Table 5.7) for the TAC that is only reduced to 381 while the performance is degraded by 14.35% but if we consider `fft1` it manages to reduce the same figures from 147 to 85 and shows only a 1.1% degradation. This further demonstrates that not all preemptive operations contribute equally and this is a theme we'll explore further in Chapter 6 with reference to specific benchmarks and their properties.

Anomalies

At this point it is worth addressing some of the anomalies that this very granular data has. If we consider Table 5.6, we can see that the columns Cache Hit (Loads +

Benchmark	Cache Hits (Loads + No Preemption)	Cache Hits (Loads + Preemptive Hit)	Cache Hits (Loads + Preemptive Miss)	Cache Hits (Store + No Preemption)	Cache Hits (Store + Preemptive Hit)	Cache Hits (Store + Preemptive Miss)
bs	37	5	2	8	0	2
janne_complex	114	14	0	43	0	1
fac	85	6	0	33	2	1
fibcall	238	0	0	115	0	2
duff	794	0	0	193	3	3
qsort-exam-int	529	35	0	144	3	14
select-int	564	34	0	137	18	14
insertsort	583	11	0	189	0	12
expint	1504	0	0	497	0	3
cover	711	178	11	364	2	2
recursion	799	13	0	374	57	4
jfdctint	1249	4	0	305	32	15
minver-int	989	103	0	261	21	10
fdct	1536	20	0	548	7	3
ud	1542	115	3	415	10	8
cnt	446	415	0	391	8	10
nsichneu	951	1251	3	2	1	4
prime	3899	0	0	1296	431	3
qurt	1140	74	11	628	17	1
ns	4598	48	8	932	0	2
ludcmp	2906	302	13	1309	29	3
fft1	3596	301	19	2651	56	2
bsort100	37515	10655	0	12424	809	14
adpcm	73667	3005	0	27147	5729	4
matmult	12195	31355	0	6934	50	4

Table 5.4: This table captures the behaviour of the Direct-Mapped TAC with regards to the cache hits it experiences over the course of running each benchmark.

Benchmark	**Cache Misses (Load)**	**Cache Misses (Store)**	**Writeback (Load)**	**Writeback (Store)**	**Writeback (Preemptive Load)**	**Writeback (Preemptive Store)**
bs	5	6	2	0	0	0
janne_complex	0	7	0	0	0	0
fac	37	44	23	37	0	0
fibcall	2	8	2	2	0	0
duff	56	147	51	80	0	0
qsort-exam-int	107	62	57	36	0	0
select-int	62	42	44	33	0	0
insertsort	11	10	7	5	0	0
expint	5	12	5	5	0	0
cover	195	7	29	1	0	0
recursion	217	276	190	235	0	0
jfdctint	272	327	159	245	0	0
minver-int	190	141	124	88	0	0
fdct	340	251	199	153	0	0
ud	283	107	115	69	0	0
cnt	158	116	45	101	0	0
nsichneu	266	486	211	271	0	0
prime	9	20	8	15	0	0
qurt	246	453	203	292	0	0
ns	815	6	112	0	0	0
ludcmp	968	1162	626	617	0	0
fft1	668	924	574	488	0	0
bsort100	8294	1643	4944	692	0	0
adpcm	6592	6216	6209	6037	0	0
matmult	19392	4718	4075	793	0	0

Table 5.5: This table captures the behaviour of the Direct-Mapped TAC with regards to the cache misses and writebacks it experiences over the course of running each benchmark.

Benchmark	Cache Hits (Loads + No Preemption)	Cache Hits (Loads + Preemptive Hit)	Cache Hits (Loads + Preemptive Miss)	Cache Hits (Store + No Preemption)	Cache Hits (Store + Preemptive Hit)	Cache Hits (Store + Preemptive Miss)
bs	44	0	4	8	0	2
janne_complex	128	0	0	43	0	1
fac	115	0	2	52	0	10
fibcall	240	0	0	115	0	2
duff	802	0	1	236	0	4
qsort-exam-int	660	0	8	187	0	23
select-int	652	0	6	177	0	22
insertsort	605	0	0	196	0	12
expint	1509	0	0	497	0	5
cover	879	0	180	366	0	4
recursion	946	0	65	573	0	54
jfdctint	1367	0	1	441	0	42
minver-int	1224	0	31	338	0	35
fdct	1771	0	2	714	0	15
ud	1821	0	93	461	0	41
cnt	889	0	110	402	0	107
nsichneu	2324	0	71	7	0	2
prime	3907	0	1	1734	0	5
qurt	1335	0	29	906	1	26
ns	4739	0	643	932	0	2
ludcmp	3608	0	186	2106	0	61
fft1	4356	0	104	3371	0	10
bsort100	52036	0	4304	14527	0	40
adpcm	82576	0	62	38707	0	91
matmult	49575	0	12631	8811	0	2059

Table 5.6: This table captures the behaviour of the Set-Associative TAC with regards to the cache hits it experiences over the course of running each benchmark.

Benchmark	**Cache Misses (Load)**	**Cache Misses (Store)**	**Writeback (Load)**	**Writeback (Store)**	**Writeback (Preemptive Load)**	**Writeback (Preemptive Store)**
bs	1	6	0	0	0	0
janne_complex	0	7	0	0	0	0
fac	11	18	11	4	2	6
fibcall	0	8	0	0	0	0
duff	47	106	21	4	1	1
qsort-exam-int	3	13	3	4	8	1
select-int	2	12	2	4	5	0
insertsort	0	3	0	0	0	0
expint	0	10	0	0	0	0
cover	36	5	5	0	34	0
recursion	18	84	18	61	57	44
jfdctint	157	193	141	164	1	14
minver-int	27	60	21	39	24	12
fdct	123	80	72	48	2	7
ud	29	38	15	17	56	21
cnt	20	16	8	3	38	82
nsichneu	76	484	68	329	66	0
prime	0	11	0	1	1	0
qurt	104	162	73	116	16	22
ns	87	6	5	0	62	0
ludcmp	395	336	211	182	92	39
fft1	124	252	85	153	62	5
bsort100	124	320	100	263	3239	12
adpcm	626	298	381	194	37	72
matmult	736	836	189	752	1878	138

Table 5.7: This table captures the behaviour of the Set-Associative TAC with regards to the cache misses and writebacks it experiences over the course of running each benchmark.

Preemptive Hit) and the analogous column for `STORE` operations are empty on every benchmark except 1. In addition, there are no reported preemptive writebacks in Table 5.5. With regards to the first point, there are several explanations as to why this is, ranging from a complex set of cache behaviours combined with the behaviour of the preemptive unit, to a bug in the counters that are used within the hardware to measure these metrics. However, we will not focus on that explicitly in the upcoming sections, for several reasons. The first is that these are very much ancillary metrics that help to give context to the main data in Figure 5.1 and Table B.1. Therefore, their accuracy is helpful, but not crucial to the overall picture that the data provides to us. Moreover, the issue that may be present in the hardware is most likely one of miscategorisation, rather than an error in measurement, because we know that there should be no more interactions with memory as we add the TAC; they should simply be re-ordered, and if you calculate the number of memory accesses these figures entail the numbers are still consistent. As a result, this data is still useful to talk broadly about the number of preemptive actions taken, which is how it will be used in Chapter 6.

The second point, on the number of writebacks, has very similar features to the first point. However, it also has the added caveat that if we look at the results reporting 0 writebacks, it is quite consistent with the number of preemptive misses recorded. Therefore, as the previous paragraph points out, this data is still useful and will be expounded on further in Chapter 6.

5.3.3 Summary

Putting all this together the results give a fairly negative impression when compared to our research questions. They show that while it's true the TAC reduces runtime, when compared to a standard cache it does so less and costs more hardware resources to do so. In some benchmarks this can be as bad as a performance reduction of 40%, however on average the degradation is less at around 8%.

In terms of reasoning as to why this is, the results seem to suggest that there are overheads being introduced by the technique that are retarding its performance as compared to a standard cache. Further it appears that individual program dynamics play a big role in exactly how effective the technique is. The next chapter will take a more in depth approach and consider the outlying data, digging deeper into individual benchmarks to help us both answer our remaining research questions positively or negatively but also to further our understanding of why and how the technique could be improved with more work in future.

Part IV

Analysis & Conclusion

6 Analysis

Having undertaken a surface-level analysis of the data we collected in the previous chapter we will now proceed more closely analyse the measurements taken. We know from the previous chapter that we saw a degradation in performance from the use of the TAC, so we will first consider some of the generalised causes of this degradation and use those causes to categorise the set of benchmarks. After that, we will consider six benchmarks, `fac`, `insertsort`, `fibcall`, `janne_complex`, `fft1` and `duff`, each showing a very large or very small degradation, to give us some insight into the best and worst performing benchmarks. We will then conclude by generalising this approach to codify the characteristics of a benchmark that will benefit from Trace Assisted Caching.

6.1 Causes of Lack of Improvement

The overall story of the data we have seen is that the addition of a TAC actively inhibited the performance of the CPU over using a standard cache. This story was repeated in both the Direct-Mapped and Set-Associative cases. In this section we will focus on the major causes of this degradation and link the benchmarks to those causes to build a broad picture of what is most inhibiting performance.

6.1.1 Lack of Capacity to Improve

The first reason why we may not see an improvement in performance is that some benchmarks may have already achieved their maximum effectiveness simply by adding a cache. For example, let us consider a program that has an active set smaller than the capacity of the cache. If we then further suppose the active set is sympathetically laid out in memory, so as not to incur any conflicts over cache locations, then once all the data has been loaded into the cache, there will be no further interactions with memory. Programs like this will run incredibly fast, because main memory has been cut out of the loop entirely. However, this situation is antithetical to Trace Assisted Caching.

The rationale behind Trace Assisted Caching is that overlapping main memory accesses, with the execution of normal instructions, will reduce runtime and improve system performance. However, cases like this are antithetical to that idea, because there are no interactions with memory to overlap. Using Trace Assistance will then become a hindrance to the performance of a program, because there are more overheads associated with Trace Assisted Caching than with using a standard cache (see Section 6.1.4).

The marker for problems of this sort are benchmarks where the recorded number of cache misses is very low as a proportion of the total number of memory requests. This is true of `janne_complex`, `fibcall`, `select-int`, `insertsort`, `expint` and `prime`, and the trend is more pronounced in the Direct-Mapped case, as shown in Figure 6.1. This figure also shows that while there is some relationship it's not as simple as this being the only contributory factor.

Furthermore, if we consider a Memory Activity Diagram[1] of memory activity across the four hardware variants that contain caches, we can see this behaviour clearly. As Figures 6.2 and 6.3 show, we see lots of activity initially and then a large period where no memory accesses occur before the program terminates.

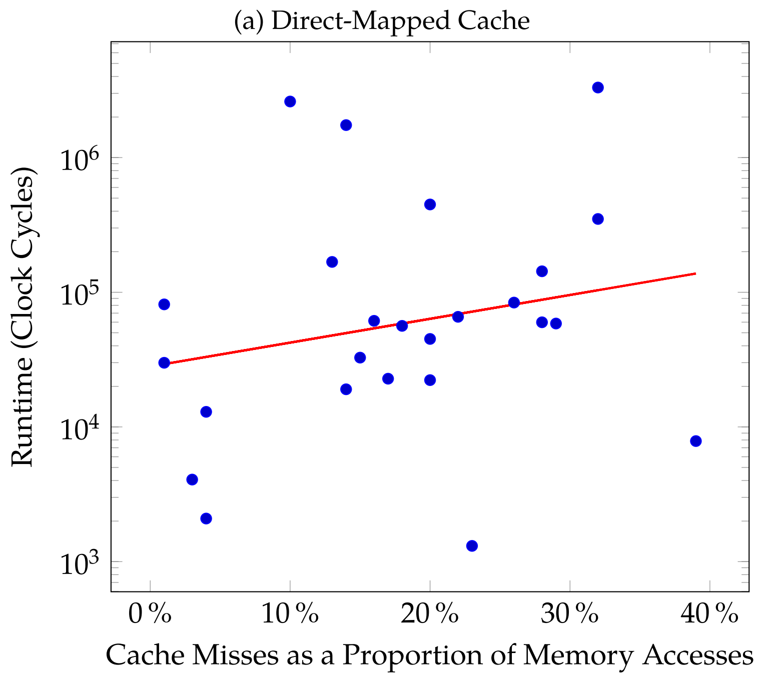
All this demonstrates that in this situation there is little that can be done by the TAC to decrease the runtime of the program, because there are no (or very few) memory operations it can overlap. Consequently, in these situations the TAC will only add overhead, leading to the longer runtimes we have recorded.

6.1.2 Gap Between Memory Instructions

The second reason why we may not see improvement is that for Trace Assisted Caching to be effective there has to be a reasonable gap between pairs of memory instructions that map to the same cache block. This is somewhat explained in Section 4.3.1, but is worth explaining again in this context.

Suppose we fetch a trace from the trace repository that was a `LOAD` from a particular memory address. Then further suppose the cache block it is to be loaded into is empty, so this piece of data can be loaded from memory preemptively and execution continues. Now, if the processor is someway behind the preemptive execution and then suppose the trace repository fetches the next trace to action and it is a `LOAD` from a different address that maps to the same cache block. If we allow this to go ahead, when the first `LOAD` is executed by the processor it will receive the data intended for the second `LOAD`, which is clearly incorrect. This obviously cannot be allowed to

[1]A 1D heatmap that shows a colour if memory is active i.e. serving a request from the processor, and no colour if not.



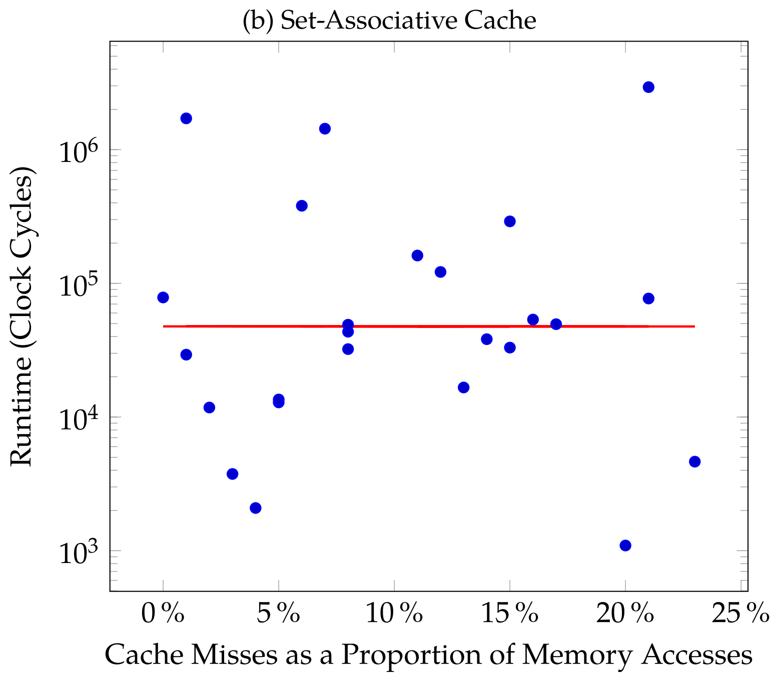


Figure 6.1: The two graphs above compare the runtime of each benchmark against the proportion of cache misses the processor experienced the Direct-Mapped and Set-Associative TAC runs of each benchmark. The trend lines are plotted using linear regression and show that while some data points track the trend line there are many outliers, which clearly shows that more than this one variable is responsible for the results experienced.

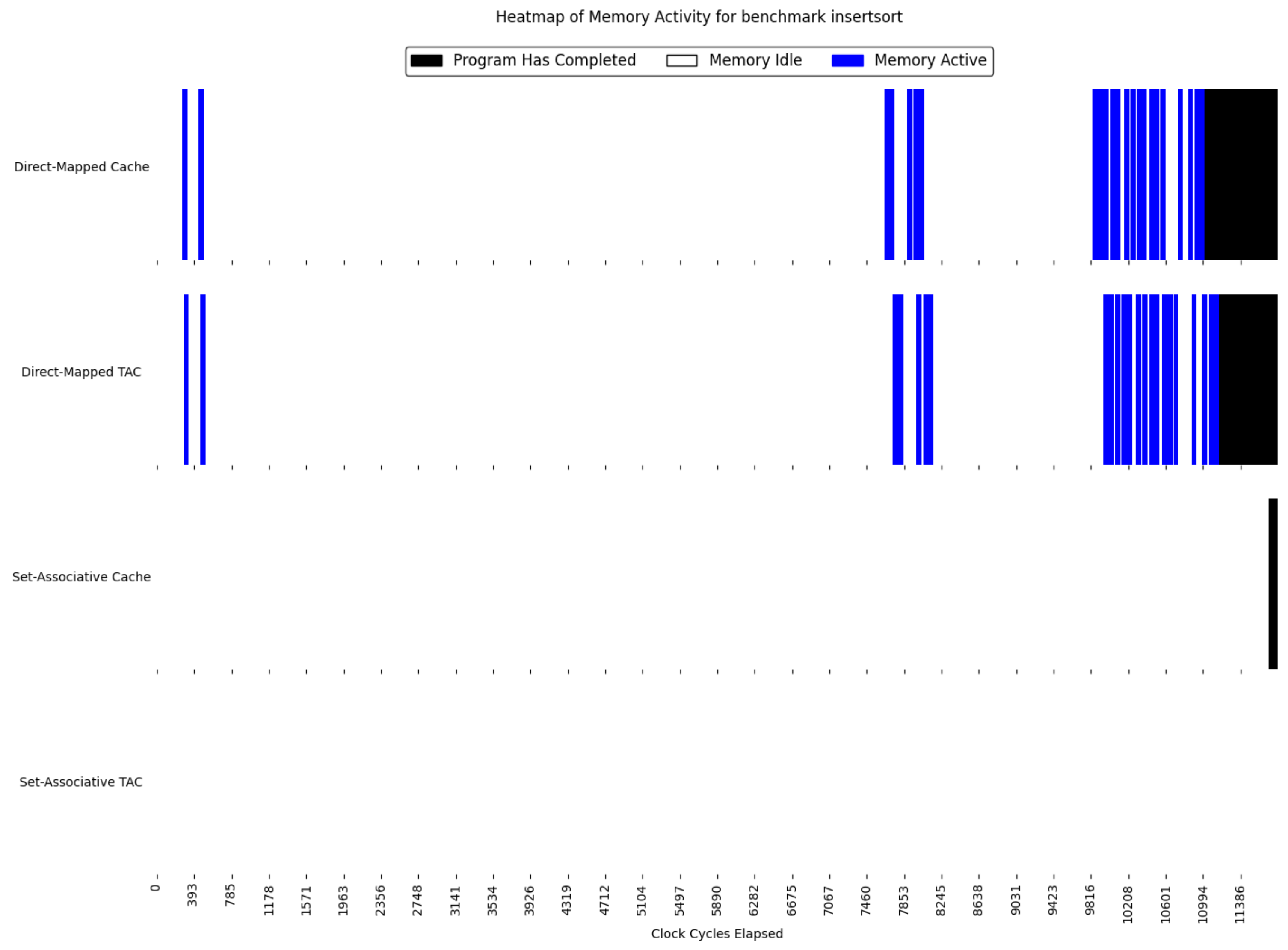


Figure 6.2: This diagram shows the slices of time where memory was active during the execution of each of the hardware variants for the `insertsort` benchmark. For the purposes of these diagrams we consider memory to be active between the start of the processor's request to memory and the `rvalid` signal going high. We can clearly see that there is an initial period of activity and then a large period of time where memory is not utilised at all. This is consistent with the behaviour described.

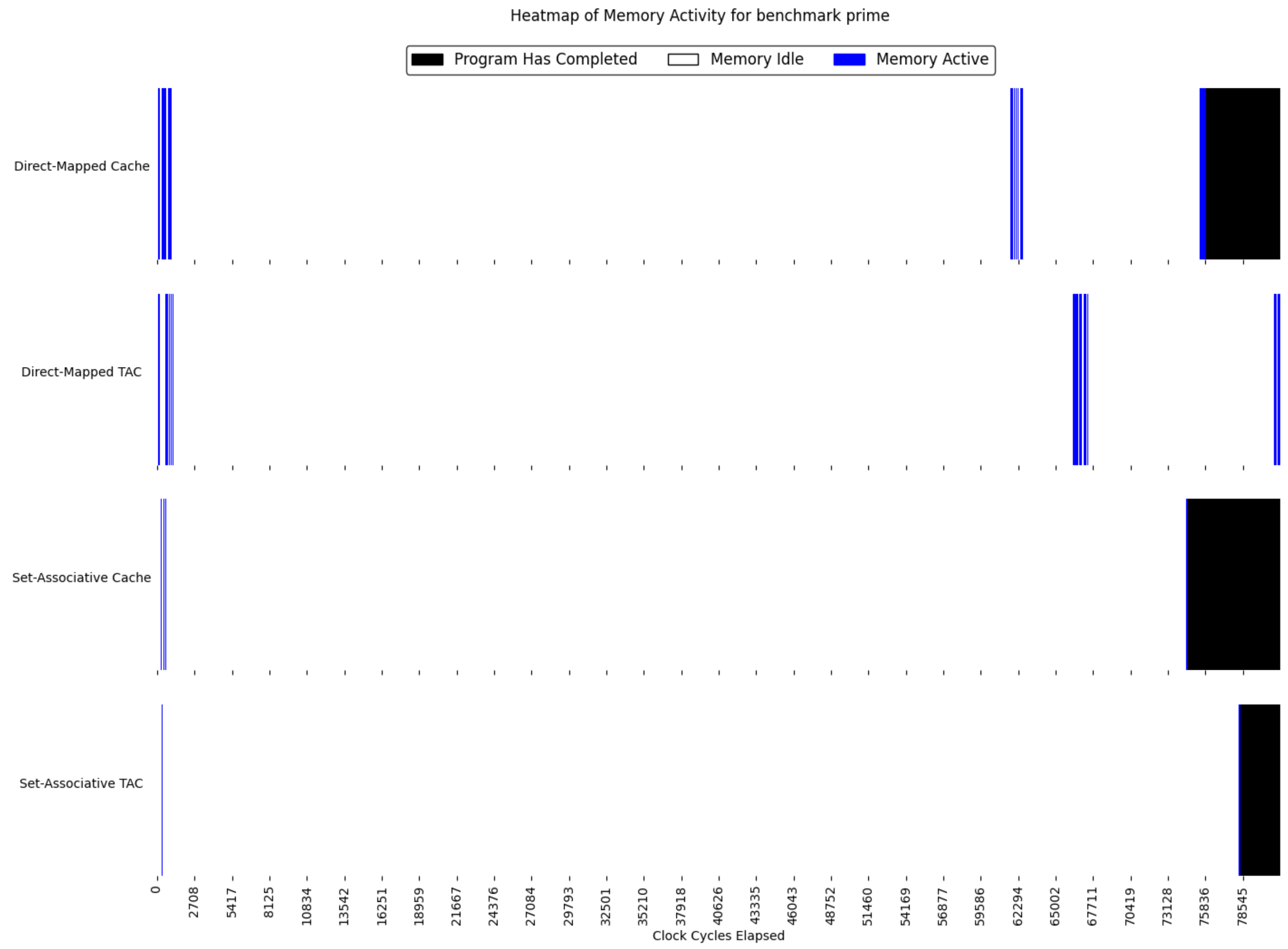


Figure 6.3: Similarly to Figure 6.2, this Memory Activity Diagram shows the slices of time where memory was active during the execution of each of the hardware variants for the `prime` benchmark. Again we see similar behaviour.

happen if we want to maintain consistency, so the only option is to effectively make the preemptive part of the cache do nothing while we wait for the processor to catch up.

The consequence of this is that the preemptive elements of this scheme can only be so effective if there is a large enough gap between memory instructions that address the same cache block to not cause blockages and stalls like this. There is an obvious 'sweet-spot' whereby the processor is far enough behind the TAC to allow the TAC to conduct preemptive actions, but close enough so that any stalling instructions will not effect the execution of the TAC for very long. To further complicate matters, this threshold or gap is dynamic because it depends on the density of instructions, the cache hardware and mapping scheme that are used and a variety of other factors. As a result, if programs are very dense in the number of memory instructions and the breadth of addresses they reference, there may be little opportunity for the TAC to make an impact.

This kind of behaviour is exemplified in benchmarks where we see little reduction in the number of misses between having and not having a TAC, or in situations where the benchmark is very memory bound, i.e. all the data cannot be held in the cache at once. This is particularly true in all the Direct-Mapped TAC measurements, as seen by comparing Tables 5.2 and 5.5, but also in `duff`, `jfdctint` and `fdct`, as shown by comparing Table 5.3 with 5.7. The reason for this is because the Direct-Mapped cache is so limited in its choice of data placement it is going to need to writeback to memory much more than the Set-Associative cache would, so the situation described above is more common.

Furthermore, this makes the whole process very sensitive to sudden large bursts of latency. For example, if preemptively, the cache encounters a `LOAD` that also requires a writeback very early on in the execution of the program, then there is a delay of at least 100 cycles while both of these operations execute. (For more details see Section 6.1.3). Once that is complete the likelihood is that another memory operation is already waiting, because the number of computation instructions required to cover this gap would be significant if they execute in 4 cycles each. To make matters worse, this process is self-perpetuating, once the preemptive unit is behind, unless there is a long run of computation instructions, it is unlikely that it will catch up again. There are ways this could be alleviated, with techniques that are more akin to OoO processors, and we will consider some of those in Section 6.3.

	`LOAD`	`STORE`
Hit	0	0
Miss	1	0
Miss with Writeback	2	1

Table 6.1: This matrix shows how many memory operations it would take to respond to a request to `LOAD` or `STORE` data based on what happens in the cache i.e. hit or miss.

6.1.3 Types of Misses

A further point to consider when talking about the performance of Trace Assisted Caching is that it is not just the amount of misses that affect the potential for an increase in performance. It is important to remember that even though we can, and often do, count all cache misses as fundamentally equivalent, this is rarely the case. This is even more pronounced when considering the preemptive case because of the unique behaviour of the TAC. If we are preemptively attempting to load a value from memory and we have to writeback a value already present in the cache, that whole action incurs a double penalty, as this necessitates two independent memory operations. At the other end of the spectrum, if we have a memory operation that does not require a writeback and is a `STORE`, then the penalty is negligible because preemptive stores cannot make any data changes to the cache.

Consequently, if we think about cache operations, they can be sorted along two dimensions. The first is whether the operation is a `LOAD` or `STORE` and the second is the type of hit or miss. This forms a matrix which is shown in Figure 6.1. If we consider Table 6.1 carefully then we can see that the type and combination of misses that a program exhibits without a TAC will have a major impact on the amount of runtime that could be saved by applying Trace Assistance. Of course this is multi-dimensional and depends on other factors such as the gaps between these hits and misses, but if we consider a benchmark like `bsort100` that incurs 4429 `LOAD` writebacks in the Set-Associative case and 8294 in the Direct-Mapped case (Tables 5.2 and 5.3), then you can imagine that it is going to be more difficult to overlap those effectively, because they are double the length of a simple `LOAD` miss, so many more computation instructions will be required.

6.1.4 Overheads Incurred

The final, and perhaps most significant problem the TAC has when trying to increase performance is the overhead it introduces. A constant problem when designing new cache optimisations is that often the overhead in performing the optimisation

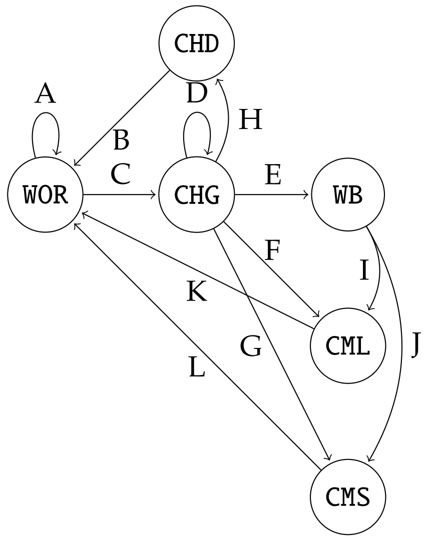


Figure 6.4: The state machine that details the operation of our standard cache implementation.

can exceed the gains that made from actually applying it. In this case, the essential problem is the book keeping and querying of the trace repository that has to occur for every memory request the CPU makes. Even if this is kept to a minimal number of cycles, this will accumulate over the course of a long running program, and may eliminate any benefit. To further re-enforce this point, if we consider the points made in Section 6.1.3, then we can also see that the types of cache interactions will have a distinct impact on the overheads involved. For example, if multiple memory operations are required, say in the case of a `LOAD` that misses and has to writeback the data before starting, then a much higher overhead will be incurred than for a straight miss.

By way of example, let us consider the process of a cache hit and miss, in the worst case, between a Direct-Mapped TAC and a standard Direct-Mapped cache. To make this easier, we will use the State Machine for the standard cache, shown in Figure 6.4. By way of explanation, the data captured for the next section was taken from simulations of the hardware inside of Vivado's simulator, this allowed us to expose the internal operations of the cache components, while abstracting away from the details of the memory implementation.

Overheads in Standard Cache Implementation

First, if we consider a cache hit on our Direct-Mapped cache implementation (Saruyu) we are already in the `WAIT_ON_REQ/WOR` state at the start, so let us assume that the `data_req` signal has just gone high to signal that the CPU requires some data. The state machine will recognise that signal has gone high, set some control signals and move to the `CACHE_HIT_GNT/CHG` state, taking one clock cycle. The control signals set query the cache to see if the data that is required are already present. This will take up one extra cycle due to the cache implementation. Hence, in the next clock cycle we can process the result from the cache and discover this request is a hit, so we transition to `CACHE_HIT_DATA/CHD`. Then, in the next clock cycle this will return the data to the CPU and return us to the `WAIT_ON_REQ` state, taking overall four clock cycles. This is the same no matter whether the memory request is a `LOAD` or a `STORE`.

If it had been a cache miss instead, then the beginning of the transaction would occur as before, except that after `CACHE_HIT_GNT`, assuming we did not need to writeback, we would transition to `SERVICE_CACHE_MISS_LOAD/CML` or `SERVICE_CACHE_MISS_STORE /CMS` as appropriate. In the latter case, we expend one clock cycle before returning to `WAIT_ON_REQ`, so this flavour of memory transaction also takes four clock cycles. However, in the former case, with a `LOAD` miss, we have to perform a read from memory, so this transaction will take $3 + A_c$ where A_c is the end-to-end time taken between requesting the memory read and having the data available to us. If a writeback has to occur in either case, then we have to add on a further A_c to account for having to write back the data to memory.

To summarise, if we consider how long in clock cycles it takes to perform each type of memory transaction, we get Table 6.2.

	`LOAD`	`STORE`
Cache Hit	4	4
Cache Miss	$3 + A_c$	4
Cache Miss (WB)	$3 + 2A_c$	$3 + A_c$

Table 6.2: This table summarises how long each type of memory transaction will take for the cache to execute. Here A_c refers to the length of time taken from first requesting a memory access to the point at which the `rvalid` signal is raised and the data is accessible.

To demonstrate these figures are reliable we can try to reconstruct the measurements from these estimates as follows. First, based on simulations of the running program, we can assume that A_c in this case is 52, as there is a 50 cycle delay on the memory operation to emulate slow main memory, and then 2 further cycles to perform the access for real. Now let us consider `fac` in Table 5.2 running on a standard Direct-Mapped cache.

To begin we have 126 cache hits that occur throughout the execution of the program (91 + 35), which each take 4 cycles from a memory perspective, so that adds up to 504 cycles. Following that, we incur 14 `LOAD` cache misses and 8 `STORE` misses, neither of which require writebacks, so $14 * (52 + 3) = 14 * 55 = 770$ and $8 * 4 = 32$ cycles, which cumulatively brings us to 1306 cycles. Finally, we have 23 `LOAD` operations that required a writeback totalling $23 * (3 + 2 * 52) = 2461$ and 37 `STORE` operations that also require a writeback totalling $37 * (3 + 52) = 2035$. Putting this all together gives us 5802 cycles. That leaves 1529 cycles to perform the rest of the computation, giving us a Clock Cycles Per Instruction (CPI) of around 3.3, very much in line with our expectations.

Overheads in the TAC

Now, turning our attention to the TAC, there are many more situations that can arise due to the preemptive operations. All these situations are documented as the column headings of Tables 5.4 and 5.5, but will be repeated here for ease of reference. These are:

- Cache Hit (No Preemptive Action) - *H*
- Cache Hit (Following a Preemptive Hit) - *HPH*
- Cache Hit (Following a Preemptive Miss) - *HPM*
- Cache Hit (Following a Preemptive Miss & Writeback) - *HPMW*
- Cache Miss (No Preemptive Action) - *M*
- Cache Miss (With Writeback) - *MW*

Each of the items in the list has a separate `LOAD` and `STORE` variant. The full list of times taken can be seen in Table 6.3. For exact details on how all these figures are derived see Appendix C.

Returning to the two tables, as we can see there is a very large difference between the amount of time taken to service similar operations between the two broad types of implementation. This means that even before the TAC has begun, it is at a disadvantage compared to the standard cache and would need to overlap a lot of memory instructions to out-perform the standard implementations. This goes a long way to explaining why we see the results that we do, with respect to no TAC entry outperforming a non-TAC, over any benchmark. Furthermore, this affects every benchmark that was run, because these overheads are incurred on every memory access no matter the outcome, so any improvements here will see improvements across the entire suite of benchmarks.

	LOAD	STORE
H	10	10
HPH	19	20
HPM	$A_c + 20$	21
HPMW	$2A_c + 20$	$A_c + 21$
M	$A_c + 10$	11
MW	$2A_c + 10$	$A_c + 11$

Table 6.3: This table shows how long it would take each different type of memory transaction to progress through the system. Of course, especially in the case of the preemptive elements, the times quoted may happen in two distinct blocks, once when the preemptive element happens and the other after that, when the data is actually requested. A further point is that some of the time allocated into the transactions for book-keeping actually occurs overlapped with the next instruction. Therefore, while these figures are indicative, they cannot be used to directly reconstruct the times taken in the data without accounting for overlapping. For more details see Appendix C.

6.2 Benchmark by Benchmark Analysis

Having now considered some of the more generalised arguments as to why this technique may have yielded disappointing results, there are several benchmarks that display abnormal behaviour with respect to the average improvement or degradation.

6.2.1 `janne_complex`

Starting with one of the smallest benchmarks, `janne_complex` is a synthetic benchmark that consists of two loops, where the maximum number of iterations in the inner loop is dependent on the current number of iterations of the outer loop. This is often used to test WCET calculations. The source code for the benchmark can be seen in Listing 6.1.

Listing 6.1: Source code that implements the `janne_complex` benchmark. The commentary has been removed for ease of presentation.

```
int complex(int a, int b)
{
  while(a < 30)
    {
      while(b < a)
        {
          if(b > 5)
```

```
            b = b * 3;
          else
            b = b + 2;
          if(b >= 10 && b <= 12)
            a = a + 10;
          else
            a = a + 1;
        }
      a = a + 2;
      b = b - 10;
    }
  return 1;
}

int main()
{
  int a = 1, b = 1, answer = 0;
  answer = complex(a, b);
  return answer;
}
```

If we consider the measurements taken, this shows identical runtime performance between a Direct-Mapped and Set-Associative cache, but it then has a nearly 20% performance degradation when compared to the equivalent TAC, as shown in Table 5.1. This is over twice the average degradation of 8%, despite the TAC managing to overlap one STORE operation in both the Set-Associative and Direct-Mapped cases. This is a classic example of the problem described in Section 6.1.4, where the overheads of the TAC are swamping any benefits it might have brought with it. Specifically, overlapping one STORE instruction would reduce the runtime by at most 50 cycles, but the overheads incurred would increase every LOAD hit by 9 cycles and there are 128 recorded cache hits with no preemptive action in the Set-Associative case. Therefore, it is unsurprising that this is the case when Trace Assisted Caching is applied.

Another interesting property of this benchmark is that after a few initial STORE misses, that are somewhat inevitable unless the cache is prepopulated, all the action takes place inside the cache itself. This is obvious from the figures as there are no writebacks and only seven cache misses that resulted from STORE operations. Consequently, memory is never accessed, so there is very little capacity to overlap the remaining operations. This adds further credence to the ideas presented in 6.1.1, that a low number of misses relative to the overall number of memory transactions leaves little space for Trace Assisted Caching to achieve anything, though as the graphs in that section showed that is not the whole story. Ideally therefore, we would expect parity of performance between the two, but as the overheads are introduced, this is sadly not the case, leading to the results we see.

6.2.2 fac

Turning now to `fac`, we see a quite different story to `janne_complex`. `fac` is again a simple benchmark that recursively calculates the sum of the first 5 factorials i.e. 0! + 1! + 2! + 3! + 4! + 5!. The code for this benchmark can be seen in Listing 6.2.

Listing 6.2: Source code that implements the `fac` benchmark to calculate the sum of factorials.

```c
int fac (int n)
{
  if (n == 0)
     return 1;
  else
     return (n * fac (n-1));
}

int main (void)
{
  int i;
  int s = 0;
  volatile int n;

  n = 5;
  for (i = 0;  i <= n; i++)
      s += fac (i);

  return (s);
}
```

If we focus on the Set-Associative case we see that the TAC vs non-TAC performance only differs by around 50 clock cycles or 1.09%, as per Table 5.1. This is an example where clearly the overheads introduced are to some extent balanced by the savings that are made through the overlapping and this is borne out when we look at the results.

As we can see in Tables 5.6 and 5.7 the TAC manages to overlap 10 of the `STORE` misses with writebacks and 2 of the `LOAD` misses with writebacks. These 12 preemptive operations also allowed another 6 operations to be converted from hits to misses, causing a substantial decrease in runtime that was effectively balanced by the overheads incurred. The reason it can do this, whilst other benchmarks do not display this behaviour, is obvious if we consider the disassembled binary of the benchmark, as in Listing 6.3.

Listing 6.3: This listing is the decompiled assembly code for `fac`, created using a `RISC-V` port of `objdump`.

```
200  fe010113    addi        sp,sp,-32
204  00112e23    sw          ra,28(sp)
208  00812c23    sw          s0,24(sp)
20C  02010413    addi        s0,sp,32
210  fea42623    sw          a0,-20(s0)
214  fec42783    lw          a5,-20(s0)
218  00079663    bnez        a5,224 <fac+0x24>
21C  00100793    li          a5,1
220  0200006f    j           240 <fac+0x40>
224  fec42783    lw          a5,-20(s0)
228  fff78793    addi        a5,a5,-1
22C  00078513    mv          a0,a5
230  fd1ff0ef    jal         ra,200 <fac>
234  00050713    mv          a4,a0
238  fec42783    lw          a5,-20(s0)
23C  02f707b3    mul         a5,a4,a5
240  00078513    mv          a0,a5
244  01c12083    lw          ra,28(sp)
248  01812403    lw          s0,24(sp)
24C  02010113    addi        sp,sp,32
250  00008067    ret
254  fe010113    addi        sp,sp,-32
258  00112e23    sw          ra,28(sp)
25C  00812c23    sw          s0,24(sp)
260  02010413    addi        s0,sp,32
264  fe042423    sw          zero,-24(s0)
268  00500793    li          a5,5
26C  fef42223    sw          a5,-28(s0)
270  fe042623    sw          zero,-20(s0)
274  0280006f    j           29c <main+0x48>
278  fec42503    lw          a0,-20(s0)
27C  f85ff0ef    jal         ra,200 <fac>
280  00050713    mv          a4,a0
284  fe842783    lw          a5,-24(s0)
```

```
288  00e787b3    add         a5,a5,a4
28C  fef42423    sw          a5,-24(s0)
290  fec42783    lw          a5,-20(s0)
294  00178793    addi        a5,a5,1
298  fef42623    sw          a5,-20(s0)
29C  fe442783    lw          a5,-28(s0)
2A0  fec42703    lw          a4,-20(s0)
2A4  fce7dae3    ble         a4,a5,278 <main+0x24>
2A8  fe842783    lw          a5,-24(s0)
2AC  00078513    mv          a0,a5
2B0  01c12083    lw          ra,28(sp)
2B4  01812403    lw          s0,24(sp)
2B8  02010113    addi        sp,sp,32
2BC  00008067    ret
```

If we look at the structure of this benchmark, at the beginning of each of the major subroutines (`0x200` and `0x254`) there are multiple store operations. As these operations can occur relatively quickly from the point of view of the TAC, the `LOAD` operations, which take much longer, can be started much sooner than would be the case if we were executing under a standard cache. In addition, the density of memory operations is lower in the most frequently traversed section of the code, with the highly dense sections occurring at the end of a subroutine or the end of the program. This demonstrates the suppositions made earlier in the chapter that it is not simply the overall amount of memory operations available that determines the effectiveness, but also the distribution of those operations throughout the program that has an impact.

The corollary of looking at `fac` is that it shows that while there is work to be done in addressing the level of overheads, it can be the case that the penalty and benefit balance out. In which case, if we could reduce the overheads involved with the technique, while retaining the ability to overlap memory operations in this way, we could conceivably reach a point where the TAC performed even better than the standard caches. This is not only because we have reduced the overheads, but also because that could potentially unlock more overlapping if more memory requests can be serviced by the TAC. Though of course, this brings into question data hazards inherent in the program, which are much harder to remedy after the program has been compiled.

6.2.3 fibcall

`fibcall` is a relatively simple benchmark that calculates the sum of the first n Fibonacci numbers in an iterative fashion. The source code for the benchmark is reproduced in Listing 6.4.

Listing 6.4: This listing implements the `fibcall` benchmark that calculates the sum of n Fibonacci numbers.

```
int fib(int n)
{
  int  i, Fnew, Fold, temp,ans;

    Fnew = 1;  Fold = 0;
    for ( i = 2; i <= n; i++ )
    {
      temp = Fnew;
      Fnew = Fnew + Fold;
      Fold = temp;
    }
    ans = Fnew;
  return ans;
}

int main()
{
  int a;

  a = 30;
  fib(a);
  return a;
}
```

In this case, the addition of the TAC leads to an increase of not around 8%, as the average would suggest, but of around 41% (1109 cycles) in the case of the Direct-Mapped Cache and 38% (1096 cycles) in the case of the Set-Associative Cache, as shown in Table 5.1. If we look at the Memory Activity Diagram of its execution, in Figure 6.5, we see that the behaviour of `fibcall` is quite similar to that of `janne_complex`, in that after the initial misses, there are long periods of time where main memory is not accessed at all. Consequently, before we even consider other factors, the potential for improvement is quite low. But why is the degradation so great in both of these cases?

If we look more closely we see that all the misses that are incurred in both cases are always `STORE` operations and there are no writebacks. Thus, there is a low

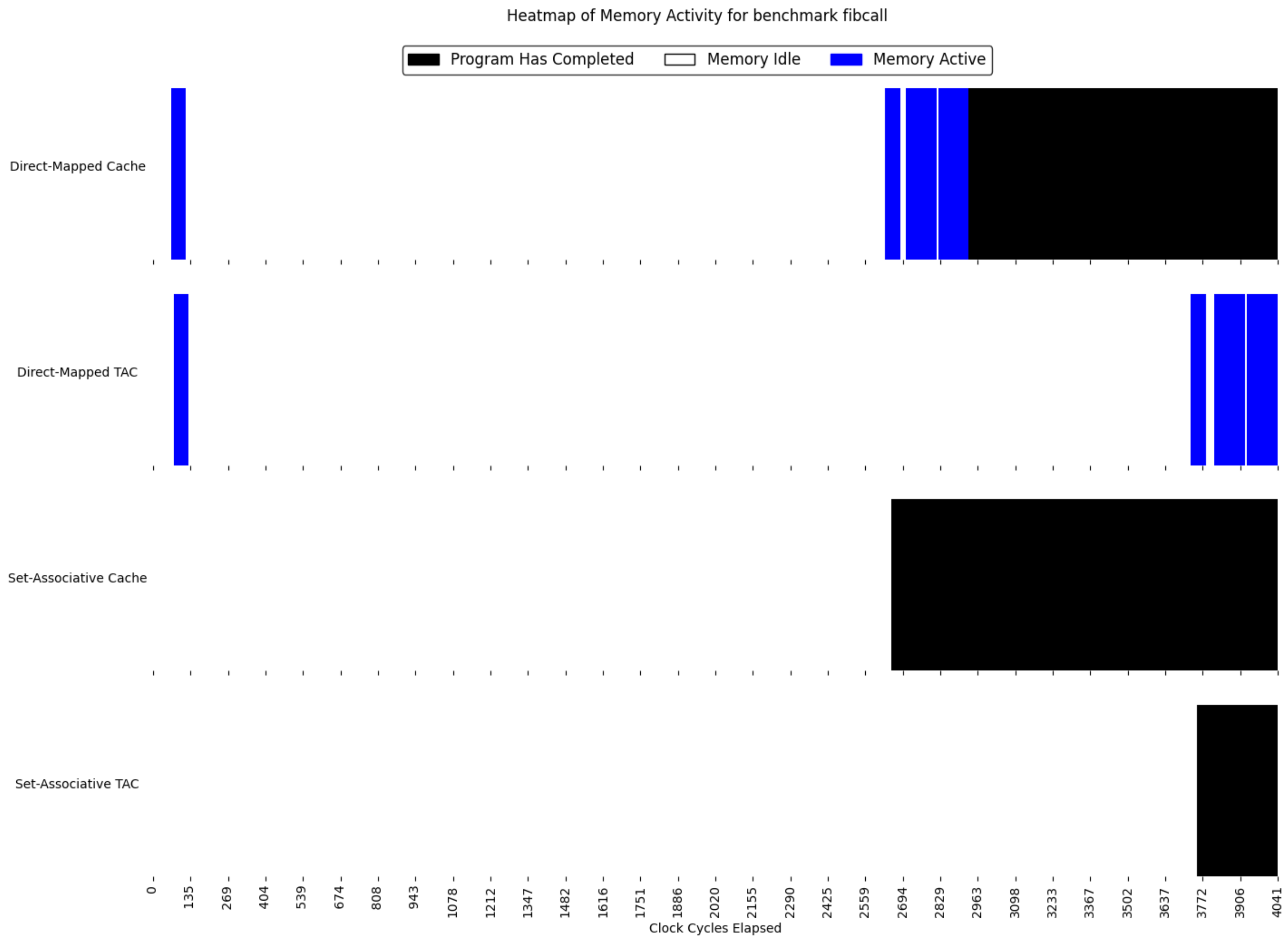


Figure 6.5: The heatmap for the execution of `fibcall`, if we consider the Direct-Mapped cases at the top, we can see behaviour analogous to that when describing `janne_complex`.

potential for overlapping before we begin, because STORE misses will not execute a memory operation. This leads us to the conclusion that everything is done in cache. If we combine that with the fact that fibcall has 70% of the instructions it executes as memory instructions, compared to just below 50% for janne_complex, we see that there are simply more chances to incur the overheads between the two benchmarks. Therefore, despite the fact that the two display relatively similar behaviour, the number of memory operations as a proportion of the total number of instructions executed has a large influence on the performance hit incurred by adding the TAC.

This is an opportune corrective to simplistic thinking around some of the issues we will see in Section 6.5. It is not as simple as saying that the more memory operations (and therefore misses) a program has, the more it will benefit, because having a higher proportion of memory operations to computation instructions also gives more opportunity for overhead costs to accumulate. Any analytical characterisation of programs to try and discover whether Trace Assisted Caching will help will need to take this into account.

6.2.4 duff

Turning now to look at duff we see a slightly different picture again. duff is based on "Duff's Device" [60] a method of unrolling loops in C and forcing the compiler to emit an unstructured loop. In terms of performance we see a 18.93% degradation when comparing the Set-Associative cache and Set-Associative TAC and a 14.98% degradation for the Direct-Mapped version as per Table 5.1. The source code for this benchmark can be seen in Listing 6.5.

Listing 6.5: This listing implements the duff benchmark, based on Duff's Device [60].

```c
#define ARRAYSIZE  100
#define INVOCATION_COUNT 43     /* exec time depends on this one! */


void duffcopy( char *to, char *from, int count)
{
  int n=(count+7)/8;
  switch(count%8){
  case 0: do{     *to++ = *from++;
  case 7:         *to++ = *from++;
  case 6:         *to++ = *from++;
  case 5:         *to++ = *from++;
  case 4:         *to++ = *from++;
  case 3:         *to++ = *from++;
  case 2:          *to++ = *from++;
  case 1:          *to++ = *from++;
  } while(--n>0);
  }
}


void initialize( char *arr, int length)
{
  int i;
  for(i=0;i<length;i++)
    {
      arr[i] = length-i;
    }
}


char source[ARRAYSIZE];
char target[ARRAYSIZE];

int main(void)
{
  initialize( source, ARRAYSIZE );
  duffcopy( source, target, INVOCATION_COUNT );
  return 0;
}
```

First, this benchmark is clearly different to the ones we have considered previously, because it operates on an array of 100 items, as dictated by the constant in the program. As a result, since arrays are contiguous in memory in C and caches work by allocating multiple entries in memory to a single cache entry, this cannot all be stored in the cache at once. As a result, it cannot simply be the case that overheads are overwhelming any benefits. That being said, there are very few gains made by the TACs with only 5 operations occurring preemptively in the Set-Associative case and 6 as per Tables 5.6 and 5.7 in the Direct-Mapped case, as per Tables 5.4 and 5.5.

If we consider the benchmark source code in Listing 6.5 we can see why this is, because there is very little computation at all occurring in this benchmark. Trace Assisted Caching as a technique assumes a reasonable balance of computation and memory access, but since Duff's Device is completely dedicated to array copying, and therefore memory accesses, there is no computation with which to overlap the memory instructions for any benefit.

But it is not only the amount of memory instructions that is important, `duff` executes 1196 memory instructions over the course of its execution, which means just over

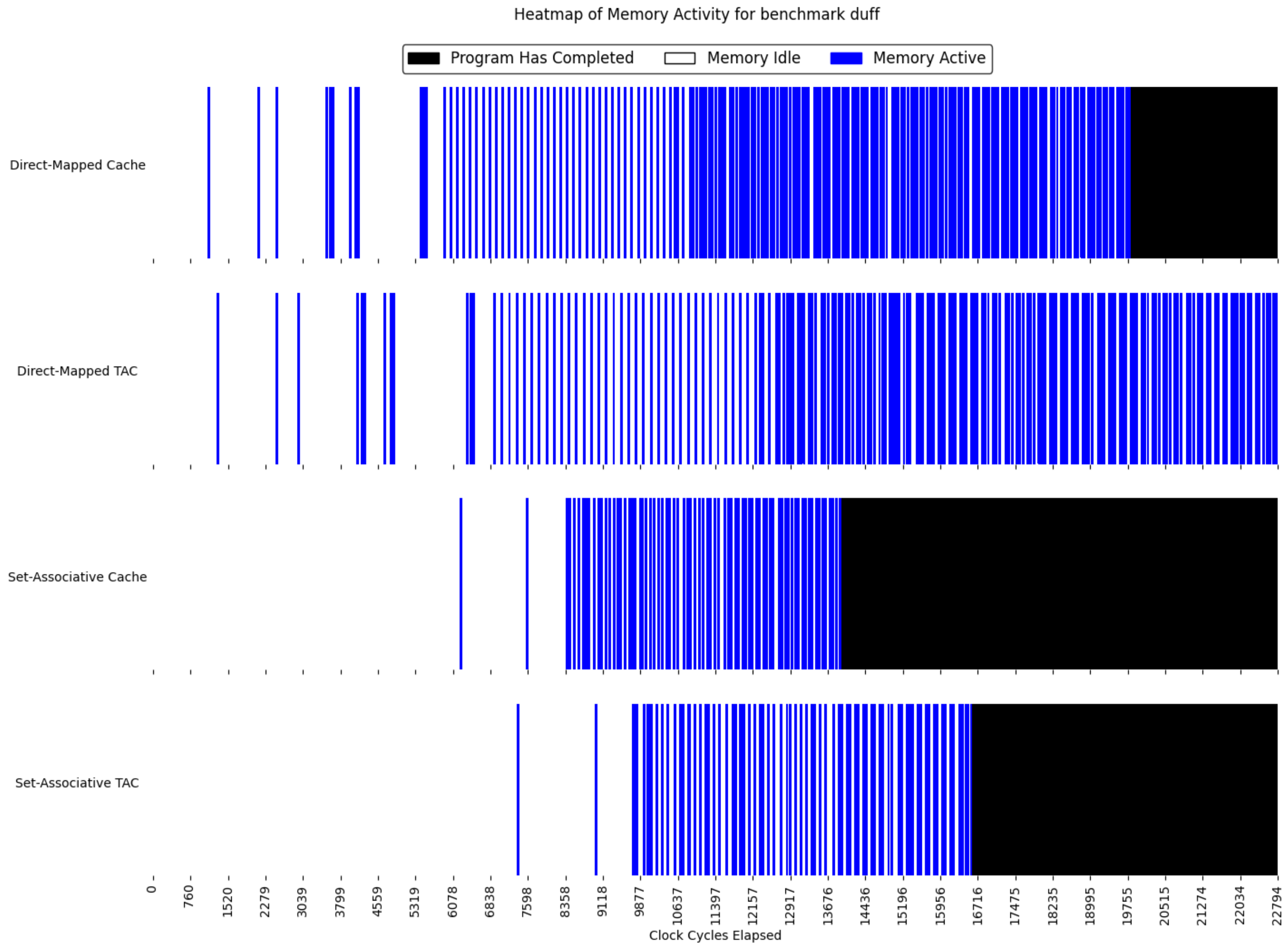


Figure 6.6: A Memory Activity Diagram for the execution of `duff`, we can see that particularly in the later parts of the program the accesses become much more clustered and close together than in earlier segments.

58% of the instructions are memory instructions. However, another benchmark like `expint` has a similar proportion, but only experiences a 7.06% degradation. Therefore, it cannot be this alone that determines effectiveness, but also the spread of those memory instructions. If we consider the Memory Activity Diagram produced for `duff` in Figure 6.6, we can see that the memory accesses are very clustered around particular execution times in the program. Combine this with the fact that we know the accesses are all to a contiguous data store and we can see the copy will be sequential, as per Listing 6.5. This is a recipe for the TAC having no time to make any preemptive moves before being overwhelmed by CPU demand. This suggests that when considering characteristics in Section 6.5, we need to not only consider absolute amounts, but also the spread of memory instructions in the program, as it can lead to programs that have broadly similar miss patterns, but very different performance characteristics, as a result.

6.2.5 `insertsort`

Turning now to consider `insertsort` we see a slightly more positive picture, with degredations of only around 1% between Standard and TACs, as per Table 5.1. `insertsort` implements the standard Insertion Sort algorithm over 10 elements and as such has a very low proportion of misses, even before any Trace Assistance is applied. Considering what we have seen in previous sections, we would imagine this would lead to a much larger degradation, but this is not the case here. Particularly in the Set-Associative case, the TAC manages to overlap 12 more operations than the standard cache, which leads to the difference of only 94 cycles between the Trace Assisted and non Trace Assisted Caches.

The reason this is so successful, relatively speaking, is that even though it is handicapped by the overheads, the majority (65%) of instructions presented have no memory component at all. This means that though the overheads are certainly present, their effect is mitigated, as proportionally there are fewer of them. Again this adds evidence to the idea that parity with the standard caches is not so far away and in reality tweaking the implementation of the TAC to take advantage of situations like this could push us to that point very quickly.

A further point here is that if we consider the disassembled executable file in Listing 6.6, we see there is a large section at the beginning of the program that is very amenable to Trace Assisted Caching. In this section we see lots of `STORE` operations, all to distinct addresses. These would normally be fast in a cache anyway, not requiring a main memory access as the cache is cold, but with Trace Assistance they can all be brought forward and executed very quickly. This benefit will only be accessible in the TAC, generating a stream of misses in a standard cache that could not be avoided.

Listing 6.6: This listing is the decompiled assembly code for `insertsort`, created using a `RISC-V` port of `objdump`. This is only the start of the `main` method of the program, but demonstrates how ammenable the early part of the program is to Trace Assisted Caching, as there are multiple stores made to distinct memory addresses in a startup phase.

```
200   fe010113  addi sp,sp,-32
204   00812e23  sw   s0,28(sp)
208   02010413  addi s0,sp,32
20C   001007b7  lui  a5,0x100
210   0007a023  sw   zero,0(a5) # 100000 <a>
214   001007b7  lui  a5,0x100
218   00078793  mv   a5,a5
21C   00b00713  li   a4,11
220   00e7a223  sw   a4,4(a5) # 100004 <a+0x4>
224   001007b7  lui  a5,0x100
228   00078793  mv   a5,a5
22C   00a00713  li   a4,10
230   00e7a423  sw   a4,8(a5) # 100008 <a+0x8>
234   001007b7  lui  a5,0x100
238   00078793  mv   a5,a5
23C   00900713  li   a4,9
240   00e7a623  sw   a4,12(a5) # 10000c <a+0xc>
244   001007b7  lui  a5,0x100
248   00078793  mv   a5,a5
24C   00800713  li   a4,8
250   00e7a823  sw   a4,16(a5) # 100010 <a+0x10>
254   001007b7  lui  a5,0x100
258   00078793  mv   a5,a5
25C   00700713  li   a4,7
260   00e7aa23  sw   a4,20(a5) # 100014 <a+0x14>
264   001007b7  lui  a5,0x100
268   00078793  mv   a5,a5
26C   00600713  li   a4,6
270   00e7ac23  sw   a4,24(a5) # 100018 <a+0x18>
274   001007b7  lui  a5,0x100
278   00078793  mv   a5,a5
```

```
27C   00500713  li    a4,5
280   00e7ae23  sw    a4,28(a5) # 10001c <a+0x1c>
284   001007b7  lui   a5,0x100
288   00078793  mv    a5,a5
28C   00400713  li    a4,4
290   02e7a023  sw    a4,32(a5) # 100020 <a+0x20>
294   001007b7  lui   a5,0x100
298   00078793  mv    a5,a5
29C   00300713  li    a4,3
2A0   02e7a223  sw    a4,36(a5) # 100024 <a+0x24>
2A4   001007b7  lui   a5,0x100
2A8   00078793  mv    a5,a5
2AC   00200713  li    a4,2
2B0   02e7a423  sw    a4,40(a5) # 100028 <a+0x28>
2B4   00200793  li    a5,2
2B8   fef42623  sw    a5,-20(s0)
2BC   0d00006f  j     38c <main+0x18c>
2C0  ...
```

It also shows an interesting point that differentiates it from the behaviour we see in `fac`, which is that `fac` performs much worse in terms of proportion of unavoidable cache misses, because it is written recursively and compiled in the same way. Because of the implementation of subroutines in the `RI5CY` every `JAL` instruction requires a stack frame and return address to be saved to memory, this causes lots of extra memory accesses that do not actually aid the advancement of the program, but are merely book-keeping to ensure the processor can continue. As `insertsort` is written in an iterative rather than recursive way, this is not necessary. Though we are not exploring compiler optimisations explicitly in this thesis, it is interesting to note that transformations of programs into those that are tail-recursive may well be advisable in this context. Reducing the number of memory operations that occur, as a proportion of the instructions executed will give more space and potential slack to allow memory instructions to be executed preemptively.

6.2.6 `fft1`

Drawing this section to a close let us consider, `fft1`. `fft1` computes a Fast Fourier transform, and from the figures we can see the performance of the Set-Associative TAC only degraded by 1.1% compared to the Standard Set-Associative cache. Of course this is still a degradation but the balance of memory to computation instructions appears to have protected it more than some of the other examples where double digit degradations were seen. It boasts only 8217 memory instructions out of 67965 instructions overall, a proportion of 12.1%, which means there is a lot of space between memory instructions for overlapping to take place. This is shown most clearly in the memory activity graphs in Figure 6.7, where if we consider the Set-Associative Cache, we see a large clump of instructions around 74744 that are spread out more efficiently in the Set-Associative TAC below it. This is also repeated around 164438 clock cycles.

The obvious questions now are: why is it not more effective even than this? And, why do we still see a performance degradation? The only reason that more memory instructions are not brought forward earlier is because of data dependencies in the program. Unfortunately, with code like the Fast-Fourier Transform this is inevitable, simply due to the very calculation being performed requiring multiple pieces of data per output result (as each of the N outputs requires a sum of N terms). This combined with the overheads is what causes the degradation, small as it is, to occur.

This pattern is repeated in other benchmarks as well, with `matmult` being the most direct parallel. Here only 18% of the instructions executed are memory instructions and so the same behaviour can be observed, as shown in Figure 6.8.

6.3 Resolving Problems of the Implementation

Now we have toured over both the general and specific causes of problems for the TAC when trying to match the performance of the standard caches let us turn to some solutions.

6.3.1 Move Towards an OoO architecture

One of the single biggest restraints on the performance of the TAC was the need to maintain sequential consistency within the programs being executed. As a result, it was observed during many of the simulation runs that the preemptive elements of the TAC would sit doing very little for long periods of time if it were the case that there were a lot of dependent memory transactions in particular parts of the program. This was compounded by the fact that it was also observed that once the preemptive parts of the TAC got behind the CPU they rarely caught up again unless there was a marked shift in the number of memory accesses made. Clearly some of this is the result of the overheads and those will be dealt with in the next section, but one option

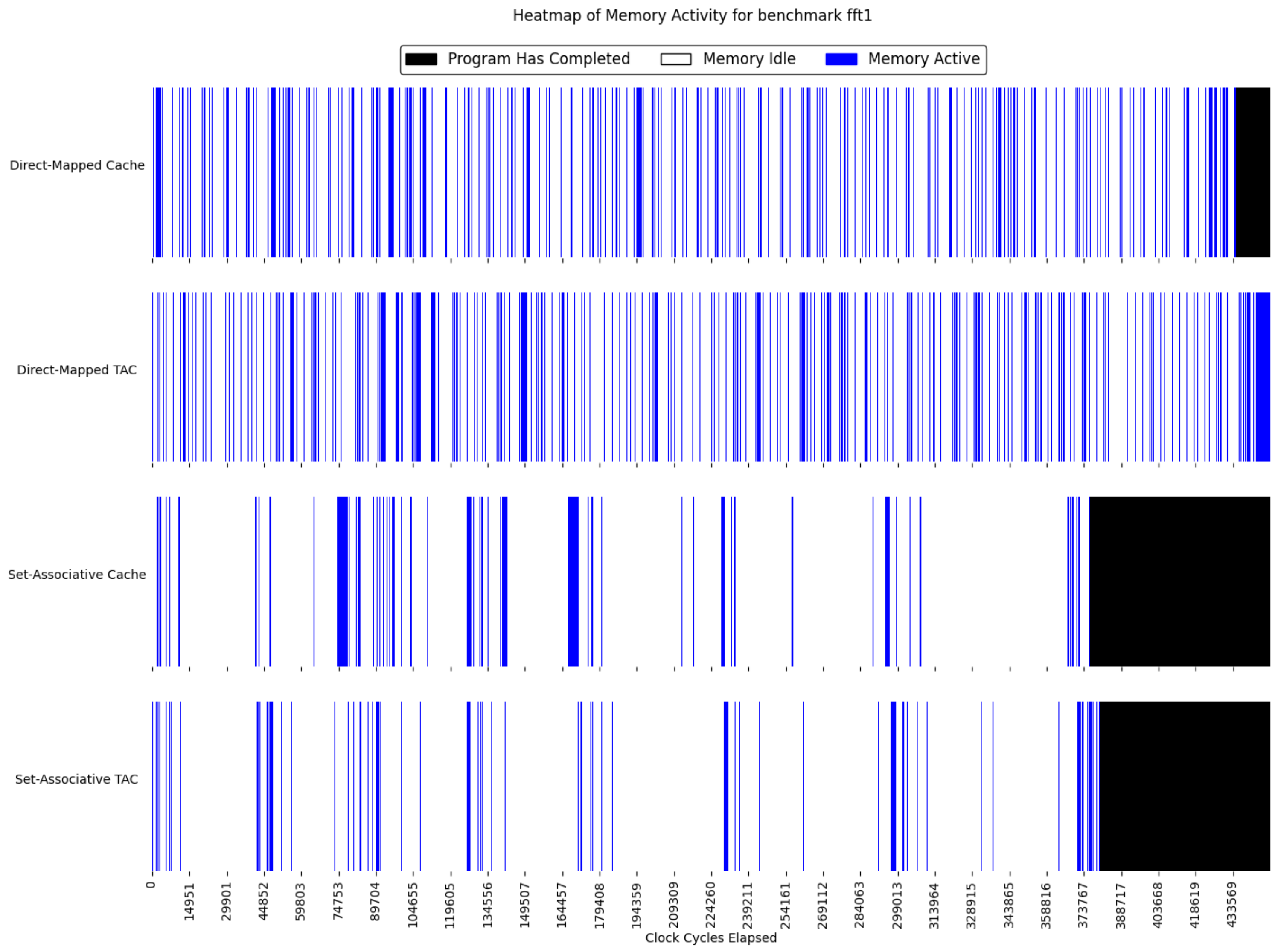


Figure 6.7: The Memory Activity Diagram for the execution of `fft1`, we can see the clustering mentioned around 74744 and 164438 clock cycles.

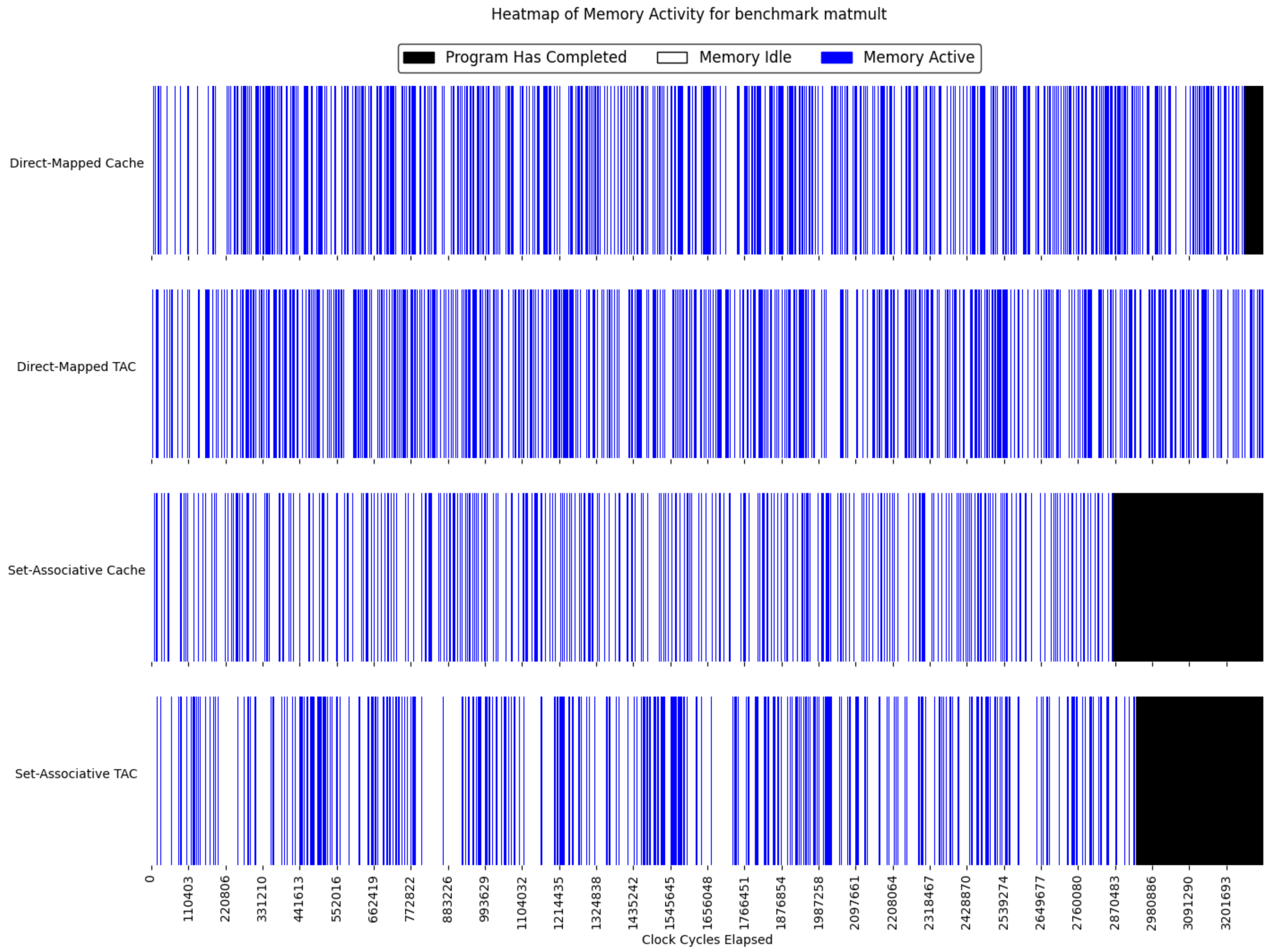


Figure 6.8: The Memory Activity Diagram for the execution of `matmult`. If we consider the Set-Associative section at the bottom, we can see that the clumping is less obvious, but certainly occurs at 441613 and 1545645.

to improve the performance is to relax the sequential consistency constraint the TAC currently imposes.

This could be done in two stages. The first would require implementing something like the scheme seen for Non-Blocking Caches [34, 112] whereby the TAC can analyse if its waiting for a read or a write in the blocking location. The cited papers use MSHRs to maintain the in-flight writes whilst reads can continue assuming they are independent of the writes that are stored in the MSHRs. This should allow the TAC to continue execution pre-emptively even though from a strictly sequentially consistent point of view it shouldn't be able to.

The second phase would be to then move even further towards a completely OoO architecture instead. The big problem that stymies OoO processors, however, is how to decide if certain re-orderings of programs are safe [225], i.e. if you make them, are the semantics of the program changed? And the big problem with calculating this is the aliasing problem referred to in Figure 2.7. As a consequence many OoO implementations are incredibly conservative and don't gain the maximum speedups they could because they have to make guarantees about program semantics. If it were possible to re-engineer the TAC so that it used the information it knew about the running of the program to re-order the memory operations it could conceivably run for longer periods without having to stall and therefore executing many more operations in a way that overlaps with computation. This would be possible because the trace would have the effective address information present within it so there would be no ambiguity as to the memory locations accessed. Consequently re-orderings could be guaranteed to be safe, with the caveat that this relies on the assumption of consistency between the traces and the actual addresses required.

It's important to remember also that moving towards an OoO architecture would increase the salience of solving the inconsistent memory access problem referenced earlier. The move towards an OoO would have to resolve the problem of effectively tracking the progress of a program as it executes so that the trace entries would be correctly associated with the correct processor instructions. Solving this would also reduce the dependence of the TAC on the program producing the same order of memory accesses on repeat runs.

In terms of a quantification this would be difficult to quantify exactly because the data that we have does not give us information about how long, or how many times the TAC was effectively stalled waiting for a memory address to be marked as 'completed'. However it's fairly obvious that the more we can execute preemptively and the longer this can run for the fewer cache misses will occur from the perspective of the processor and consequently the shorter the runtime will be. This is certainly something that is worth exploring, perhaps in future work.

6.3.2 Reducing Overheads

Reducing the overheads that are incurred from the use of the TAC would have the biggest impact on the runtimes that were measured through the course of these experiments. The fact that, amongst other problems, the time to get a cache response doubles, even in the case of a cache hit, is highly undesirable but there are several options that could be pursued to reduce the overheads, ranging in complexity.

The first, and most simple option, would be to invert responsibilities slightly in the case of cache requests that originate in the CPU rather than from the trace repository. If a request like this was made the trace-assisted part of the cache would be entirely ignored and the request would be processed as though it was a request to a standard cache. If just this happened then there would be problems with keeping the state of the trace repository consistent with what's really been executed. However this is where the inversion of responsibility arises, because the cache would then publish it's state the trace repository in an asynchronous fashion so the costly action of having to update both the internal cache representation in the TAC and the trace repository could be cut out.

If that were to happen we could eliminate the need to pass through the `UPDATE_MAPPING` and `UPDATE_TRACE_REPO` states, saving 4 cycles from this operation. That would flow through the other operations as well reducing the length of all of them by at least 4 cycles, if not 8 in the cases where preemptive action is taken.

The second and slightly more involved process would be to further reduce the number of cycles required by integrating more of the components into the same SystemVerilog module. At the moment there are multiple examples of transactions between modules taking 3 or more cycles due to handshakes and acknowledgements. Were those to be integrated there would be no need for the synchronisation so that would be another substantial reduction across all the scenarios. Furthermore integrating more of the modules would reduce the duplication that is present within the components. For example the trace repository has to store a representation of the cache in order to track the state of what has and has not been actioned. Integrating further would remove the need for this because the trace repository could access the state of the cache directly, rather than having to synchronise its own representation. This would be similar to the inversion of responsibilities referenced earlier.

All this together would make it possible to reduce the amount of time taken per scenario that arises in Table 6.3. In the case of simple Hits and Misses to 6 cycles, only 2 more than a standard implementation and the other scenarios by 8 or more clock cycles depending upon the exact details of their passage through the state machine. This would not only have an impact on the overall run-time because each scenario takes less time, but the shorter that the preemptive actions are the more that can be executed in the gaps between memory instructions so this has a double benefit.

6.4 Applicability of Results

We have now seen many specific examples of the TAC degrading performance for specific benchmarks and have also posed some solutions that would allow us to reduce the overheads it introduces. However on question we still have not addressed is: exactly how much are the results we have presented dependent on the design choices made in Chapter 4? This clearly has big consequences for how applicable these results might be if Trace Assisted Caching were to be more broadly applied. Therefore, let us consider a number of questions:

1. How dependent are the results on choosing a `RISC-V` processor, configured as we did i.e. with a simple 4-stage pipeline?
2. How dependent are the results on choosing the `RISC-V` ISA, as opposed to another?
3. How dependent are the results on the particular size of cache chosen and how might a larger or smaller cache impact the results?
4. How dependent are the results on the Mälardalen benchmark workloads exercising the processor in a particular way? How might this be different with other benchmark suites?

6.4.1 Dependence on Processor Configuration

The processor we used in these experiments was a 4-stage `RISC-V` processor, but how much did this impact the results we measured? Clearly if we had chosen an OoO processor, as opposed to an in-order processor, then we would have introduced a large amount of non-determinism into the execution. That would mean this technique would fail to be of much use in its current form. That is not to say the technique would be useless, as the information it provides would be useful, but it would need to be reformulated to work with the new processor type, as described in Section 6.3. So, in that sense, the results are highly dependent on the in-order nature of the processor.

If we consider the length of the pipeline as another architectural choice, would a deeper pipeline have exhibited better or worse performance, from the point of view of the metrics we measured? Deeper pipelines tend to lead to higher clock speeds, as more instructions can be in-flight at once. This could lead to a performance increase, but also this opens up more opportunities for data hazards to arise, as the instruction window widens. Therefore, it is very difficult to say without data whether this would be of benefit or not. From the point of view of the technique, an increased instruction window may actually lead to fewer opportunities to overlap memory instructions, because of the increasing probability that the CPU will want to execute a memory instruction. Clearly more experiments would be needed to confirm this, but it would not be impossible to imagine a situation where the pipeline becomes so deep that the CPU is constantly requesting memory addresses and the TAC is rendered useless.

What if we considered a smaller pipeline, would this have a huge impact? Clearly, reducing the size of the instruction window lowers the probability that the CPU will be executing an instruction with memory implications. It could therefore be argued that the logical conclusion would be to reduce the pipeline size to 0 to maximise the chance for overlapping. While this argument has some merits, there will come a point where the benefit gained will be swamped by the lower overall clock-speed and the data-hazards that are inherent in the programs. There is no point in allowing the maximum degree of look-ahead to the TAC, but then designing a program that has lots of data hazards, so that the TAC is stalled waiting a lot of the time. Therefore, it would seem that pipeline depth could affect the performance of this technique a lot and a lot of further evidence would need to be gathered in order to determine the optimal pipeline length, all other factors being equal.

6.4.2 Dependence on ISA Choice

A further question that is worth discussing is: how dependent are the results on the choice of ISA? The point has already been made that the `RISC-V` ISA is very helpful to us in trying to track memory interactions, because only two instruction `LOAD` and `STORE` allow memory access. Therefore, it is very easy to filter out the instructions with memory implications. However, suppose we changed the ISA to a different, more CISC-like ISA, how might this change our results?

In the first place, the specifics of the ISA would matter a huge amount. For example, if we were allowed to use memory locations as operands to assembly language instructions, it would break one of the key features that allows Trace Assisted Caching to work. This is the fact that we can link instructions to the memory accesses they generate. If we ended up in a situation where that were not true and perhaps non-determinism were involved in the decision of when a memory operand was resolved, that could hamper the ability of the TAC to be effective, because it would be unclear when a preemptive instruction was safe to execute. Microcoded architectures might be more of a challenge too, but again it would depend on the exact details of their implementation, for the reasons stated above.

The key question we have to answer when considering ISAs is: Is there a clear link between an instruction being issued and a memory request being made? If this link involves non-determinism or is in any way ambiguous, the TAC is going to struggle, because it relies on the fact that we can predict when an instruction will be safe to execute preemptively. In an architecture where memory operations are allowed in a broader range of categories, this is much harder to predict. If there is non-determinism, then the likelihood of it introducing errors becomes ever increased.

6.4.3 Dependence on the Size of the Cache

For the experiments performed the size of the cache remained consistent, however varying the size of the cache is one of the easiest ways to affect greater cache performance across a large number of metrics. For example increasing the size of the cache decreases the number of memory addresses that will map to the same address under a Direct-Mapped scheme and increases the number of alternative positions available in a Set-Associative scheme. But how would a reduction or increase in the size of the cache affect the TAC?

In the first place an increase in the size of the cache would cause fewer cache misses which would mean there would be less work for the TAC to perform as the increase in the size of the cache is effectively doing the work of the TAC, by increasing the number of options available for data placement. This should allow a reduction in the number of dependent memory operations, particularly in the Set-Associative case and so that should allow the TAC to progress further without effectively stalling. That being said however, this will only work up to a point, as the cache comes closer and closer to the size of the working set of the program the effectiveness of Trace Assisted Caching will actually decrease because the fewer memory operations there are to overlap the less 'latent performance' there is within the program. This is shown by benchmarks like `fibcall` where there is very little memory traffic once the working set is loaded into memory, and the performance is degraded as a result. Consequently were the size of the cache to be increased to this level for each benchmark we would see similar behaviour.

A reduction in cache size would probably have a similar effect for the opposite reason, as the cache size reduced and more and more memory traffic was seen there would be few opportunities to overlap memory and computation, leading to to the TAC potentially being overwhelmed by the amount of operations the CPU required. An increase in the number of cache misses (due to the lower cache capacity) would also lead to a corresponding increase in the need for `LOAD` writebacks so it would make it twice as difficult to perform these instructions preemptively, as they are twice as long as a standard memory interaction. Therefore perversely, reducing the size of the cache could have an even greater impact as there would be more memory operations to overlap but without a corresponding increase in computation instructions to hide this difference.

6.4.4 Dependence on Choice of Benchmark

To close this section it's important to consider the impact of our choice of benchmark upon the outcomes we have seen. The Mälardalen benchmark is designed to assess a processor's performance in the field of WCET calculations an as such contains within it lots of features that may not be found in standard programs. Many of the benchmarks contain multiple nested loops, loops with bounds fed in by the user, matrices, recursion and unstructured code, all features that a good WCET estimation tool should support but not representative of the kind of programs that most benefit the TAC.

For example benchmarks like `janne_complex`, `fibcall` and others that don't exhibit a huge benefit when using a TAC are really concerned with the control structure in the program and making that difficult to analyse. This is in contrast to other benchmarks like `fft1` which, whilst still performing computation, better balance the number of computation and memory-based instructions, leading to better performance from the TAC overall. While it is good to exercise the difficult cases to give a realistic idea of the limitations of the technique it would have been beneficial to gain an idea of the performance on a more representative set of benchmarks.

In that case its conceivable that the average level of performance degradation would go down because a key determinant of performance, and this is explored further in 6.5, is the mix of computation and memory-based instructions. A different benchmark suite, that was less concerned with producing corner-cases for WCET would have provided this and could have reduced the average degredation significantly, especially if it could have avoided the computation or control flow focused benchmarks, like `fibcall` and `tac`. Interesting future work would be to look to turn this question around slightly and to try and generate a benchmark set that obtained the optimal performance for the TAC but this would be intimately related to the problem presented in Section 6.5 or deciding on the characteristics of programs to accentuate and whether it's possible to generate those in an automated fashion or not.

6.5 Programs That Benefit from Trace Assisted Caching

So with all this information in mind the question that seems most prescient is: can we characterise the kinds of programs that are most amenable to Trace Assisted Caching? Some of the in depth analysis of the benchmarks presented in this Chapter do start to point towards a few key features that correlate to increased performance from the TAC, but this section will make those connections explicit.

The first of these is relatively straightforward, in that Trace Assisted Caching works best when applied to programs that have a capacity to be improved, i.e. have a relatively large proportion of misses that could be overlapped and converted into hits. This is what we see in the negative in benchmarks like `fibcall` and in the positive in `insertsort`. However, some caution has to be applied here, as it is not as simple as suggesting that a high miss count will lead to better TAC performance, as shown by Figure 6.1. This leads to the second point, there must be space within the program for the overlapping to occur. It is not good having a very high miss count, because there are so many memory requests that the memory system could never hope to satisfy them. Consequently, there is a balance to be struck between a high enough proportion of misses to allow overlapping and enough space between memory operations to allow the overlapping to occur. This is something that should be quantified, but would require a different set of experiments that explore how artificially increasing the miss rate and density of memory instructions alters the performance of the TAC, something the current data we possess does not allow.

A third point is that the benchmarks most amenable to Trace Assisted Caching generally have a much higher proportion of computation instructions to memory instructions, as we saw in our examination of `fft1` and `matmult`. This happens because the more computation instructions introduced, the more time there is for instructions to be overlapped. Thus, the better the performance overall. Of course, this is also a reflection of the fact that, in general, the fewer memory operations you have, the faster the program is going to run, due to the fact that instructions that do not need to travel 'off-chip' will take much less time than those that do. Of course, this is not to suggest that artificial instructions should be added in order to increase the proportion of computational instructions, but if a program is already computationally intensive, this could be a good candidate to be assisted by Trace Assisted Caching.

A final characteristic is something of a combination effect and we see it in benchmarks like `duff`, where the reason it is so problematic for the TAC to help is because all the memory accesses are 'close together' in memory. This means they will not distribute well over the cache, but also they are all dependent on each other. This leads to high amounts of misses that are difficult to counter, because they occur in a cluster and rely on data that is not produced until it is too late for the TAC to act. Consequently, a program will be amenable to Trace Assisted Caching if it can avoid clusters of memory accesses that have data hazards between them and where address computation for those accesses cannot occur until very close to them. This may need to be enforced by the compiler rather than by the programmer, but if this were the case it is much more likely that Trace Assisted Caching would be effective, because, unlike in the case of `duff`, more of the operations have a higher chance of being overlapped.

6.6 Summary

In summary, in this analysis we have considered the data created by running the same set of Benchmarks over 5 different hardware variations and observing the differences in performance between them. We have seen that while the TAC did not outperform the standard caches on any benchmark, the reasons for that are not unknown and some solutions proposed. We have also formulated a set of loose criteria that appear to characterise those benchmarks that are particularly amenable to the technique and it should be possible to use those to predict, based on measurements of the programs running on standard hardware, whether it is worth investigating a TAC to improve the performance of the program.

In the final chapter we will consider our results in the round and we will also consider what future work this informs. Finally we will present how this research fits into the larger picture of the field as a whole and how that should influence future work going forward.

7 Conclusion & Further Work

In Chapter 1 we posed the following research questions:

1. Can feeding trace information back to a cache be used to decrease the runtime of programs?
2. Can this approach outperform a processor running the same computation but with a standard cache?
3. Does this approach introduce any overheads when compared to a processor with a standard cache?
4. Under what circumstances does the addition of trace information give the maximal benefit?

In order to answer these questions we designed new pieces of hardware and measured the impact those hardware changes had on performance. These measurements were both from the perspective of the overall runtime of the program and from the perspective of the cache behaviour the hardware exhibited. From doing this we are now in a position to effectively answer these four questions, place our work within the context of the wider field of cache design, propose future work to extend what has been achieved and finally to summarise the contributions of this thesis.

7.1 Answering the Research Questions

In terms of the first research question, the results presented in Chapter 5 and analysed further in Chapter 6 definitely show that feeding back trace information can reduce the runtime of programs. We consistently saw through the presentation of the results that there were no circumstances in which having a TAC actively degraded the performance of the processor, as compared to not having a cache.

The next research question however we have to answer in the negative, despite the fact that in some cases the degradation was only 1% we never observed a situation whereby the TAC outperformed a cache of the same size when running the same computation. Throughout Chapter 6 we analysed some of the reasons for these performance degredations; which chiefly revolved around the introduction of unacceptable overheads. These overheads prevent the gains the TAC makes in overlapping computation with memory instructions from having more of an impact. This in turn answers the third research question, as we saw in Chapter 6 and the work in Appendix C, overheads are introduced for every code path through the TAC when you compare the same operation taking place in a standard cache, the most egregious example being that the time to service a cache hit doubles when using a TAC. The reason for a lot of these overheads is the need to synchronise state between the different modules that make up the hardware as well as the protocols for communicating between them, in Chapter 6 we explore solutions for resolving this in a future version of the hardware.

Turning to the final research question, this thesis has allowed us to begin to understand the characteristics that make certain programs much more amenable to Trace Assisted Caching than others. Examples of this include having a high proportion of misses that can be overlapped, having a good proportion of computation instructions and memory accessing instructions, and having the data that is to be used well spread out in memory. These metrics would need to be made much more precise and the relationship between them and the performance of the TAC much more rigorously defined to be able to use them as predictive factors of performance. In order to do this you would need to define many more experiments where these factors could be varied independently to establish the sensitivity of performance to these factors. In this thesis we've certainly shown they do have an impact with the examples presented, and it will be for future researchers to precisely quantify their impact and the sensitivity of performance to these measures.

In summary this thesis has presented a system that does reduce runtime but is restrained from producing a net gain because of the overheads it incurs in its operation. There are solutions the thesis posits as to how this could be mitigated with further research, but at the present time a TAC will perform 8% worse on average than a standard cache of the same size, while consuming many times the resources of the standard cache.

7.2 Contributions

With all that in mind what contributions to knowledge does this thesis make? The first is that it presents a novel technique for reducing the overall runtime of programs in Trace-Assisted Caching. This technique works by the recording and application of traces to running programs to effectively allow incredibly accurate prefetching. This is demonstrated through experiments which show runtime reductions compared to a standard CPU with no cache. The technique does not perform as well as standard caches, but the thesis presents several potential solutions that when analysed show that a net gain should be possible with changes to the particular implementation of Trace-Assisted Caching presented here.

The second contribution is a methodological one, the trends in cache research up to this point had very much been in the vein of cramming the largest cache possible onto the available silicon. However this thesis shows that there is potential benefit to investigating caches that are more integrated into the systems they are part of, not only being ‘conscious‘ of themselves but of the state of other components in the system. Further to that this thesis provides a software toolkit [171, 172] that can be used to investigate caches both in simulation and in real hardware, with measurement and end-to-end orchestration built in. In summary we have not only started to open up a new vein of research but are also providing the tools to allow others to go even further.

7.3 Future Work

Of course there is much future work that could be done to expand the scope of this work and increase the performance of the TAC. Some of this is detailed below, with some brief discussion of what solutions in that area might look like.

7.3.1 Applications to High Performance Computing

Our work with Trace Assisted Caching has focused on medium-sized embedded systems and this has driven the design decisions made and the choice of processor upon which to experiment within this thesis. However, if we consider some of the work done in the area of HPC it may well be possible to apply Trace-Assisted Caching here too. The essential principle of the technique would stay the same but there are several issues that would have to be solved before HPC applications could be effectively evaluated.

First would be how does Trace Assisted Caching work in a world of multiple processors, multiple cores per processor and multiple levels of shared caching? Of course this is becoming a problem for embedded systems too due to the advances of Moore’s law, but in HPC applications this is even more pronounced. To some extent this depends on the coalescence of future work around a standardised design for shared caching and cache coherency in the world of multi-core but if we consider an architecture with dedicated L1 caches and shared L2 caches, trace-assisted caching could be used either locally, so it only influenced the movement of data in the L1 caches or globally.

The second issue is also around how we apply this kind of work to more complex cache architectures, common in more complex processors. In recent papers [78, 79, 207] we've seen 3 or 4 levels of caching as standard and even moves to define the whole cache architecture from a pool of generic resources so it won't be known until runtime. For this thesis trace-assisted caching was proposed as operating over a single level of caching but with multiple levels of caching it would be possible to use it influence not only the movement of data between main memory and the cache but between individual levels of the cache itself. This is of course speculative but it could be possible to reduce the number of misses in lower levels of the cache, where they are more costly, by more tightly optimising the contents of the higher cache levels through Trace Assisted Caching.

7.3.2 Improving the Fidelity and Stored Size of Captured Traces

One of the key assumptions made in this thesis was the relative predictability of memory accesses, specifically that between runs of a program the memory accesses would be consistent, allowing one run to be representative of the program as well as allowing the trace repository to consistently match its progress with that of the CPU. Clearly in reality this isn't the case and two key problems arise, the first is that if commands like `malloc` are used within the program, this introduces non-determinism which clearly would break the two previously stated assumptions. The second problem is one of programs whose path of execution is dictated by data provided at runtime. In the pathological case this might look like a `switch` statement at the start of the program that, based on user input chooses one of several very different code paths. In order to make trace-assisted caching more resilient to these cases, there are several enhancements that could be made.

The first thing to improve would be to give the trace repository the chance to aggregate data across multiple runs of the program, augmenting each location with some kind of probabilistic measure. At present we do a single run of the program and then attempt to improve the performance from that trace, but one run of a program with non-determinism does not capture the depth and complexity of the addresses it accesses. Consequently there should be some mechanism to allow the trace recorder to record and aggregate multiple runs of the program. Hence, when the cache is trying to make a decision, it can pick the most likely rather than most recent memory address for each memory instruction. A further enhancement here might be, in concert with the compiler and linker, to mark instructions as non-deterministic and to therefore ignore them. Clearly this reduces the potential for memory operation overlapping but if there is no practical way to predict them it may be more advisable to accept the limitation.

Along a similar line, trying to reduce the size of the traces that need to be stored 'on-chip' would severely reduce the resource burden that trace-assisted caching places upon any system. At the moment, as was demonstrated in Chapter 4, the trace repository is one of the biggest consumers of resource and the reason for that is the 131072 entry BRAM array it maintains to store the traces. One way to severely reduce that would be to apply the concepts from memory hierarchies and perform wider reads for say 4 or 8 trace entries at a time, from a very cheap but slow external memory store, throwing away old entries as their utility expires. This already happens in the TAC to some extent where it maintains a working-set of the in-flight memory operations, a similar idea could be applied to the trace repository.

7.3.3 Quantifying the Link Between Slack and Effectiveness

One of the key elements of Chapter 6 was the recognition that there are some features of programs that make them more amenable to this technique than others. Part of that discussion is around a quantity called slack that we can loosely define as the aggregate amount of time the memory bus that connects the CPU and cache spends idle. It would be of great use to the study of Trace Assisted Caching to more formally quantify what this quantity exactly is and how it links to the effectiveness of the technique overall. For example, we know that simply measuring the total amount of time the memory spends idle is not sufficient, because questions of distribution are in play too, so there is a lot of very fertile research to be done to tease apart those ideas. If the ultimate outcome of this research is a way to learn, either from source code or from a compiled executable, whether a program is likely to benefit from Trace Assisted Caching, it would be a hugely positive step forward.

7.3.4 Expanding the TAC to Other Processors

A final piece of future work would be to expand the TAC to support other instruction sets and other processors. At the moment, there is only support for the `RISC-V` ISA and for only one processor that implements that instruction set. This was chosen because it was adaptable and open, but the real challenge would be to define common interfaces this system can use, so that it could conceivably be deployed into any system that provided the information it needed. Several challenges would have to be overcome, including the fact that in more CISC-like processors more than just `LOAD` and `STORE` can access memory. In addition, the inclusion of microcode or microops can make the matching of causes to effects more difficult, especially if the new processor is an OoO processor, as suggested in earlier sections.

It would also be positive to quantify exactly what information is required to enact Trace Assisted Caching and to see if there is a correspondence between that and the information provided in the Nexus Debug Standard[87] at any level. This would allow us to provide a more portable alternative, based on widely accepted standards. This was processor manufacturers and designers could easily know what information is required rather than having to re-engineer a bespoke solution for any new processor that may want to use the system.

Part V

Appendices

A Trace Recorder (Gouram) Implementation

To implement trace recording we need a way to track the execution of each instruction as it passes through the various pipeline stages of the processor. Further to that, we need to track the effective addresses of each memory instruction as it is generated by the running program. The basic construction for this new piece of hardware will be made of two sub-modules. The first will track the `IF/ID` phase of the pipeline execution and the second will track the `EX` phase. We do not need to track the `WB` phase, because it will have no bearing on either the effective address or the timing of other instructions as it is merely a formality that results get written back to registers. This overall architecture can be seen in Figure A.1. First to help us understand how each phase will work it is important that we understand the memory protocol that is implemented by the `RI5CY` processor. With that in hand we can move forward to describing each of sub-blocks and then the overall trace recorder module and the data it produces.

A.1 `RI5CY` Memory Protocol

The memory protocol that is implemented by the `RI5CY` is documented in the processor manual [12]. However, it bears slightly further explanation. There are certain parts of the protocol that we will rely on or have to workaround in order for Gouram to function correctly. To begin, there are 8 signals that the Load-Store Unit (LSU) uses to communicate with the memory hardware and these are listed in Figure A.2, reproduced from the processor manual:

The protocol then proceeds thus. When an instruction requires access to memory the LSU sets `data_req_o` high, whilst at the same time placing the calculated address (`data_addr_o`), byte enable bits (`data_be_o`), any data to be written to memory (`data_wdata_o`) and selecting a read or a write with `data_we_o`. Then the processor waits for the memory system to respond by setting `data_gnt_i` high. Once this has happened, the processor can change any of the 4 signals it set originally, assuming them to be cached in the memory controller now `data_gnt_i` is high. This may

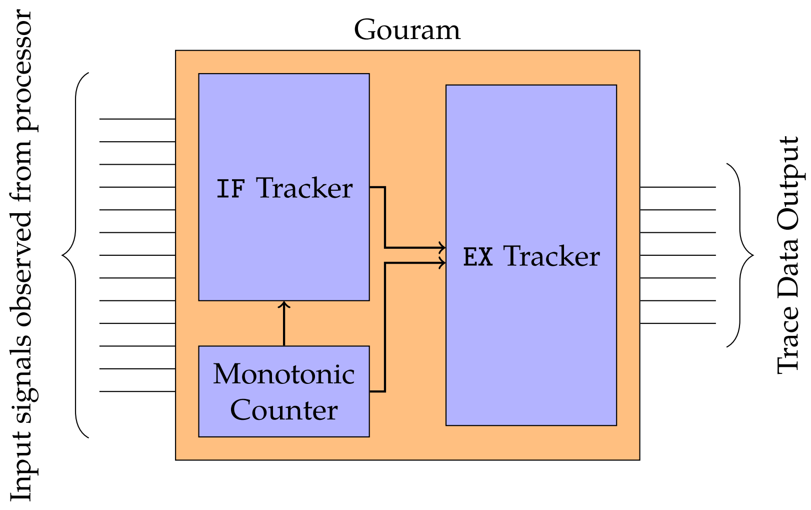


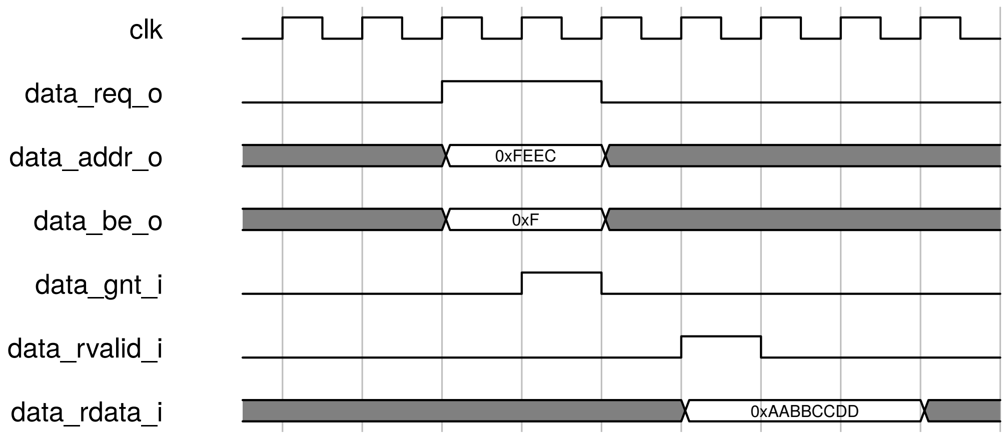
Figure A.1: The modules and connections that make up Gouram and its tracking capabilities. There are other ancillary connections that are omitted from this diagram, including the clock and reset lines for each module, to add clarity to the diagram.

Signal	Bit Width	Direction	Description
`data_req_o`	1	Output	Request ready, must stay high until `data_gnt_i` is high for one cycle.
`data_addr_o`	32	Output	Address
`data_we_o`	1	Output	Write Enable, high for writes, low for reads. Sent together with `data_req_o`.
`data_be_o`	4	Output	Byte Enable. Is set for the bytes to write/read, sent together with `data_req_o`.
`data_wdata_o`	32	Output	Data to be written to memory, sent together with `data_req_o`.
`data_rdata_i`	32	Input	Data read from memory.
`data_rvalid_i`	1	Input	`data_rdata_i` holds valid data when `data_rvalid_i` is high. This signal will be high for exactly one cycle per request.
`data_gnt_i`	1	Input	The other side accepted the request. `data_addr_o` may change in the next cycle.

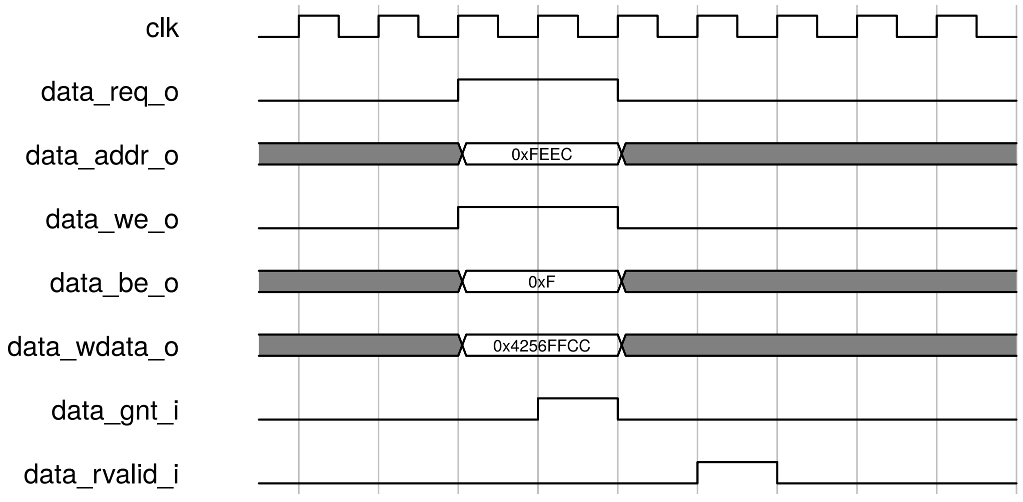
Figure A.2: List of input and output signals provided by the LSU to implement the memory protocol for the RI5CY, reproduced from the processor manual [12].

happen in the same cycle that `data_req_o` goes high or it may take several cycles with a slower memory technology.

After the grant, the memory system will execute the load or store as required and once it has completed it will set `data_rvalid_i` high. This will happen after at least 1 clock cycle from the setting of `data_gnt_i` to high. Once `data_rvalid_i` is high `data_rdata_i` will contain the fetched data from memory in the case of a `LOAD` or arbitrary data in the case of a `STORE`. If another memory request is queued, `data_req_o` will be set high at the same time that `data_rvalid_i` is and the processor continues. Several examples of timing diagrams are included in Figure A.3 to illustrate how this protocol works.

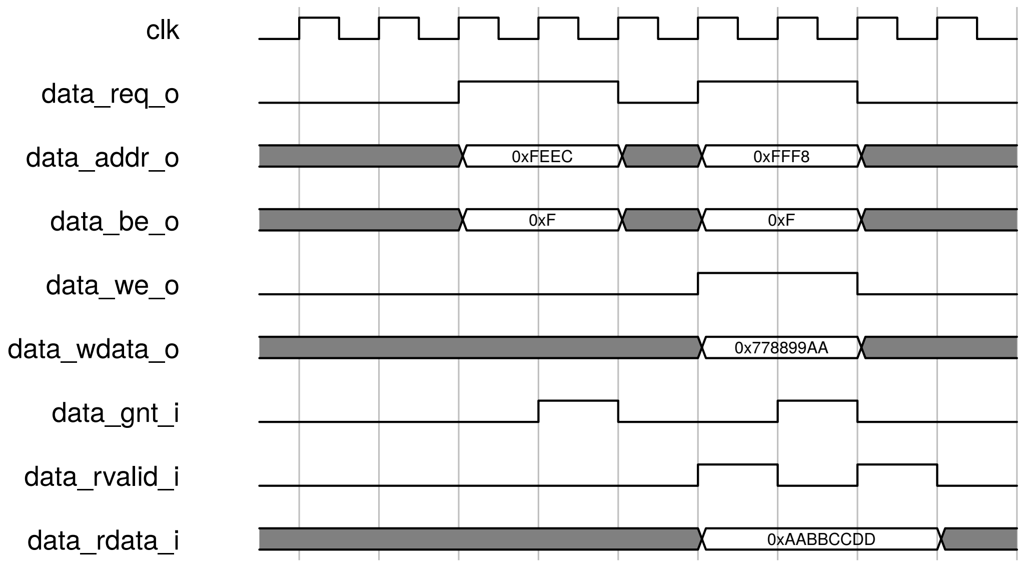


(a) A `LOAD` instruction to retrieve the contents of address `0xFEEC` from memory.



(b) A `STORE` instruction to `0xFEEC` from the processor.

It should be pointed out that this protocol works for the instruction memory as well, but uses a reduced number of signals, as the instruction memory cannot be written to. These signals are labelled `inst_req_o` etc.



(c) A `LOAD` followed immediately by a `STORE`

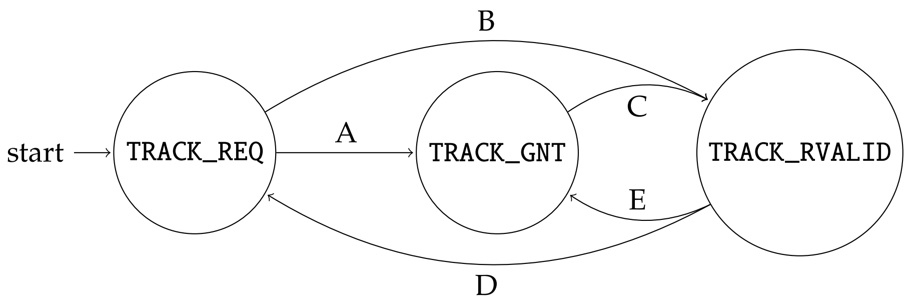
Figure A.3: Several examples of the signal transitions that occur when memory transactions happen. More of these diagrams can be seen in the `RI5CY` instruction manual. It should be noted, however, that the diagram in the manual labelled "back-to-back" does not occur in our context, because of the memory implementation used.

A.2 The `IF` Module

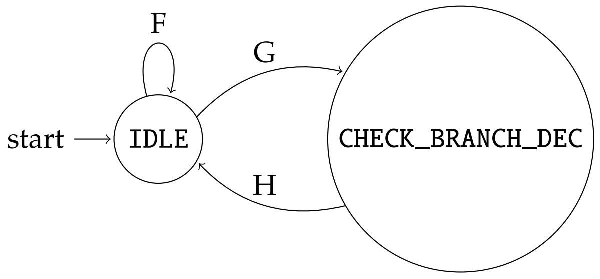
Now that we have the memory protocol in hand, we can begin to construct submodules to record various parts of the execution of instructions. Looking at this generically, the first thing that will happen is the instruction will be fetched from the instruction memory. As we can have no idea what the instruction is going to be until it has been fetched, we have no choice but to track everything fetched by the processor and to then throw out the non-memory instructions later. This leads us to defining a module called the `IF` module, which will follow a predefined state machine. The machine and its transitions can be seen below, and by example we will work through each of the transitions to describe its function. The SystemVerilog code that was produced for this and subsequent hardware pieces can be seen in Zenodo [172].

A.2.1 Instruction Fetch State Machine

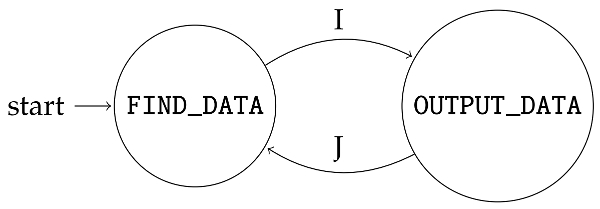
Upon boot, each of the state machines will be reset to the `IDLE` state and each clock cycle will check for the signals necessary to transition to the other states. The first thing that will happen is the program counter will attempt to fetch the next instruction,



(a) Instruction Fetch State Machine - This effectively tracks the internal state of the fetch process



(b) Branch Decision State Machine - This tracks if a branch decision needs to be made and acts accordingly.



(c) Output State Machine - This uses the results from the first two state machines to output the correct data to the next phase of the process

Figure A.4: The three state machines that make up the execution of the IF Module inside Gouram. These state machines all work concurrently as per the semantics of SystemVerilog.

as pointed to by the program counter. This will cause the `instr_req_o` signal to be set high, which will trigger the `IDLE` state to make a decision as to whether to take transition A or B. The RI5CY manual defines that `instr_gnt_i` could become high in the same clock cycle as `instr_req_o`. If that is the case, we then need to transition to `TRACK_RVALID`, otherwise the `gnt` signal will be missed, so transition B is taken. Otherwise, it is transition A. If transition B is taken then the instruction address and the value of monotonic counter are stored, as they could change in the next clock cycle, which means the information would be lost. This data is stored in a buffer that builds up the required information over time and then sets a flag to mark it as ready to be passed to the next phase.

If transition A is taken, the state machine then waits for `instr_gnt_i` to go high and then captures the same information as described previously. In either case, once we arrive in state `TRACK_RVALID`, the state machine waits for `instr_rvalid_i` to be set high and when that happens the instruction data is captured into the buffer along with the time at which the `instr_rvalid_i` went high, again from the monotonic counter. After this, either transition D is taken to bring us back to waiting for a new `req` signal or because it is possible for `rvalid` and `req` signals to overlap, it is also possible that we might have to transition back to `TRACK_GNT` instead. This is covered by transition E.

A.2.2 Branch Decision State Machine

It is easy to think at this point that our job is complete and we should simply output the data now it has been captured. However, the problem with this is that the instructions that are fetched could very easily be a `JUMP` or `BRANCH` instruction and this is not resolved until the Decode phase. Furthermore, because that takes a non-trivial amount of time, it is often the case that the processor will fetch instructions that are never actually executed, because they are invalidated once the branch or jump has been taken. Consequently, before we decide if we can output the instruction we have just captured, we have to wait for its decode phase to end and any branch conditions to be calculated. This is especially complicated by the fact that the instructions affected are the ones that occur in the window between the calculation of the jump or branch address and the successful fetch of the branch instruction. This is shown in Figure A.5.

When we want to track instructions that execute while a branch condition is being calculated, we work in the following way. When the decode phase is complete for a potential branching instruction, the code will extract the instruction from the tracking buffer and also will test whether this instruction was granted after the last branch

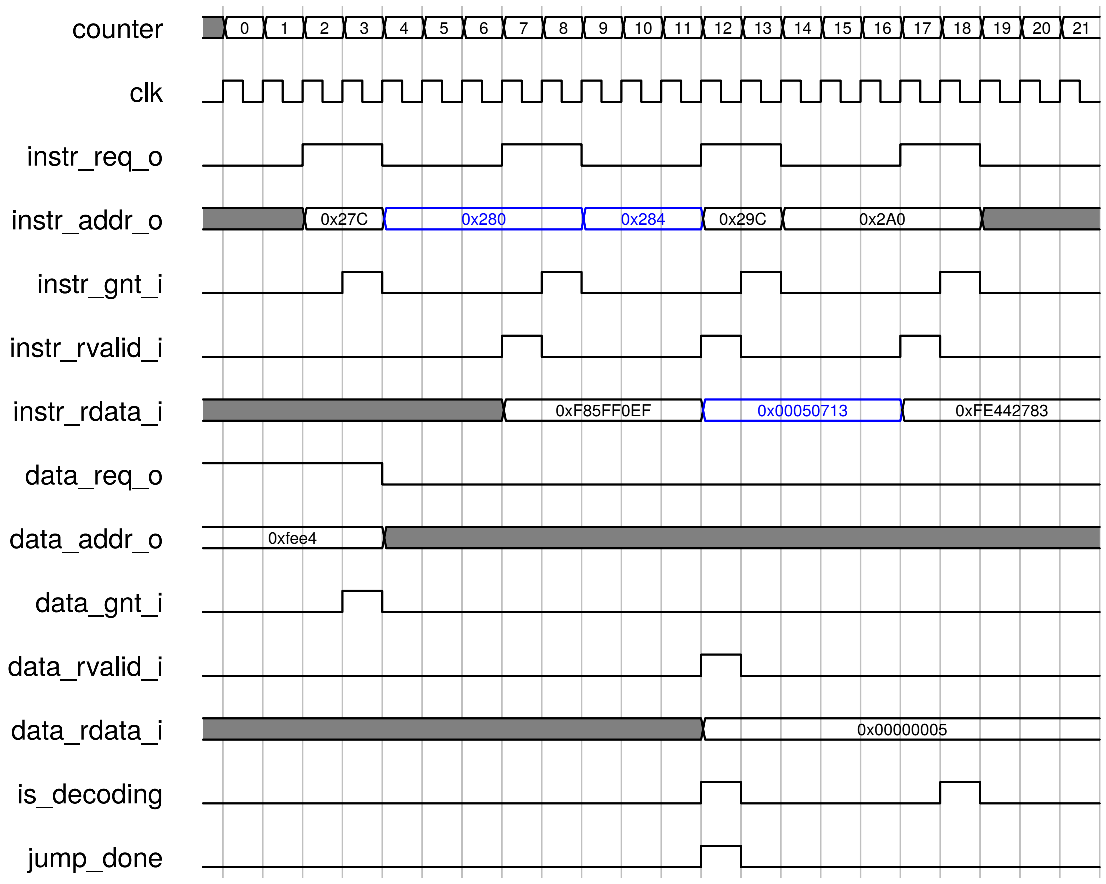


Figure A.5: Due to the long running data memory fetch that completes during Clock Cycle 12, the decode phase for the fetched `0xF85FF0EF`, a jump instruction, does not get decoded until Clock Cycle 12. Due to the architecture of the `RI5CY` processor jump addresses are calculated in the Decode Phase, so the decision as to whether to jump or not is not made until Clock Cycle 12 either. However, as the blue highlighted signals show, the instruction at address `0x280` has already been fetched when the decision is made to jump to address `0x29C`. This occurs, and so the next fetch is correct and the processor has a method of ignoring these incorrect fetches. This means we have to implement something similar so we do not track instructions that never happened. This example explicitly targets jump instructions, but the delay is even more pronounced in branch instructions, as the target address, and by proxy branch decision, are calculated in the `EX` phase.

decision was made. For the sake of argument let us assume that is true, so we are not at present in a branching state. The next operation will be to store the time at which the decode phase ended and then we will test what kind of instruction we are dealing with. This is important because there are 3 situations that could arise here:

1. The instruction is not a branching or jump instruction and has not been output while the processor is in a branching state. In which case we can simply output this into the next module in the tracker.
2. The instruction is a jump instruction, in which case it will be executed immediately, because the calculation of the address is done in the Decode phase for optimisation reasons.
3. The instruction is a branching instruction, so we have to wait for the end of the execution phase before we know if we have to jump.

If we are in situation 3, we trigger the Branch Decision State Machine to take transition G. We also set the `branching` variable such that we now know that we are waiting for a branch decision to be made. This means that no more of the tracking buffer will be processed until we have reached a branch decision. Once that happens, we store the time at which the branch decision was made, what it was and update the cut off time accordingly. We also mark the instruction as ready for output in the tracking buffer. After this, transition H is taken to move us back to the `IDLE` state. If we had been in situation 2, something similar would have happened, but it would have happened immediately after the end of the decode phase.

A.2.3 The Output State Machine

As all the state machines are asynchronous, except for where they explicitly synchronise, the last state machine is comparatively simple. The `FIND_DATA` state simply sits and scans through the tracking buffer to see if there are any pieces of data that can be output. If it finds one, it checks to see if it is a `LOAD` or a `STORE` instruction and it also checks that it was not fetched during a period where a branch decision was pending and eventually taken. This stops the situation where the processor eagerly fetched the next sequential instruction, but actually ended up branching and so executed a completely different instruction. At this point it takes transition I to the `OUTPUT_DATA` state. Once in that state, the data is transferred to the next module that makes up Gouram, along with some other data to help find the length of the execution phase. The data is marked as having been output and then transition J is taken and the process begins again.

A.3 Examples

What is described is very complicated, because tracking the behaviour of the processor as it calculates branches is highly non-trivial. The best way to explore how this works is by following two examples from end to end. The first example will be the instruction at address `0x258` in Figure 3.5, which is `0x00112e23`. The second will be a more complex branching example by following `0xfce7dae3` at address `0x2a4`, which also appears in the same figure.

A.3.1 Simple Load Example

The diagram of the signals issued by the memory system can be seen in Figure A.6. We will also track the contents of the tracking buffer and the state machines so we can see how all of the aspects of this fit together. An empty entry in the tracking buffer and the state machines at the start of the execution can be seen in Figure A.7 respectively. The names of the states in the machine have been shortened into acronyms for reasons of space, but they are the same as the diagram in Figure A.4.

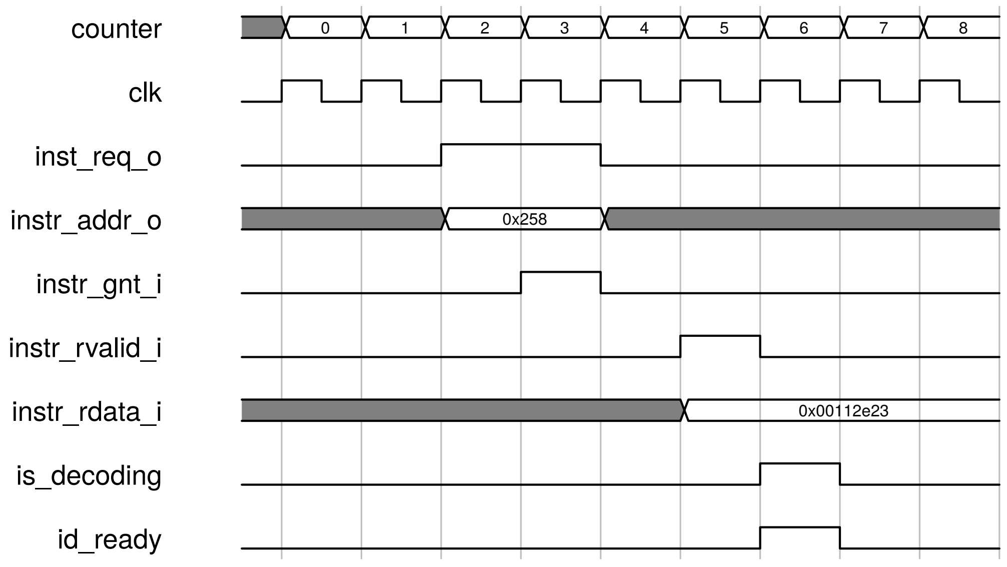


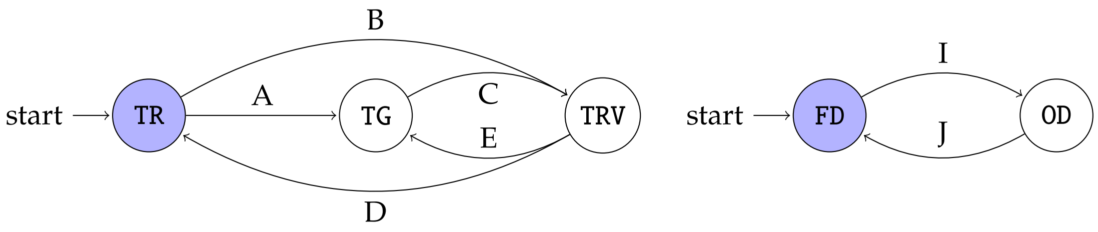
Figure A.6: Fetching the Instruction `0x00112e23` from Memory.

The process begins as `instr_req_o` goes high at Clock Cycle 2 and this will cause the Instruction Fetch State Machine to take transition A to the `TRACK_GNT` phase. In doing so the tracking buffer will have its entry cleared out. This is necessary because it is a circular buffer, so there could be data left over from a previous instruction. Then the value of the counter will be stored at the start of the `instr_req`. This can be seen in Figure A.8.

INST	ADDR	REQ_START	GNT	RVALID	DEC_END
X	X	X	X	X	X

BRANCH_DEC	BRANCH_DEC_TIME	OUTPUT_READY	FINISHED
X	X	0	0

(a) Schematic of a single record in the tracking buffer that will be filled out as this example progresses.



(b) A reduced version of the previously described state machines to allow us to follow the progress of the hardware as we progress through the example. The light blue highlight indicates the current state of the machine, with the Output State.

Figure A.7: The state of the `IF` tracker, we will return to updated versions of these diagrams as we progress through the example

INST	ADDR	REQ_START	GNT	RVALID	DEC_END
X	X	2	X	X	X

BRANCH_DEC	BRANCH_DEC_TIME	OUTPUT_READY	FINISHED
X	X	0	0

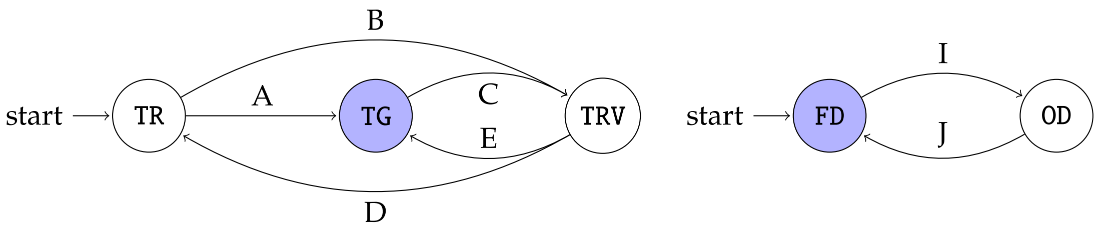


Figure A.8: In this phase of the example the start of an instruction transaction has been detected, so we progress to the `TRACK_GRANT` phase and record the start of the request in the `REQ_START` field of the tracking buffer.

Then the grant phase will continue until clock cycle 3, when `instr_gnt_i` goes high. This will cause the tracking buffer to be updated with the instructions address and the time at which the `instr_gnt_i` went high and cause transition C to be taken. Then at Clock Cycle 5, `TRACK_RVALID` will store the final pieces in the tracking buffer. The net result of these two steps can be seen in Figure A.9. Following this transition, D will be taken back to `TRACK_REQ` to track the next instruction.

INST	ADDR	REQ_START	GNT	RVALID	DEC_END
0x00112e23	0x258	2	3	5	X

BRANCH_DEC	BRANCH_DEC_TIME	OUTPUT_READY	FINISHED
X	X	0	0

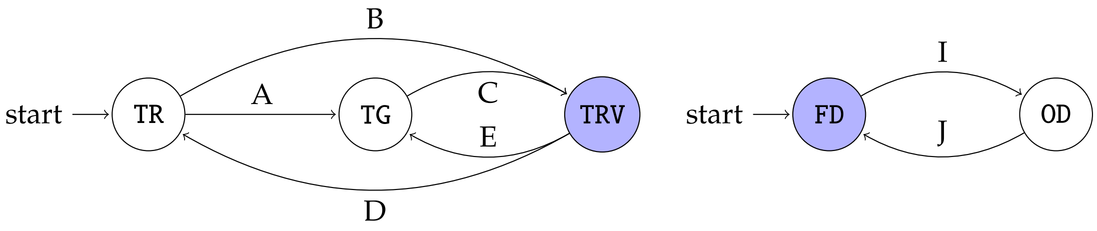


Figure A.9: In this phase, the grant signal is detected, which means we can populate the `ADDR` and `GNT` fields of the tracking buffer, as they are guaranteed at this point. Furthermore, when transition C is taken and we move into the `TRACK_RVALID` state we can also populate the `RVALID` and `INST` fields.

Once the decode phase of this instruction is complete, the processor signals this by setting the internal signals `is_decoding` and `id_ready` high in clock cycle 6. At this point the tracking buffer will be linearly scanned and the entry generated for `0x00112e23` will be marked as the earliest unprocessed index. Since we have not yet encountered a branch or jump instruction, we are not in a pending branching state. Consequently we store the end of the decode phase and then mark this instruction as ready for output.

In the next clock cycle, the Output State Machine will scan the tracking buffer until it finds data that is marked ready for output (it additionally checks some other conditions but we will omit these for now). It will find the data for `0x00112e23`, and run a check to ensure this is a `LOAD`/`STORE` instruction by looking at its opcode and so will take transition I to the `OUTPUT_DATA` state, where it will transfer the data into the next module. Following that, it will mark `0x00112E23` as finished and take transition J to the `FIND_DATA` state. This can be seen in Figure A.11.

INST	ADDR	REQ_START	GNT	RVALID	DEC_END
0x00112e23	0x258	2	3	5	6

BRANCH_DEC	BRANCH_DEC_TIME	OUTPUT_READY	FINISHED
X	X	1	0

Figure A.10: When the end of the decode phase is detected, we can mark the instruction as ready to output and track the time at which the decode phase is completed.

INST	ADDR	REQ_START	GNT	RVALID	DEC_END
0x00112e23	0x258	2	3	5	6

BRANCH_DEC	BRANCH_DEC_TIME	OUTPUT_READY	FINISHED
X	X	1	1

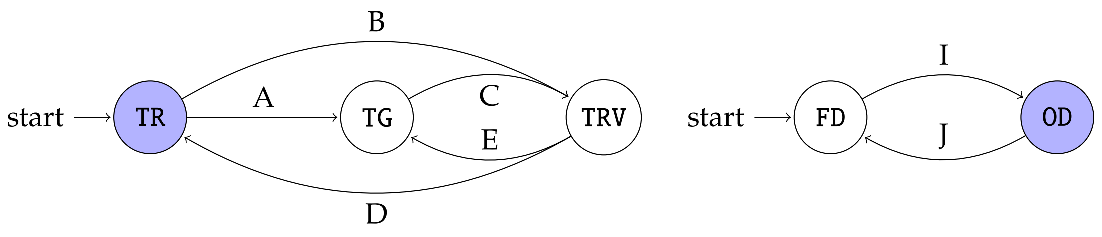


Figure A.11: Now we are in the output data phase, this record is marked as finished, so it will not be considered again when the tracking buffer is searched. We then move back to looking for another instruction transaction in the `TRACK_REQ` state.

A.3.2 Complex Branching Example

The previous example is relatively simple, but introducing branching causes several other factors to have to be taken into account. For this example we will use the signal diagram in Figure A.12. An important point to note about this diagram is that a `LOAD` from data memory is already in progress when this diagram starts, this prevents the decode phase for `0x2A4` starting as soon as it is fetched and sets this train of events in motion.

The dynamics of the Instruction Fetch State Machine are quite similar. However, the big change occurs when the decode phase ends for `0xFCE7DAE3`. At the start it proceeds very much as before. `0xFCE7DAE3` is identified by the module as the earliest unprocessed instruction and we are not currently branching, so we mark the end of the decode phase, but then we detect `0xFCE7DAE3` as a branching instruction. Therefore, we set the `branch_pointer` to mark this instruction as the object of the branching behaviour and then trigger the Branch Decision State Machine to take transition G. The net result of this can be seen in Figure A.13.

Now, because `0xFCE7DAE3` is a branching instruction that relies on a condition, it needs to enter the execution phase in order to calculate whether a branch will be taken or not. However, rather than simply stalling the processor when this happens, the next instruction sequentially is fetched from instruction memory, in this case `0xFE842783`. Due to the previously mentioned fetch from data memory, this fetch actually completes before a branch decision can be made, causing a new entry to enter the tracking buffer, as in Figure A.14

Let us assume that in this case the branch is in fact taken, so the next instruction to be executed will be `0xFEC42503`, at address `0x278`. In the `CHECK_BRANCH_DECISION` phase, the `branch_decision` signal goes high, so we then store the time at which that occurred and set the cut off time as 112. This cut off time is important, because it defines the time at which an instruction grant would need to take place after to be a valid fetch, forming a window with the lower bound set by the grant time of the branching instruction itself. A visual representation of this can be seen in Figure A.15.

We then also set a marker to state that we are in a branching state and mark the branching instruction as ready to output. The effect of these concurrent changes, both the fetch from `0x2A8` and the effect of the branch decision state machine can be seen in Figure A.16. The Branch Decision State Machine then returns to the `IDLE` state.

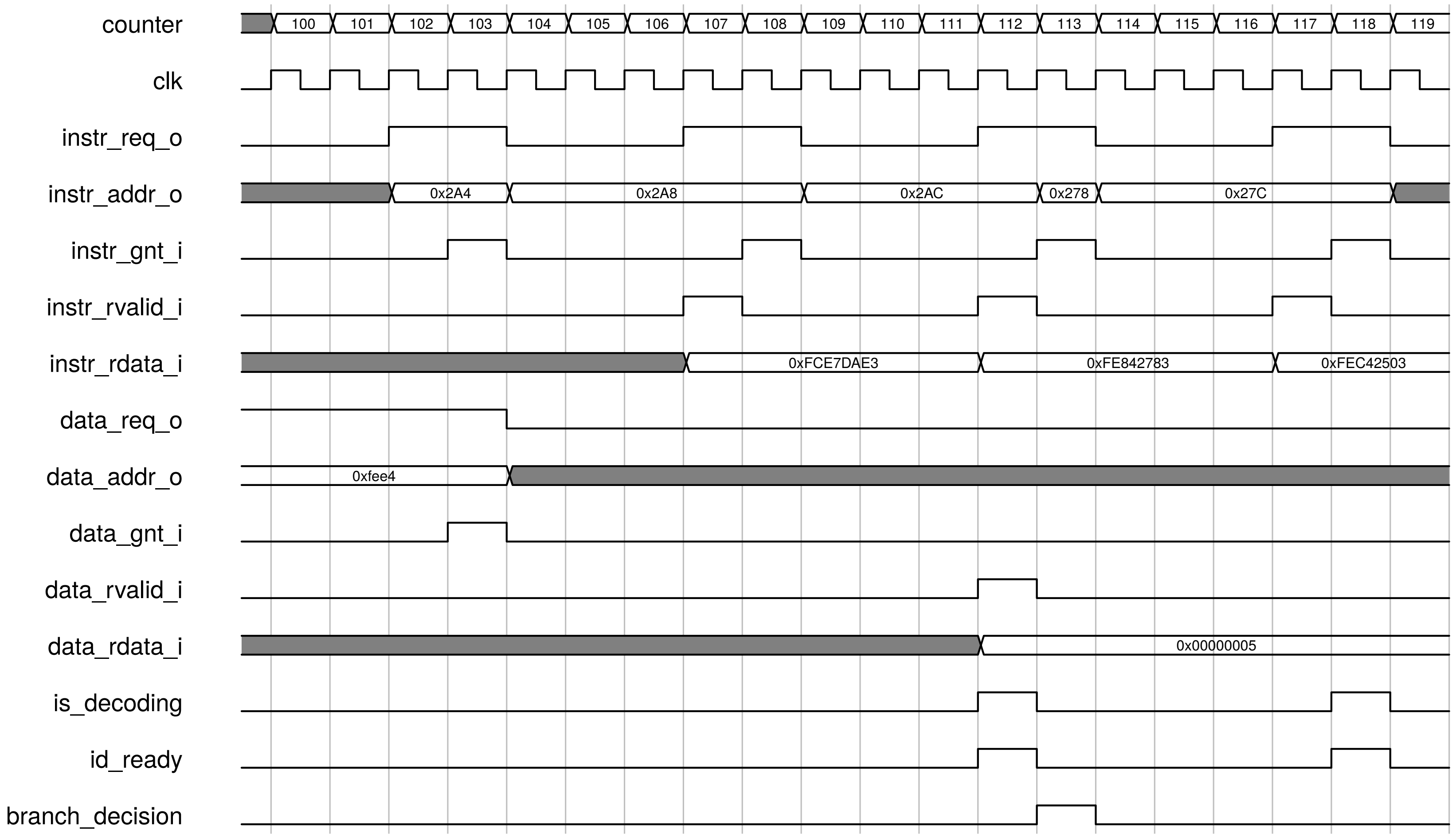


Figure A.12: In this signal diagram we join as a `LOAD` from data memory is already in progress, only finishing at cycle 112. This causes a large delay in between the successful fetch of the branching instruction and its eventual decoding and address calculation. The processor has built in mechanisms to stop any incorrect fetches that occur in between from propagating, so this is something we have to emulate in the tracker.

INST	ADDR	REQ_START	GNT	RVALID	DEC_END
0xFCE7DAE3	0x2A4	2	3	7	12

BRANCH_DEC	BRANCH_DEC_TIME	OUTPUT_READY	FINISHED
X	X	0	0

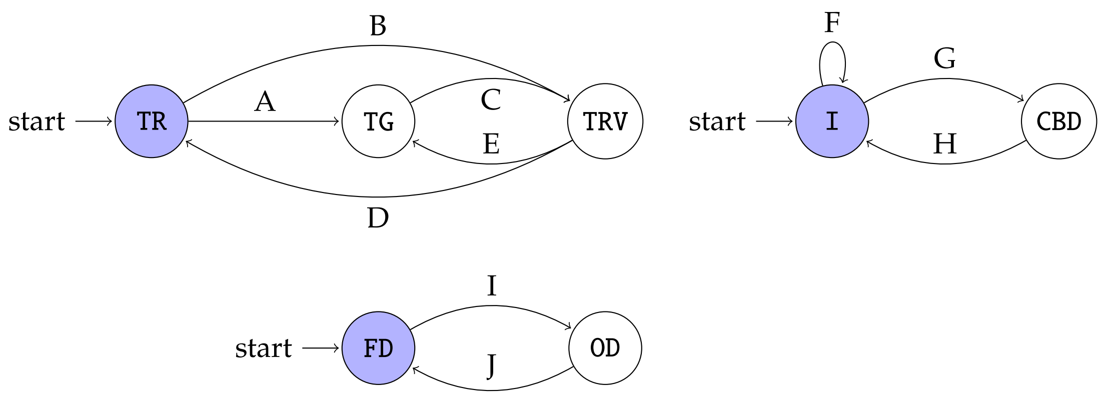


Figure A.13: At this point we have already fetched and decoded the instruction in question. However, because `0xFCE7DAE3` is a branching instruction we have to engage the Branch Decision State Machine to help us decide whether fetches made before the condition is known are valid or not.

INST	ADDR	REQ_START	GNT	RVALID	DEC_END
0xFCE7DAE3	0x2A4	102	103	107	112
0xFE842783	0x2A8	107	108	112	X

BRANCH_DEC	BRANCH_DEC_TIME	OUTPUT_READY	FINISHED
X	X	0	0
X	X	0	0

Figure A.14: As the decoding and address calculation are delayed, a second record enters the tracking buffer. This record may be an entirely legitimate fetch or it may be an oversight on the part of the processor. We need to know definitively in order to achieve a trace that has parity with what was executed.

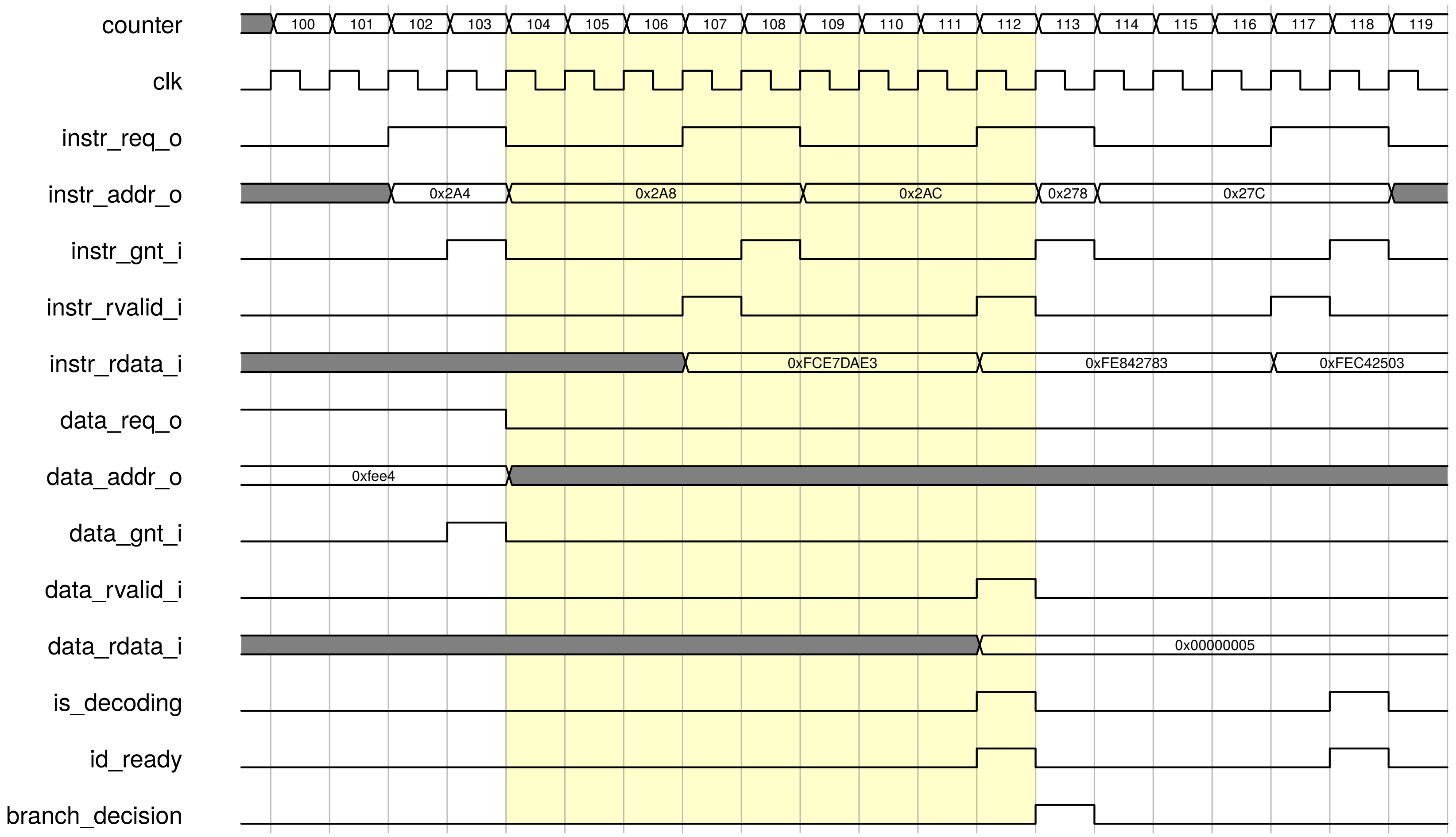


Figure A.15: The highlighted region indicates the region within which any successful fetches should be ignored. It ranges from the cycle after the grant of the branching instruction to the cycle in which the branch decision signal goes high. Now because the grant signal for `0x2A8` falls in this window, it must be disregarded.

INST	ADDR	REQ_START	GNT	RVALID	DEC_END
0xFCE7DAE3	0x2A4	102	103	107	112
0xFE842783	0x2A8	107	108	112	X

BRANCH_DEC	BRANCH_DEC_TIME	OUTPUT_READY	FINISHED
1	112	0	0
X	X	0	0

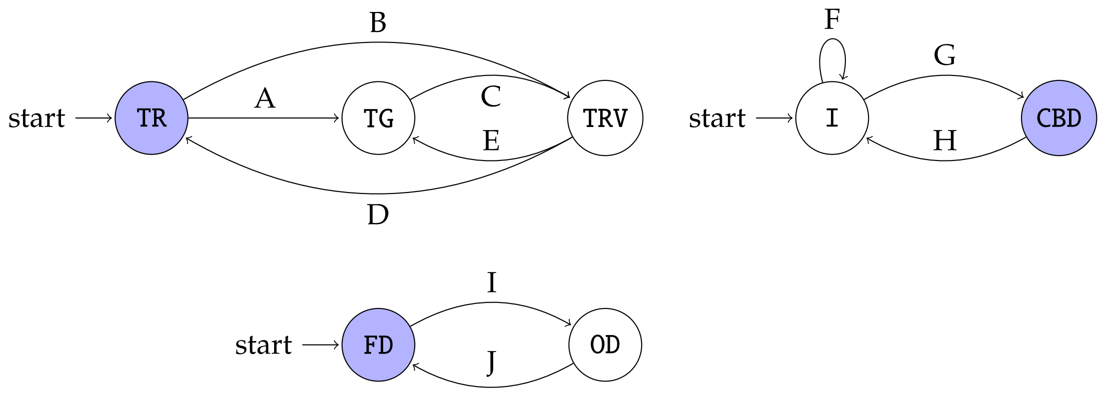


Figure A.16: At some point after `CHECK_BRANCH_DECISION` is entered, the `branch_decision` flag goes high to indicate the branch is to be taken. That means that because `0x2A8` follows `0x2A4` sequentially it must be disregarded and the next decode phase assigned to the next instruction after it to ensure consistency.

Now the Output State Machine will deal with both the accumulated instructions. First it will process the `0xFCE7DAE3` and this will simply be set to finished immediately as it is not a `LOAD` or `STORE` instruction. However, then it reaches the next instruction after clock cycle 108. At first it will a similar process to the simple example, it will be selected as the earliest unprocessed index but then, because, as can be seen in Figure A.15 its grant occurs before the cut off time, we were still in branching state when the fetch occurred so this fetch is invalid. Consequently we simply mark this instruction in the tracking buffer as finished, and then search again for another instruction to assign this decode phase to. The net result of this can be seen in the final contents of the tracking buffer, as shown in Figure A.17.

INST	ADDR	REQ_START	GNT	RVALID	DEC_END
`0xFCE7DAE3`	`0x2A4`	`102`	`103`	`107`	`112`
`0xFE842783`	`0x2A8`	`107`	`108`	`112`	`X`
`0xFEC42503`	`0x278`	`112`	`113`	`117`	`118`

BRANCH_DEC	BRANCH_DEC_TIME	OUTPUT_READY	FINISHED
`1`	`112`	`1`	`1`
`X`	`X`	`0`	`1`
`X`	`X`	`1`	`0`

Figure A.17: The final state of the tracking buffer: the branching instruction is marked as finished, even though it was not output due to it not being a memory operation; the incorrect fetch is marked as finished and not output as it was incorrectly tracked; and finally, the next correct fetch is ready for output the next time the `OUTPUT_DATA` state is returned to.

This mirrors what happens in the processor where instructions that are incorrectly predicted are invalidated and therefore never propagate throughout the processor. Once we reach an instruction that is beyond the calculated cut off point, we are non-longer in a branching state and so can start outputting data again in a similar fashion to before.

A.4 The `EX` Module

Once the process has completed in the `IF` module, the pertinent data is passed into the `EX` module, which tracks the data memory querying part of the process. Since we have already filtered out a lot of the instructions, the process in this phase is slightly simpler. However, since there are dependencies between instructions, this module cannot act synchronously with the processor. This means there are several signal buffers that record previously seen signal values and are then queried, so that the beginning and the end of the transactions can be seen.

A.4.1 The Main State Machine

This module consists of a single state machine that is constructed as per Figure A.18. The process begins in `EX_START`, where a trace buffer that is populated by the data coming in from the `IF` module signals that data is available to be processed. If this is the case, transition A is taken to the `GET_DATA` state. In that state, the data is extracted from the buffer and held in an internal trace buffer, so it can be built up before being eventually sent to the trace repository to be recalled later. Importantly, in this phase a query is also sent to a signal tracking module that asks if the start of a memory request event has been seen between the current time and the end of the decode phase of the data that has just come from the buffer.

The signal tracker is a relatively simplistic implementation of a buffer with surrounding logic to detect the start and end of transactions. It features a circular buffer that is a configurable size to track each of the signals of interest and then can look back over the signal values of the past to attempt to detect the timing of the values associated with particular memory transactions. Special care has to be taken to track these events properly and there is a feedback cycle between the `EX` module and the signal trackers to ensure that events associated with old events are not considered when new events are asked for.

After transition, C is taken to the `CHECK_MEM_REQ` state, the state machine will wait in the `CHECK_MEM_REQ` state until the signal tracker has reported back its discoveries. This introduces a single cycle of delay, for which the pertinent memory signals are also captured. Therefore, these can be added to the information reported back by the signal tracker. This information is reported in the form of a pair that indicates the start and end of the data memory request. Now there are four possibilities at this point and the module will take different actions accordingly, thus:

1. The requesting phase of the memory transaction occurred entirely in the past tracked by the signal tracker. If this is the case, then we record the start time of the transaction and then set up a parallel query with 2 signal trackers, one for the associated `rvalid` values and the other for the data address associated with the transaction. We know these must occur between the current time and the end of the requesting phase.

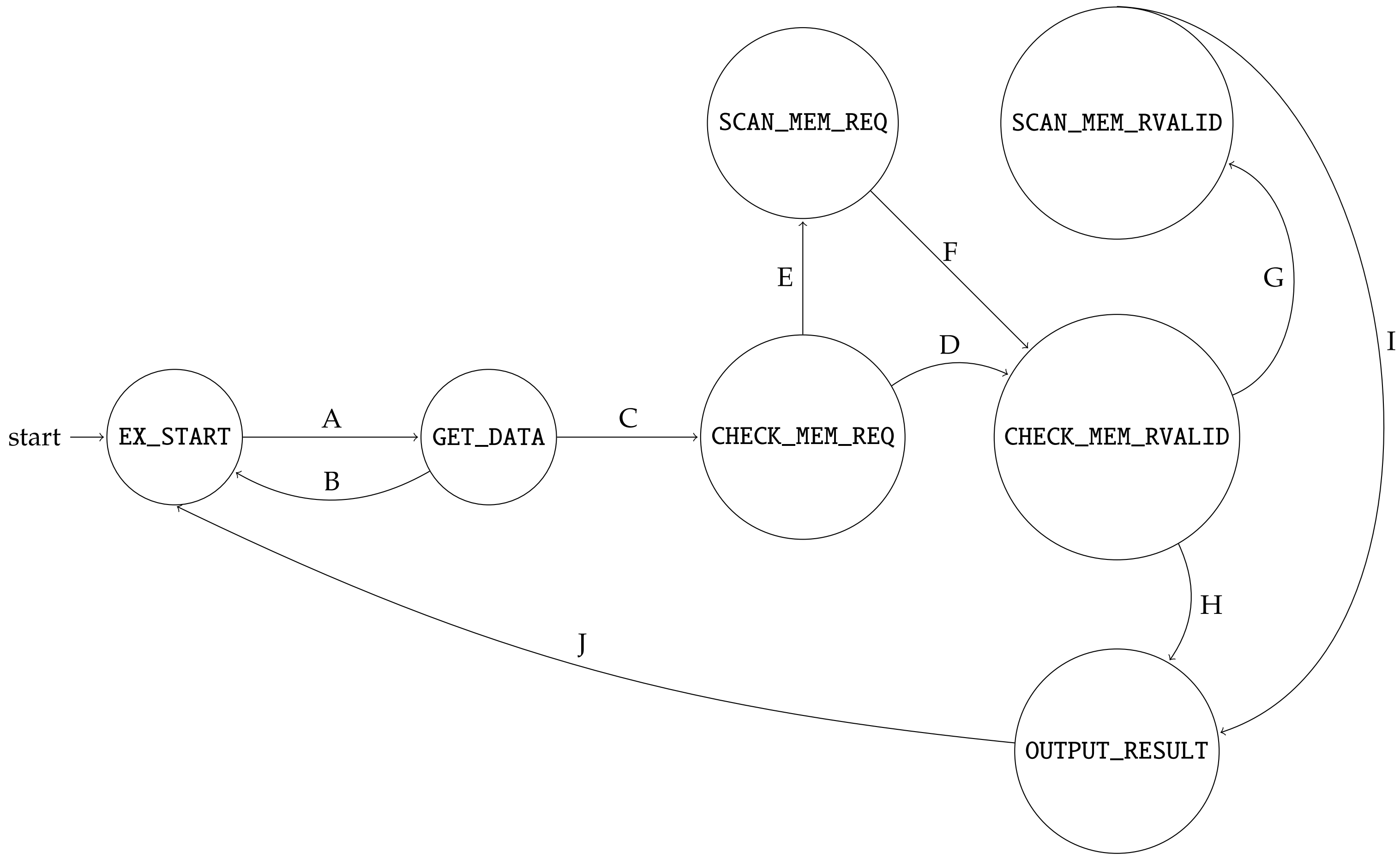


Figure A.18: The central state machine that controls the operation of the `EX` module. This works asynchronously with the processor, operating in a producer/consumer model with the `IF` module, which works synchronously with the processor.

2. The requesting phase started in the one cycle between asking the signal tracker and it reporting a result. In which case, we act as in Situation 1, but set a different window to look for `rvalid` and address values.
3. The requesting phase is starting in the current clock cycle, so again act as Situation 1, but with different parameters.
4. The memory request has not yet started.

If we are in Situations 1 - 3 we can take transition D and advance to `CHECK_MEM_RVALID`, but situation 4 requires us to take transition E and sit in state `SCAN_MEM_REQ`, which polls the `data_req_o` signal until it sees the signal go high. At that point transition F can be taken to `CHECK_MEM_RVALID`, with the new signal tracker appropriately primed.

Once in the `CHECK_MEM_RVALID` state, we have to wait for 2 separate processes to return a value. The first is the search for the address. As we already know the point at which the request started, this is a simple matter of looking back that many clock cycles. The search for the `rvalid` is similar, but has to scan over a window, as the search for the `req` signal did. Once the address is returned and the signal is valid, this is stored in the internal trace buffer and a flag is set. At the same time the signal buffer is searching for the time at which the `rvalid` signal occurred and it reports this as a pair in a very similar way to checking for the `req` signal as before. Once this pair has returned, it is checked to decide if we are in analogous situations to those above: either the signal was entirely in the past, it happened while the search was going on, it is happening right now or it has not happened yet.

Once it has been classified into one of those categories, this information is combined with the address information and stored in the trace buffer if necessary. Then a decision is made, either we need to scan for the `rvalid` signal because it has not happened yet, so transition G is taken and we sit polling the `rvalid` signal in `SCAN_MEM_RVALID` or we move to output the result to the trace repository in `OUTPUT_RESULT`. Once in `OUTPUT_RESULT` the now complete trace element is sent to the trace repository and transition J is taken back to the starting state.

A.4.2 Example

To demonstrate this process fully we will discuss the progression of `0x00112E23` through this state machine. First, we will assume that the memory transaction associated with this instruction happened as in Figure A.19. Let us assume we have now arrived at clock cycle 57, the point the data associated with `0x00112e23` is pulled out of the trace buffer.

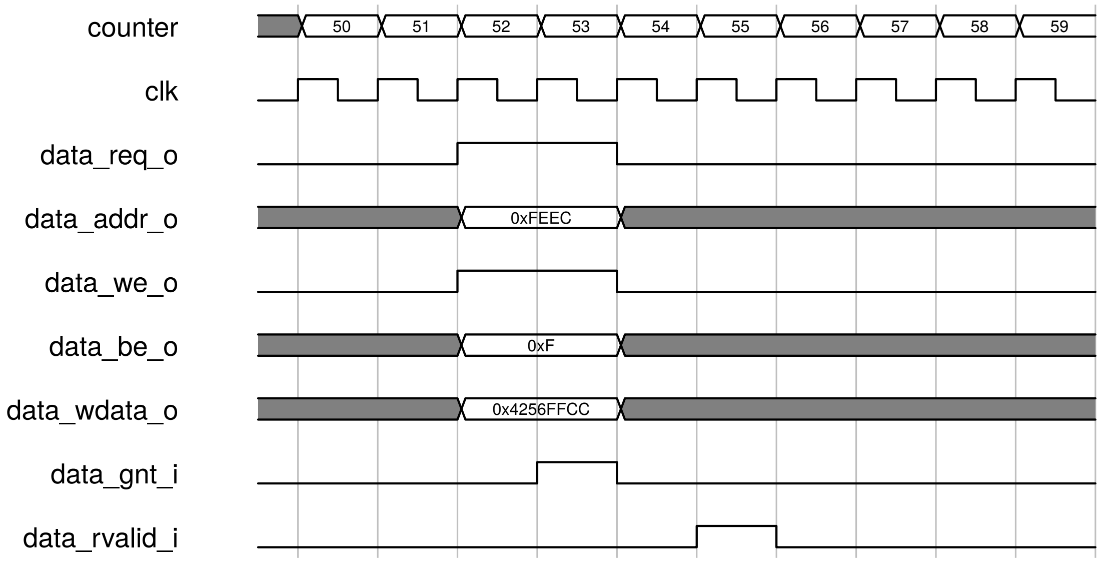


Figure A.19: A memory transaction to triggered by instruction `0x00112E23`, a `STORE` to `0xFEFC`.

We start in the `EX_START` phase and then the `data_present` flag goes high to indicate data is present. We then transition into the `GET_DATA` phase on the next clock cycle (58), where the data associated with `0x00112e23` is transferred into the internal trace buffer of the `EX` module. At the same time, we trigger the signal tracker to check for `data_req_o` signals in the last 7 clock cycles, as the decode phase for this instruction ended at 51 and we are currently in clock cycle 58. We then transition to the `CHECK_MEMORY_REQS` phase where we wait for the signal tracker to return. At the same time as this, we are tracking the state of the `data_req_o` signal, so we do not have a blindspot where the signal tracker has no records.

The signal tracker will then begin to scan the area highlighted in blue in Figure A.20. Once the scanning is complete, in clock cycle 59, it will return a pair (52,53) indicating the start and end of the transaction. We then store the start in the trace buffer and then trigger two more signal buffer searches for the address and the `rvalid` times. These will be asked to look 6 clock cycles back because the current time is now 59 clock cycles and the end of the `req` phase was at 53. This can further be seen as the green coloured section in Figure A.21.

We then move to the `CHECK_MEM_RVALID` phase where after 1 clock cycle the address signal tracker returns the value `0xFEEC` and this is stored in the trace buffer. In the next cycle the `rvalid` tracker returns a pair with the value 55 and 55, which means we have managed to find the `rvalid` in the window tracked by the signal tracker. We store this information in the signal buffer and then move to the `OUTPUT_RESULT` phase.

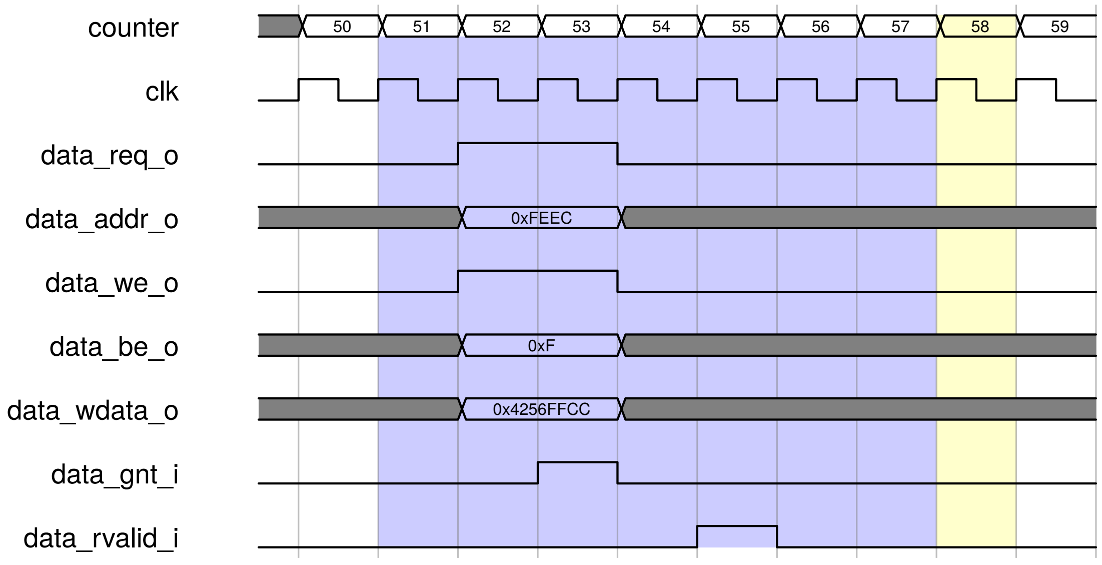


Figure A.20: The yellow highlighted region of the signal diagram indicates the current time at which the request to the signal tracker is made. The blue region then indicates the signals that the signal tracker has stored in its internal buffer and therefore can be queried. Due to not wanting to cause a race condition, the current clock cycle cannot be queried and so has to be tracked separately.

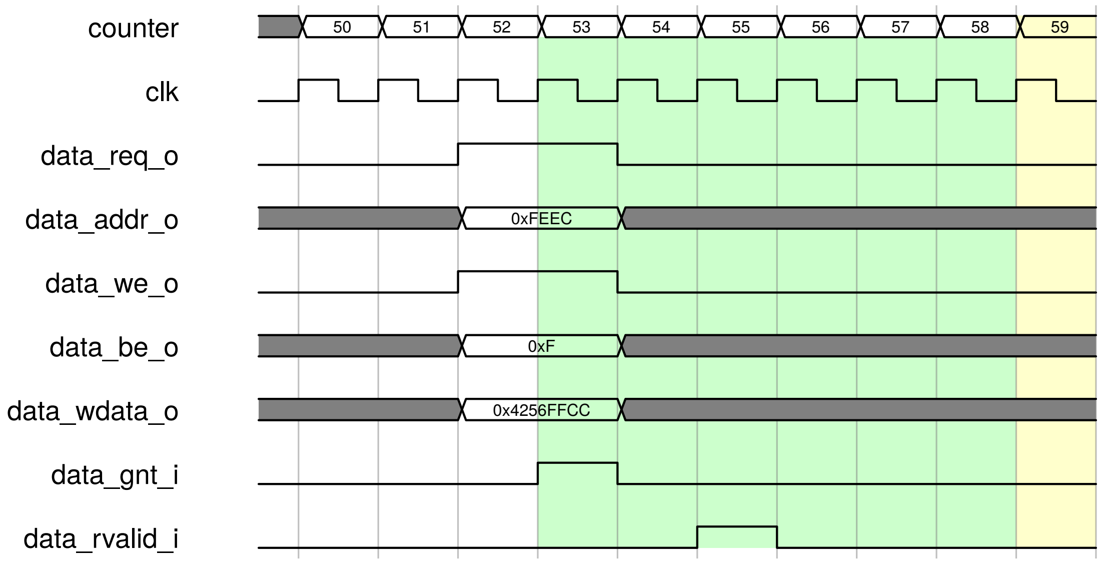


Figure A.21: The green highlighted region of the diagram indicates the area that will be queried by the signal tracker, to find the `addr` value and `rvalid` time. This is similar to the previous example. The time at which the request was made also has to be stored so it can be queried later, but this is dealt with outside the signal tracker.

Finally we have all the information required about this instruction's execution, so we output this data to be stored in the trace repository to be recalled later as required.

B Trace Recorder (Gouram) Implementation

This Appendix details the raw test results for each benchmark that were presented in Chapter 5. These can be seen on the next page.

Benchmark	Instruction Count	CPU Memory Requests	No Cache	Direct-Mapped Cache	Set-Associative Cache	Direct-Mapped TAC	Set-Associative TAC
bs	143	65	3762	1207	990	1308	1094
janne_complex	348	179	10306	1745	1745	2088	2087
fac	463	208	12133	7331	4588	7852	4637
fibcall	531	365	20494	2956	2655	4065	3751
insertsort	2292	816	47812	12782	11685	12943	11779
select-int	2202	871	50212	18087	12233	19036	12862
qsort-exam-int	2144	894	51629	20835	12187	22261	13525
duff	2047	1196	67781	19837	13987	22809	16635
cover	3672	1470	85619	31308	32202	32702	33056
cnt	7236	1544	108261	54511	52362	56281	53598
minver-int	4372	1715	99610	42206	29382	45012	32217
recursion	4070	1740	102930	54276	35317	58576	38221
expint	3459	2021	120046	28095	27341	29950	29270
jfdctint	4244	2204	124183	53410	42937	59837	49489
ud	6605	2483	147342	56702	44957	61355	49020
qurt	19487	2570	218446	138064	117011	143123	121626
fdct	5338	2705	152684	59301	38386	65694	43467
nsichneu	8126	2964	173511	78484	71527	83812	77005
prime	11771	5658	335896	75915	74626	81273	78412
ns	22359	6409	412216	162450	155743	167967	161456
ludcmp	44522	6692	531080	336652	281458	350079	290711
fft1	67964	8217	723781	434812	376260	448540	380409
bsort100	194422	71354	4206306	1664337	1384797	1742266	1434788
adpcm	259169	122360	7044636	2354271	1498442	2607582	1713484
matmult	433484	74648	5148631	3257762	2865108	3312116	2935222

Table B.1: Data for each benchmark indicating the number of instructions that each benchmark generates when compiled and the number of memory requests made by the CPU during the benchmark's run.

C Calculation of Trace-Assisted Cache Overheads

When running a more complex cache, it will naturally incur a higher overhead than would a simpler cache of the same size. Further to that, since this new cache can undertake preemptive actions beyond that of a standard cache, there are more situations that can arise. This appendix details the calculations of the approximate level of overhead incurred for each type of transaction the cache may undertake.

To aid us in this discussion we will first recall that the set of transactions that can be undertaken is the following:

- Cache Hit (No Preemptive Action) - *H*
- Cache Hit (Following a Preemptive Hit) - *HPH*
- Cache Hit (Following a Preemptive Miss) - *HPM*
- Cache Hit (Following a Preemptive Miss & Writeback) - *HPMW*
- Cache Miss (No Preemptive Action) - *M*
- Cache Miss (With Writeback) - *MW*

We will also recall that each of these transactions has a separate `LOAD/STORE` variant. To further aid our discussion we reproduce Figure 4.4 as Figure C.1. We will tackle each transaction one-by-one over the following appendix, showing how the overheads accumulate through each transaction, eventually culminating in the table shown in 6.3. It is worth pointing out before we begin this appendix that the figures we arrive at only detail the time spent serving a particular memory instruction. They do not, in the case of HPH, HPM or HPMW, account for the gap between the preemptive action and then the cache hit.

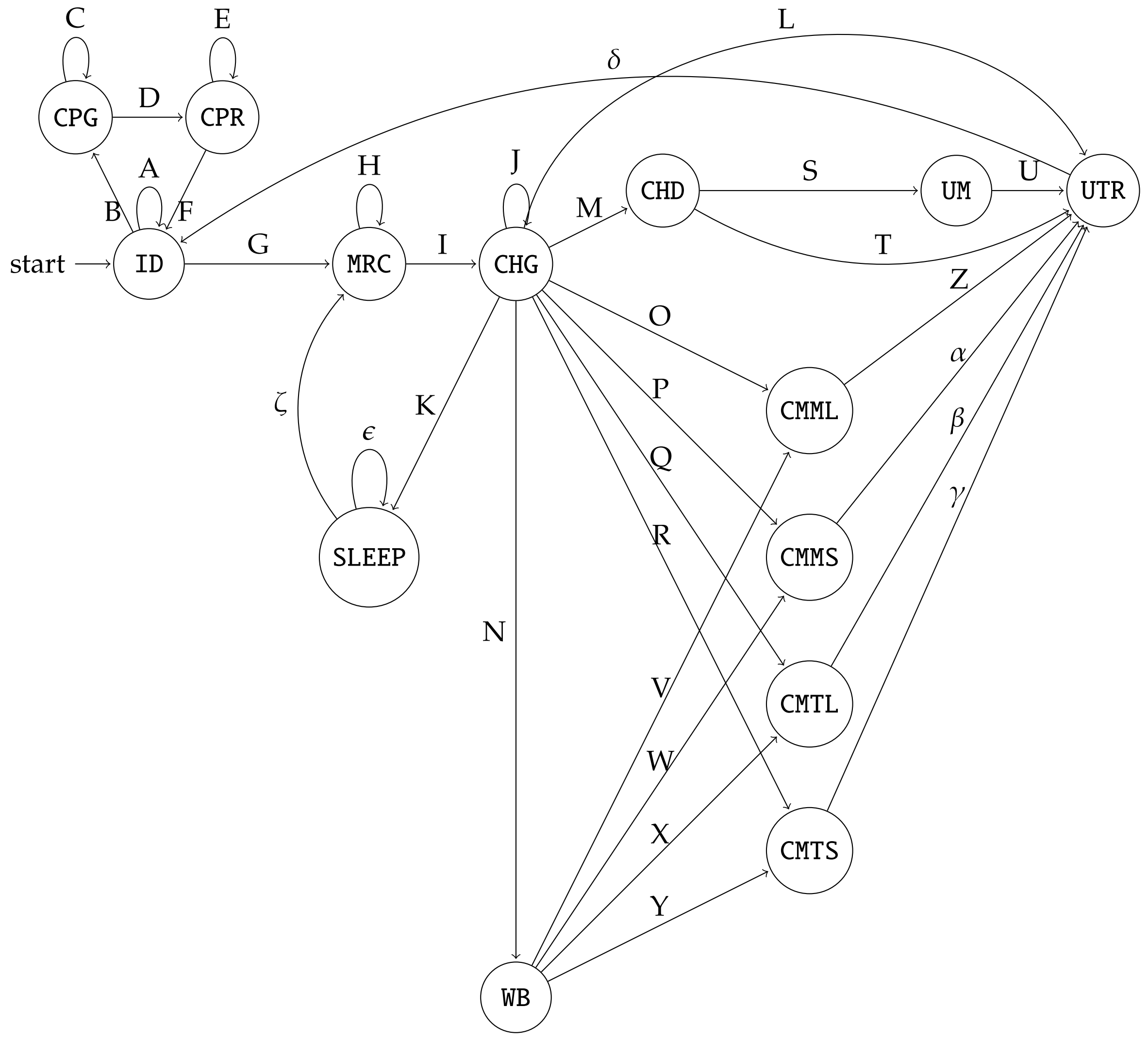


Figure C.1: State machine showing the operation of the TAC. This appendix will breakdown the operations that traverse the state machine to show how overheads accumulate throughout the system.

C.1 Cache Hit (No Preemptive Action)

To begin we will look at one of the simplest cases the cache displays, a hit that is instigated by the CPU and has no preemptive component. This begins in the `IDLE` (`ID`) state where 1 clock cycle is spent before taken transition `G` to the `MAKE_REQUEST_TO_CACHE` state, `MRC`. Now since the request is coming from the CPU, 1 cycle will be consumed to set up the signals to check the state of the cache, used in the next state to see if an early exit is required, and then another cycle is consumed to actually query the cache for the data required. Then transition `I` is taken to the `CACHE_HIT_GNT` (`CHG`) state.

Already at 3 cycles, transition `J` will be taken several times, until the cache has responded to the initial request. Throughout most simulations this was only once round the loop, consuming 1 cycle. After that, having discovered the data is in the cache, it will take transition `M` to the `CACHE_HIT_DATA` (`CHD`) state consuming another cycle. Therefore, we have consumed 5 so far. In this state various signals are set so as to inform the CPU that the data was in fact in the cache and then we transition, via `S`, to the `UPDATE_MAPPING` (`UM`) state. In this state 3 cycles are consumed whilst communication takes place with the trace repository to get back the correct cache index for this piece of data, allowing the TAC to update its internal representation of the cache, and then transition `U` is taken to the `UPDATE_TRACE_REPO` (`UTR`) state. Finally here 1 cycle is consumed before transition δ is taken back to the `IDLE` state.

If we add all this up, then we see the following: `ID` contributes 1 cycle, `MRC` contributes 2 cycles, `CHG` contributes 2 cycles, `CHD` contributes 1 cycle, `UM` adds 3 cycles and `UTR` adds 1 cycle. This leads to a Cache Hit, `LOAD` or `STORE`, taking 10 cycles from start to finish.

C.2 Cache Hit (Following a Preemptive Hit) - *HPH*

In the case that we have a cache hit, but it has been proceeded by a preemptive hit, then this is a 2-stage process, the second phase of which has been covered in Section C.1. The first part though will be a little different, and will differ dependent on if we are dealing with a `LOAD` or a `STORE`. The first part of the preemptive action will be the same as in Section C.1 so `ID` will be the starting state and then `MRC` will be entered. At this point, however, because this is in the preemptive portion of the execution we need to query the trace repository properly to retrieve the trace we want to action. In the worst case, i.e. there is no buffered trace ready to serve, this will take 4 cycles and there is an additional 2 cycles added to perform the same functions we saw previously in Section C.1.

At this point, if the instruction is a `LOAD` we only spend 1 cycle in `CHG` before transitioning to `UTR`, because there is no work to be done. This is a preemptive hit by definition, so the data is already present. Therefore, there is nothing to be done other than to update the trace repository. If the instruction is a `STORE` we need to update the state of the cache, so a further cycle is expended to do that, before transitioning to `UTR` where a further 1 cycle is consumed.

If we put this together, we see our preemptive hit takes, in the `LOAD` case $1+6+1+1 = 9$, and $1 + 6 + 2 + 1 = 10$ in the `STORE` case. This gets added to the 10 cycles from Section C.1. Therefore, we consume 19 cycles spread over two distinct periods if the memory instruction is a `LOAD`, and 20 cycles if the instruction is a `STORE`.

C.3 Cache Hit (Following a Preemptive Miss) - *HPM*

Now if we encounter a preemptive miss, more actions need to be taken in both the `LOAD` and `STORE` case. The first part of the process is very similar to that which we saw in Section C.2, but it diverges after the `CHG` state, by which point 9 cycles have been consumed. At this point the cache will detect that there has been a miss, so action needs to be taken to retrieve the data from memory. In this situation we know we will not have to writeback any data, so the process will proceed via transition `Q` or `R` to `SERVICE_CACHE_MISS_TRACE_LOAD` (`CMTL`) or `SERVICE_CACHE_MISS_TRACE_STORE` (`CMTS`) as appropriate. In the former case, this will then trigger a memory transaction to fetch the data from memory and this will consume a large number of clock cycles. To keep the figures generic and not tied down to a specific memory implementation we will use A_c, as we did in Chapter 6. After this transition, β will be taken, where 1 cycle will be consumed updating the trace repository, before returning to the `IDLE` state.

If the memory operation is a `STORE`, however, then `CMTS` will only consume 1 cycle, as all it does is to mark the spot in the cache as reserved and as it does not have to writeback, has no further work to do. This is then followed by the same 1 cycle to update the trace repository as in the `LOAD` state. So putting this together, a `LOAD` transaction, that preemptively misses will take $9 + A_c + 1 = A_c + 10$ clock cycles, whilst a `STORE` transaction will take $9 + 1 + 1 = 11$ clock cycles. We then need to add a further 10 cycles for the cache hit behaviour we have already recorded in Section C.1. This leads to $A_c + 20$ cycles for a `LOAD` and 21 cycles for a `STORE`.

C.4 Cache Hit (Following a Preemptive Miss & Writeback) - *HPMW*

In the previous section we considered a Cache Hit preceded by a preemptive miss, but what if the cache block is not empty when the preemptive miss occurs? In this case we need to traverse an extra state to write the data back to the cache, before the miss is resolved. This behaviour is exactly the same whether the memory operation is a `LOAD` or a `STORE`. Practically progress around the state machine is exactly the same as in Section C.3, but instead of taking transition `O` or `P` the cache signals that a writeback is required and so takes transition `N` instead. While in state `N` it performs a `STORE` operation to writeback the data already present in the cache. This takes A_c clock cycles, as it is a whole memory transaction, and then control progresses along transition `V` or `W` as appropriate. The rest of the operation is as per Section C.3.

As this is additive on top of the process described in Section C.3, we simply need to add A_c to each of the figures we discovered previously. That tells us that a `LOAD` operation that finds itself in this situation will take $2A_c + 20$ cycles to complete and a `STORE` operation $A_c + 21$ cycles to complete.

C.5 Cache Miss (No Preemptive Action) - *M*

Turning now to cache misses, these are handled very similarly to preemptive misses and take the same amount of clock cycles, but different states are traversed. Rather than transitions `Q` and `R` being taken, transitions `O` and `P` are taken instead, leading to the state `SERVICE_CACHE_MISS_MEM_LOAD` (`CMML`) state or the `SERVICE_CACHE_MISS_MEM_STORE` (`CMMS`) state, respectively. When in these states `CMML` takes A_c cycles, as we have to fetch the required data from memory and `CMMS` takes only 1 cycle, as the data only needs to be written into the cache.

Because there is no preemptive component here, the cache is simply reacting to the demands of the CPU, as per Section C.3, this situation consumes $A_c + 10$ clock cycles in the case of a `LOAD` and 11 clock cycles in the case of a `STORE`.

C.6 Cache Miss (With Writeback) - *MW*

Finally we consider a cache miss with writeback. Again this proceeds very similarly to its preemptive cousins, everything will occur as in Section C.5, but instead of transitions `O` and `P`, transition `N` is taken to the `WRITEBACK` state (`WB`). Here A_c cycles will be consumed regardless of the type of memory transaction and then transitions `W` or `V` will be taken and the process will continue as per Section C.5. This adds an extra A_c clock cycles to each measure of time, which leads to a `LOAD` transaction taking $2A_c + 10$ clock cycles and a `STORE` taking $A_c + 11$ clock cycles.

C.7 Summary & Final Table

Now we have seen how each of the measures of time is built up, we can present the final table.

	`LOAD`	`STORE`
H	10	10
HPH	19	20
HPM	$A_c + 20$	21
HPMW	$2A_c + 20$	$A_c + 21$
M	$A_c + 10$	11
MW	$2A_c + 10$	$A_c + 11$

Table C.1: This table shows how long it would take each different type of memory transaction to progress through the system. This is the same table presented in Chapter 6.

Acronyms

AIP Access Interval Predictor.

ARC Adapative Replacement Cache.

BIP Bimodal Insertion Policy.

BRAM Block Random Access Memory.

CISC Complex Instruction Set Computer.

CMP Chip Multi-Processor.

COTS Commercial Off-The-Shelf.

CPI Clock Cycles Per Instruction.

CPU Central Processing Unit.

DIG Degree of Inter-reference Gap.

DIP Dynamic Insertion Policy.

DMA Direct Memory Access.

DRAM Dynamic Random Access Memory.

DSP Digital Signal Processor.

EELRU Early Eviction Least Recently Used.

FF Flip-Flop.

FIFO First-In-First-Out.

FPGA Field Programmable Gate Array.

HLRU History Least Recently Used.

HPC High Performance Computing.

ILA Integrated Logic Analyser.

IPC Instruction-per-Cycle.

IRR Inter-Reference Recency.

ISA Instruction Set Architecture.

ISS Instruction Set Simulator.

LACS Locality-Aware Cost-Sensitive.

LFU Least Frequently Used.

LFUDA Least Frequently Used with Dynamic Aging.

LFU-K Least Frequently Used (with *k* previous references).

LIRS Low Inter-reference Recency Set.

LLC Last Level Cache.

LRRe Least Recently Referenced.

LRU Least Recently Used.

LRU-K Least Recently Used (with *k* previous references).

LRU-WAR Least Recently Used with Working Area Restriction.

LSU Load-Store Unit.

LUT LookUp Table.

LvP Live Time Predictor.

MFU Most Frequently Used.

MPKI Misses per Thousand Instructions.

MPSoC Multi-Processor System-on-Chip.

MRRe Most Recently Referenced.

MRU Most Recently Used.

MSHR Miss Information/Status Holding Register.

NoC Network-on-Chip.

NRU Not Recently Used.

OBL	One Block Lookahead.
OoO	Out-of-Order.
ORL	Online Reference Locality.
OS	Operating System.
PC	Program Counter.
PRL	Profile Reference Locality.
RISC	Reduced Instruction Set Computer.
RRIP	Re-reference Interval Prediction.
SDRAM	Synchronous Dynamic Random Access Memory.
SFIFO	Segemented First-In-First-Out.
SMACK	System Metric for Applciation Cache Knowledge.
SoC	System-on-Chip.
SPAID	Speculatively Prefetching Anticipated Interprocedural Dereferences.
SRAM	Static Random Access Memory.
TAC	Trace-Assisted Cache.
VIP	AXI Verification IP.
WCET	Worst-Case Execution Time.
WFL	Wood, Fernandez and Long.
XPM	Xilinx Parameterized Macro.

Bibliography

[1] Adwan AbdelFattah and Aiman Abu Samra. "Least Recently Plus Five Least Frequently Replacement Policy (LR+5LF)". en. In: *The International Arab Journal of Information Technology* 9.1 (2012), p. 6.

[2] A. Agarwal and S.D. Pudar. "Column-Associative Caches: A Technique For Reducing The Miss Rate Of Direct-Mapped Caches". In: *Proceedings of the 20th Annual International Symposium on Computer Architecture*. May 1993, pp. 179–190. DOI: 10.1109/ISCA.1993.698559.

[3] A. Agarwal, K. Roy and T.N. Vijaykumar. "Exploring High Bandwidth Pipelined Cache Architecture for Scaled Technology". In: *Automation and Test in Europe Conference and Exhibition 2003 Design*. Mar. 2003, pp. 778–783. DOI: 10.1109/DATE.2003.1253701.

[4] J. Aguilar and E. L. Leiss. "A Coherence-Replacement Protocol For Web Proxy Cache Systems". In: *International Journal of Computers and Applications* 28.1 (Jan. 2006), pp. 12–18. ISSN: 1206-212X. DOI: 10.1080/1206212X.2006.11441783.

[5] Jose Aguilar and Ernst Leiss. "A General Adaptive Cache Coherency-Replacement Protocol for Web Proxy Cache Systems". en. In: *Computación y Sistemas* 8.1 (2004), pp. 1–14.

[6] T. Alexander and G. Kedem. "Distributed Prefetch-Buffer/Cache Design for High Performance Memory Systems". In: *Proceedings. Second International Symposium on High-Performance Computer Architecture*. Feb. 1996, pp. 254–263. DOI: 10.1109/HPCA.1996.501191.

[7] J. Alghazo, A. Akaaboune and N. Botros. "SF-LRU Cache Replacement Algorithm". In: *Records of the 2004 International Workshop on Memory Technology, Design and Testing, 2004.* Aug. 2004, pp. 19–24. DOI: 10.1109/MTDT.2004.1327979.

[8] Erik R Altman, Vinod K Agarwal and Guang R Gao. "A Novel Methodology Using Genetic Algorithms for the Design of Caches and Cache Replacement Policy". en. In: *Proceedings of the 5th International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, 1993, pp. 392–399. ISBN: 1-55860-299-2.

[9] R.E. Aly, B.R. Nallamilli and M.A. Bayoumi. "Variable-Way Set Associative Cache Design for Embedded System Applications". In: *2003 46th Midwest Symposium on Circuits and Systems*. Vol. 3. Dec. 2003, 1435–1438 Vol. 3. DOI: 10.1109/MWSCAS.2003.1562565.

[10] *AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite*. en. Feb. 2013.

[11] K. M. AnandKumar et al. "A Hybrid Cache Replacement Policy for Heterogeneous Multi-Cores". In: *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. Sept. 2014, pp. 594–599. DOI: 10.1109/ICACCI.2014.6968209.

[12] Andreas Traber, Michael Gautschi and Pasquale Schiavone Davide. *RI5CY: User Manual*. Nov. 2017.

[13] Ismail Ari et al. "ACME: Adaptive Caching Using Multiple Experts". In: *Distributed Data & Structures 4, Records of the 4th International Meeting (WDAS 2002), Paris, France, March 20-23, 2002*. Ed. by Witold Litwin and Gérard Lévy. Vol. 14. Proceedings in Informatics. Carleton Scientific, 2002, pp. 143–158.

[14] Martin Arlitt et al. "Evaluating Content Management Techniques for Web Proxy Caches". en. In: *ACM SIGMETRICS Performance Evaluation Review* 27.4 (Mar. 2000), pp. 3–11. ISSN: 01635999. DOI: 10.1145/346000.346003.

[15] H. Arora, S. Banerjee and V. Davina. "A Composite Data Prefetcher Framework for Multilevel Caches". In: *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. Sept. 2014, pp. 1827–1833. DOI: 10.1109/ICACCI.2014.6968442.

[16] Abu Asaduzzaman, Mark P. Allen and Tania Jareen. "An Effective Locking-Free Caching Technique for Power-Aware Multicore Computing Systems". In: *2014 International Conference on Informatics, Electronics Vision (ICIEV)*. May 2014, pp. 1–6. DOI: 10.1109/ICIEV.2014.6850861.

[17] M. Azimi, B. Prasad and K. Bhat. "Two Level Cache Architectures". In: *Digest of Papers COMPCON Spring 1992*. Feb. 1992, pp. 344–349. DOI: 10.1109/CMPCON.1992.186736.

[18] Jean-Loup Baer and Tien-Fu Chen. "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty". In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 176–186. ISBN: 978-0-89791-459-8. DOI: 10.1145/125826.125932.

[19] Sorav Bansal and Dharmendra S. Modha. “CAR: Clock with Adaptive Replacement”. In: *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*. FAST ’04. Berkeley, CA, USA: USENIX Association, 2004, pp. 187–200.

[20] B. Batson and T.N. Vijaykumar. “Reactive-Associative Caches”. In: *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. Sept. 2001, pp. 49–60. DOI: 10.1109/PACT.2001.953287.

[21] L. A. Belady. “A Study of Replacement Algorithms for a Virtual-Storage Computer”. In: *IBM Systems Journal* 5.2 (1966), pp. 78–101. ISSN: 0018-8670. DOI: 10.1147/sj.52.0078.

[22] L. A. Belady, R. A. Nelson and G. S. Shedler. “An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine”. In: *Communications of the ACM* 12.6 (June 1969), pp. 349–353. ISSN: 00010782. DOI: 10.1145/363011.363155.

[23] Samson Belayneh and David R. Kaeli. “A Discussion on Non-Blocking/Lockup-Free Caches”. en. In: *ACM SIGARCH Computer Architecture News* 24.3 (June 1996), pp. 18–25. ISSN: 01635964. DOI: 10.1145/381718.381727.

[24] Christian Bienia et al. “The PARSEC Benchmark Suite: Characterization and Architectural Implications”. en. In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques - PACT ’08*. Toronto, Ontario, Canada: ACM Press, 2008, p. 72. ISBN: 978-1-60558-282-5. DOI: 10.1145/1454115.1454128.

[25] Nathan Binkert et al. “The Gem5 Simulator”. In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: https://doi.org/10.1145/2024716.2024718.

[26] F. Bodin and A. Seznec. “Skewed Associativity Improves Program Performance and Enhances Predictability”. In: *IEEE Transactions on Computers* 46.5 (May 1997), pp. 530–544. ISSN: 2326-3814. DOI: 10.1109/12.589219.

[27] B. Calder, D. Grunwald and J. Emer. “Predictive Sequential Associative Cache”. In: *Proceedings. Second International Symposium on High-Performance Computer Architecture*. Feb. 1996, pp. 244–253. DOI: 10.1109/HPCA.1996.501190.

[28] David Callahan, Ken Kennedy and Allan Porterfield. “Software Prefetching”. en. In: ACM Press, 1991, pp. 40–52. ISBN: 978-0-89791-380-5. DOI: 10.1145/106972.106979.

[29] Chung-yi Chang, Anthony James McGregor and Geoffrey Holmes. *The LRU*WWW Proxy Cache Document Replacement Algorithm*. en. Working Paper. 1999.

[30] H. Chang, C. Chiang and Y. Yu. “An Adaptive Buffer Cache Management Scheme”. In: *2016 International Computer Symposium (ICS)*. Dec. 2016, pp. 124–127. DOI: 10.1109/ICS.2016.0033.

[31] Hsung-Pin Chang and Cheng-Pang Chiang. “PARC: A Novel OS Cache Manager”. en. In: *Software: Practice and Experience* 48.12 (2018), pp. 2193–2222. ISSN: 1097-024X. DOI: 10.1002/spe.2633.

[32] Mainak Chaudhuri. “Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-Level Caches”. en. In: ACM Press, 2009, p. 401. ISBN: 978-1-60558-798-1. DOI: 10.1145/1669112.1669164.

[33] Tien-Fu Chen and Jean-Loup Baer. “Effective Hardware-Based Data Prefetching for High-Performance Processors”. In: *IEEE Transactions on Computers* 44.5 (May 1995), pp. 609–623. ISSN: 2326-3814. DOI: 10.1109/12.381947.

[34] Tien-Fu Chen and Jean-Loup Baer. “Reducing Memory Latency via Non-Blocking and Prefetching Caches”. In: *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS V. Boston, Massachusetts, USA: Association for Computing Machinery, Sept. 1992, pp. 51–61. ISBN: 978-0-89791-534-2. DOI: 10.1145/143365.143486.

[35] Z. Chen et al. “SSARC: The Short-Sighted Adaptive Replacement Cache”. In: *2009 11th IEEE International Conference on High Performance Computing and Communications*. June 2009, pp. 551–556. DOI: 10.1109/HPCC.2009.82.

[36] Hyunseung Choo, Young Jae Lee and Seong-Moo Yoo. “DIG: Degree of Inter-Reference Gap for a Dynamic Buffer Cache Management”. In: *Information Sciences* 176.8 (Apr. 2006), pp. 1032–1044. ISSN: 0020-0255. DOI: 10.1016/j.ins.2005.01.018.

[37] J.D. Collins et al. “Dynamic Speculative Precomputation”. In: *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*. Dec. 2001, pp. 306–317. DOI: 10.1109/MICRO.2001.991128.

[38] J. Collins et al. “Pointer Cache Assisted Prefetching”. en. In: IEEE Comput. Soc, 2002, pp. 62–73. ISBN: 978-0-7695-1859-6. DOI: 10.1109/MICRO.2002.1176239.

[39] Robert Cooksey, Stephan Jourdan and Dirk Grunwald. “A Stateless, Content-Directed Data Prefetching Mechanism”. In: *ACM SIGOPS Operating Systems Review* 36.5 (Oct. 2002), pp. 279–290. ISSN: 0163-5980. DOI: 10.1145/635508.605427.

[40] Fernando J. Corbató. “A Paging Experiment with the Multics System”. In: *In Honor of Philip M. Morse*. Ed. by Herman Feshbach, K. Uno Ingard and Philip M. Morse. Cambridge: M.I.T. Press, 1969, pp. 217–228. ISBN: 978-0-262-06028-8.

[41] *CoreSight Base System Architecture*. en. July 18.

[42] Jike Cui and Mansur. H. Samadzadeh. “A New Hybrid Approach to Exploit Localities: LRFU with Adaptive Prefetching”. In: *SIGMETRICS Perform. Eval. Rev.* 31.3 (Dec. 2003), pp. 37–43. ISSN: 0163-5999. DOI: 10.1145/974036.974041.

[43] Fredrik Dahlgren, Michel Dubois and Per Stenstrom. “Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors”. In: *1993 International Conference on Parallel Processing - ICPP’93*. Vol. 1. Aug. 1993, pp. 56–63. DOI: 10.1109/ICPP.1993.92.

[44] Gille Damien. “Study of Different Cache Line Replacement Algorithms in Embedded Systems”. Masters Thesis. Stockholm: KTH Royal Institute of Technology, 2007.

[45] S. Das and A. Banerjee. “An Arbitration on Cache Replacements Based on Frequency — Recency Product Values”. In: *2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA)*. Jan. 2016, pp. 1–6. DOI: 10.1109/VLSI-SATA.2016.7593031.

[46] Shirshendu Das and Hemangee K. Kapoor. “Dynamic Associativity Management Using Fellow Sets”. In: *2013 International Symposium on Electronic System Design*. Dec. 2013, pp. 133–137. DOI: 10.1109/ISED.2013.33.

[47] Shirshendu Das and Hemangee K. Kapoor. “Latency Aware Block Replacement for L1 Caches in Chip Multiprocessor”. In: *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. July 2017, pp. 182–187. DOI: 10.1109/ISVLSI.2017.40.

[48] Shirshendu Das and Hemangee K. Kapoor. “Victim Retention for Reducing Cache Misses in Tiled Chip Multiprocessors”. en. In: *Microprocessors and Microsystems* 38.4 (June 2014), pp. 263–275. ISSN: 01419331. DOI: 10.1016/j.micpro.2013.11.005.

[49] Shirshendu Das et al. “Random-LRU: A Replacement Policy for Chip Multiprocessors”. en. In: *VLSI Design and Test*. Ed. by Manoj Singh Gaur et al. Communications in Computer and Information Science. Berlin, Heidelberg: Springer, 2013, pp. 204–213. ISBN: 978-3-642-42024-5. DOI: 10.1007/978-3-642-42024-5_25.

[50] N. Decker et al. “Online Analysis of Debug Trace Data for Embedded Systems”. In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2018, pp. 851–856. DOI: 10.23919/DATE.2018.8342124.

[51] Bobbala Lakshmi Deepika and Byeong Kil Lee. “Hybrid-Way Cache for Mobile Processors”. In: *2011 Eighth International Conference on Information Technology: New Generations*. Apr. 2011, pp. 707–712. DOI: 10.1109/ITNG.2011.125.

[52] L. Delshadtehrani et al. "Nile: A Programmable Monitoring Coprocessor". In: *IEEE Computer Architecture Letters* 17.1 (Jan. 2018), pp. 92–95. ISSN: 1556-6056. DOI: `10.1109/LCA.2017.2784416`.

[53] Peter J. Denning. "The Working Set Model for Program Behavior". In: *Communications of the ACM* 11.5 (May 1968), pp. 323–333. ISSN: 00010782. DOI: `10.1145/363095.363141`.

[54] Peter J. Denning. "Thrashing: Its Causes and Prevention". en. In: ACM Press, 1968, p. 915. DOI: `10.1145/1476589.1476705`.

[55] Yannick Deville. "A Low-Cost Usage-Based Replacement Algorithm for Cache Memories". en. In: *ACM SIGARCH Computer Architecture News* 18.4 (Dec. 1990), pp. 52–58. ISSN: 01635964. DOI: `10.1145/121973.121979`.

[56] Chen Ding and K. Kennedy. "The Memory of Bandwidth Bottleneck and Its Amelioration by a Compiler". In: *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*. May 2000, pp. 181–189. DOI: `10.1109/IPDPS.2000.845980`.

[57] A. Djordjalian. "Minimally-Skewed-Associative Caches". In: *14th Symposium on Computer Architecture and High Performance Computing, 2002. Proceedings.* Oct. 2002, pp. 100–107. DOI: `10.1109/CAHPC.2002.1180765`.

[58] Donghee Lee et al. "Implementation and Performance Evaluation of the LRFU Replacement Policy". In: *Proceedings 23rd Euromicro Conference New Frontiers of Information Technology - Short Contributions -*. Sept. 1997, pp. 106–111. DOI: `10.1109/EMSCNT.1997.658446`.

[59] Donghee Lee et al. "LRFU: A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies". In: *IEEE Transactions on Computers* 50.12 (Dec. 2001), pp. 1352–1361. DOI: `10.1109/TC.2001.970573`.

[60] Tom Duff. *Re: Explanation, Please!* English. Aug. 1988.

[61] Nam Duong et al. "SCORE: A Score-Based Memory Cache Replacement Policy". en. In: (June 2010).

[62] Haakon Dybdahl, Per Stenström and Lasse Natvig. "An LRU-Based Replacement Algorithm Augmented with Frequency of Access in Shared Chip-Multiprocessor Caches". In: *Proceedings of the 2006 Workshop on MEmory Performance: DEaling with Applications, Systems and Architectures*. MEDEA '06. New York, NY, USA: ACM, 2006, pp. 45–52. ISBN: 978-1-59593-568-7. DOI: `10.1145/1166133.1166139`.

[63] R. Fares et al. “Performance Evaluation of Traditional Caching Policies on a Large System with Petabytes of Data”. In: *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*. June 2012, pp. 227–234. DOI: 10.1109/NAS.2012.32.

[64] K.I. Farkas, N.P. Jouppi and P. Chow. “How Useful Are Non-Blocking Loads, Stream Buffers and Speculative Execution in Multiple Issue Processors?” In: *Proceedings of 1995 1st IEEE Symposium on High Performance Computer Architecture*. Jan. 1995, pp. 78–89. DOI: 10.1109/HPCA.1995.386553.

[65] John W. C. Fu and Janak H. Patel. “Data Prefetching in Multiprocessor Vector Cache Memories”. In: *Proceedings of the 18th Annual International Symposium on Computer Architecture*. ISCA ’91. Toronto, Ontario, Canada: Association for Computing Machinery, Apr. 1991, pp. 54–63. ISBN: 978-0-89791-394-2. DOI: 10.1145/115952.115959.

[66] John W. C. Fu, Janak H. Patel and Bob L. Janssens. “Stride Directed Prefetching in Scalar Processors”. In: *ACM SIGMICRO Newsletter* 23.1-2 (Dec. 1992), pp. 102–110. ISSN: 1050-916X. DOI: 10.1145/144965.145006.

[67] Carlo A. Furia et al. “Modeling Time in Computing: A Taxonomy and a Comparative Survey”. en. In: *ACM Computing Surveys* 42.2 (Feb. 2010), pp. 1–59. ISSN: 03600300. DOI: 10.1145/1667062.1667063.

[68] G. Gao et al. “Collective Loop Fusion for Array Contraction”. In: *Languages and Compilers for Parallel Computing*. Ed. by Utpal Banerjee et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 281–295. ISBN: 978-3-540-48201-7.

[69] Jamie Garside. “Real-Time Prefetching On Shared-Memory Multi-Core Systems”. Doctoral. York, UK: The University of York, July 2015.

[70] Michael Gautschi et al. “Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices”. en. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (Oct. 2017), pp. 2700–2713. ISSN: 1063-8210, 1557-9999. DOI: 10.1109/TVLSI.2017.2654506.

[71] Gideon Glass and Pei Cao. “Adaptive Page Replacement Based on Memory Reference Behavior”. In: *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS ’97. New York, NY, USA: ACM, 1997, pp. 115–126. ISBN: 978-0-89791-909-8. DOI: 10.1145/258612.258681.

[72] J.I. Gomez et al. “Optimizing the Memory Bandwidth with Loop Morphing”. In: *Proceedings. 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2004.* Sept. 2004, pp. 213–223. DOI: 10.1109/ASAP.2004.1342472.

[73] Robert B Gramacy et al. “Adaptive Caching by Refetching”. In: *Advances in Neural Information Processing Systems 15*. Ed. by S. Becker, S. Thrun and K. Obermayer. MIT Press, 2003, pp. 1489–1496.

[74] Xiaoming Gu and Chen Ding. “On the Theory and Potential of LRU-MRU Collaborative Cache Management”. In: *Proceedings of the International Symposium on Memory Management*. ISMM '11. New York, NY, USA: ACM, 2011, pp. 43–54. ISBN: 978-1-4503-0263-0. DOI: `10.1145/1993478.1993485`.

[75] Erika Gunadi and Mikko H Lipasti. “Cache pipelining with partial operand knowledge”. In: *Proceedings of the Workshop on Complexity-Effective Design*. 2004.

[76] Jan Gustafsson et al. “The Mälardalen WCET Benchmarks – Past, Present and Future”. In: *WCET2010*. Ed. by Björn Lisper. Brussels, Belgium: OCG, July 2010, pp. 137–147.

[77] E.G. Hallnor and S.K. Reinhardt. “A Fully Associative Software-Managed Cache Design”. In: *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*. June 2000, pp. 107–116. DOI: `10.1145/339647.339660`.

[78] Fazal Hameed, Lars Bauer and Jörg Henkel. “Adaptive Cache Management for a Combined SRAM and DRAM Cache Hierarchy for Multi-Cores”. In: *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2013, pp. 77–82. DOI: `10.7873/DATE.2013.030`.

[79] Fazal Hameed, Lars Bauer and Jörg Henkel. “Reducing Latency in an SRAM/-DRAM Cache Hierarchy via a Novel Tag-Cache Architecture”. In: *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. June 2014, pp. 1–6. DOI: `10.1145/2593069.2593197`.

[80] H. Hassan et al. “ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality”. In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Mar. 2016, pp. 581–593. DOI: `10.1109/HPCA.2016.7446096`.

[81] Spinal HDL. *VexRiscv*. Spinal HDL. Nov. 2018.

[82] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Sixth. Cambridge, MA: Morgan Kaufmann Publishers, 2019. ISBN: 978-0-12-811905-1.

[83] John L. Hennessy and David A. Patterson. *Computer Organization and Design: The Hardware/Software Interface*. RISC-V edition. Cambridge, Massachusetts: Morgan Kaufmann Publishers, an imprint of Elsevier, 2018. ISBN: 978-0-12-812275-4.

[84] Seokin Hong and Soontae Kim. “AVICA: An Access-Time Variation Insensitive L1 Cache Architecture”. In: *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2013, pp. 65–70. DOI: 10.7873/DATE.2013.028.

[85] D. Hormdee, J.D. Garside and S.B. Furber. “An Asynchronous Victim Cache”. In: *Proceedings Euromicro Symposium on Digital System Design. Architectures, Methods and Tools*. Sept. 2002, pp. 4–11. DOI: 10.1109/DSD.2002.1115345.

[86] X. Hu and S. Chen. “Applications of On-Chip Trace on Debugging Embedded Processor”. In: *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*. Vol. 1. July 2007, pp. 140–145. DOI: 10.1109/SNPD.2007.227.

[87] *IEEE-ISTO 5001-2003 - The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface*. Standard. New Jersey, USA: IEEE Industry Standards and Technology Organization, June 2012.

[88] P. Jain et al. “Software-Assisted Cache Replacement Mechanisms for Embedded Systems”. en. In: IEEE, 2001, pp. 119–126. ISBN: 978-0-7803-7247-4. DOI: 10.1109/ICCAD.2001.968607.

[89] Aamer Jaleel et al. “High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)”. en. In: *Proceedings of the 37th Annual International Symposium on Computer Architecture - ISCA '10*. Saint-Malo, France: ACM Press, 2010, p. 60. ISBN: 978-1-4503-0053-7. DOI: 10.1145/1815961.1815971.

[90] J. Jeong and M. Dubois. “Cost-Sensitive Cache Replacement Algorithms”. en. In: IEEE Comput. Soc, 2003, pp. 327–337. ISBN: 978-0-7695-1871-8. DOI: 10.1109/HPCA.2003.1183550.

[91] Jaeheon Jeong and Michel Dubois. “Optimal Replacements in Caches with Two Miss Costs”. In: *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’99. New York, NY, USA: ACM, 1999, pp. 155–164. ISBN: 978-1-58113-124-6. DOI: 10.1145/305619.305636.

[92] Song Jiang and Xiaodong Zhang. “LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance”. en. In: *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems - SIGMETRICS '02*. Marina Del Rey, California: ACM Press, 2002, p. 31. ISBN: 978-1-58113-531-2. DOI: 10.1145/511334.511340.

[93] Theodore Johnson and Dennis Shasha. “2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm”. In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB ’94. San Francisco,

CA, USA: Morgan Kaufmann Publishers Inc., Sept. 1994, pp. 439–450. ISBN: 978-1-55860-153-6.

[94] Jongmoo Choi et al. "Design, Implementation, and Performance Evaluation of a Detection-Based Adaptive Block Replacement Scheme". en. In: *IEEE Transactions on Computers* 51.7 (July 2002), pp. 793–800. ISSN: 0018-9340. DOI: 10.1109/TC.2002.1017699.

[95] N.P. Jouppi and S.J.E. Wilton. "Tradeoffs in Two-Level on-Chip Caching". In: *Proceedings of 21 International Symposium on Computer Architecture*. Apr. 1994, pp. 34–45. DOI: 10.1109/ISCA.1994.288163.

[96] Norman P. Jouppi. "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers". In: *Proceedings of the 17th Annual International Symposium on Computer Architecture*. ISCA '90. New York, NY, USA: ACM, 1990, pp. 364–373. ISBN: 978-0-89791-366-9. DOI: 10.1145/325164.325162.

[97] F. Juan and L. Chengyan. "An Improved Multi-Core Shared Cache Replacement Algorithm". In: *2012 11th International Symposium on Distributed Computing and Applications to Business, Engineering Science*. Oct. 2012, pp. 13–17. DOI: 10.1109/DCABES.2012.39.

[98] Martin Kampe, Per Stenstrom and Michel Dubois. "Self-Correcting LRU Replacement Policies". In: *In Proceedings of the 1st Conference on Computing Frontiers*. 2004, pp. 181–191.

[99] R. Karedla, J. S. Love and B. G. Wherry. "Caching Strategies to Improve Disk System Performance". In: *Computer* 27.3 (Mar. 1994), pp. 38–46. ISSN: 0018-9162. DOI: 10.1109/2.268884.

[100] Russell Kegley et al. *Predictive Cache Modeling and Analysis*. en. Technical Report AFRL-RI-RS-TR-2011-266. Fort Worth, TX: Lockheed Martin Aeronautics Corporation, Nov. 2011.

[101] Terence Kelly, Sugih Jamin and Jeffrey K. MacKie-Mason. "Variable Qos from Shared Web Caches: User-Centered Design and Value-Sensitive Replacement". en. In: *SSRN Electronic Journal* (1999). ISSN: 1556-5068. DOI: 10.2139/ssrn.975737.

[102] K. Kelwade et al. "Reputation Based Cache Management Policy for Performance Improvement". In: *2017 International Conference on Intelligent Sustainable Systems (ICISS)*. Dec. 2017, pp. 582–587. DOI: 10.1109/ISS1.2017.8389236.

[103] Georgios Keramidas, Pavlos Petoumenos and Stefanos Kaxiras. "Cache Replacement Based on Reuse-Distance Prediction". en. In: IEEE, Oct. 2007, pp. 245–250. ISBN: 978-1-4244-1257-0. DOI: 10.1109/ICCD.2007.4601909.

[104] R.E. Kessler et al. “Inexpensive Implementations Of Set-Associativity”. In: *The 16th Annual International Symposium on Computer Architecture*. May 1989, pp. 131–139. DOI: 10.1109/ISCA.1989.714547.

[105] S. M. Khan, Z. Wang and D. A. Jiménez. “Decoupled Dynamic Cache Segmentation”. In: *IEEE International Symposium on High-Performance Comp Architecture*. Feb. 2012, pp. 1–12. DOI: 10.1109/HPCA.2012.6169030.

[106] Samira Khan et al. “Using Dead Blocks as a Virtual Victim Cache”. In: *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Sept. 2010, pp. 489–500.

[107] Mazen Kharbutli and Rami Sheikh. “LACS: A Locality-Aware Cost-Sensitive Cache Replacement Algorithm”. In: *IEEE Transactions on Computers* 63.8 (Aug. 2014), pp. 1975–1987. ISSN: 2326-3814. DOI: 10.1109/TC.2013.61.

[108] Mazen Kharbutli and Yan Solihin. “Counter-Based Cache Replacement Algorithms”. In: *Proceedings of the 2005 International Conference on Computer Design*. ICCD ’05. USA: IEEE Computer Society, Oct. 2005, pp. 61–68. ISBN: 978-0-7695-2451-1. DOI: 10.1109/ICCD.2005.41.

[109] J. Kim et al. “Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines”. In: *2018 IEEE 36th International Conference on Computer Design (ICCD)*. Oct. 2018, pp. 282–291. DOI: 10.1109/ICCD.2018.00051.

[110] Jong Min Kim et al. “A Low-Overhead High-Performance Unified Buffer Management Scheme That Exploits Sequential and Looping References”. In: *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*. OSDI’00. Berkeley, CA, USA: USENIX Association, 2000.

[111] Jun Kiniwa and Tiko Kameda. “Lookahead Scheduling Requests for Efficient Paging”. en. In: *RIMS Kôkyûroku*. Algorithms and Theory of Computing 1041 (Apr. 1998), pp. 27–34.

[112] David Kroft. “Lockup-Free Instruction Fetch/Prefetch Cache Organization”. In: *Proceedings of the 8th Annual Symposium on Computer Architecture*. ISCA ’81. Minneapolis, Minnesota, USA: IEEE Computer Society Press, May 1981, pp. 81–87.

[113] Jaekyu Lee, Hyesoon Kim and Richard Vuduc. “When Prefetching Works, When It Doesn’t, and Why”. In: *ACM Transactions on Architecture and Code Optimization* 9.1 (Mar. 2012), 2:1–2:29. ISSN: 1544-3566. DOI: 10.1145/2133382.2133384.

[114] Lin Li and Albrecht Mayer. "Trace-Based Analysis Methodology of Program Flash Contention in Embedded Multicore Systems". en. In: *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Research Publishing Services, 2016, pp. 199–204. ISBN: 978-3-9815370-7-9. DOI: `10.3850/9783981537079_0442`.

[115] Z. Li, D. Liu and H. Bi. "CRFP: A Novel Adaptive Replacement Policy Combined the LRU and LFU Policies". In: *2008 IEEE 8th International Conference on Computer and Information Technology Workshops*. July 2008, pp. 72–79. DOI: `10.1109/CIT.2008.Workshops.22`.

[116] Ankur Limaye and Tosiron Adegbija. "A Workload Characterization of the SPEC CPU2017 Benchmark Suite". In: *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Belfast: IEEE, Apr. 2018, pp. 149–158. ISBN: 978-1-5386-5010-3. DOI: `10.1109/ISPASS.2018.00028`.

[117] Wei-Fen Lin, S.K. Reinhardt and D. Burger. "Reducing DRAM Latencies with an Integrated Memory Hierarchy Design". In: *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. Jan. 2001, pp. 301–312. DOI: `10.1109/HPCA.2001.903272`.

[118] Wei-Fen Lin and Steven K Reinhardt. *Predicting Last-Touch References under Optimal Replacement*. en. Technical Report CSE-TR-447-02. University of Michigan, 2002, p. 17.

[119] M.H. Lipasti et al. "SPAID: Software Prefetching in Pointer- and Call-Intensive Environments". In: *Proceedings of the 28th Annual International Symposium on Microarchitecture*. Nov. 1995, pp. 231–236. DOI: `10.1109/MICRO.1995.476830`.

[120] Haiming Liu et al. "Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency". In: *2008 41st IEEE/ACM International Symposium on Microarchitecture*. Nov. 2008, pp. 222–233. DOI: `10.1109/MICRO.2008.4771793`.

[121] Gabriel H. Loh and Mark D. Hill. "Efficiently Enabling Conventional Block Sizes for Very Large Die-Stacked DRAM Caches". In: *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Dec. 2011, pp. 454–464.

[122] Chi-Keung Luk and Todd C. Mowry. "Compiler-Based Prefetching for Recursive Data Structures". In: *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VII. Cambridge, Massachusetts, USA: Association for Computing Machinery, Sept. 1996, pp. 222–233. ISBN: 978-0-89791-767-4. DOI: `10.1145/237090.237190`.

[123] L. Luo et al. “Design and Realization of an Optimized Memory Access Scheduler”. In: *2010 Third International Joint Conference on Computational Science and Optimization*. Vol. 2. May 2010, pp. 288–292. DOI: 10.1109/CSO.2010.81.

[124] R Manikantan, Kaushik Rajan and R Govindarajan. “NUcache: An Efficient Multicore Cache Organization Based on Next-Use Distance”. In: *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. Feb. 2011, pp. 243–253. DOI: 10.1109/HPCA.2011.5749733.

[125] P. Marchal, F. Catthoor and J.I. Gomez. “Optimizing the Memory Bandwidth with Loop Fusion”. In: *International Conference on Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004.* Sept. 2004, pp. 188–193. DOI: 10.1109/CODESS.2004.241216.

[126] A.J. Martin et al. “The Design of an Asynchronous MIPS R3000 Microprocessor”. In: *Proceedings Seventeenth Conference on Advanced Research in VLSI*. Sept. 1997, pp. 164–181. DOI: 10.1109/ARVLSI.1997.634853.

[127] R. L. Mattson et al. “Evaluation techniques for storage hierarchies”. In: *IBM Systems Journal* 9.2 (1970), pp. 78–117. DOI: 10.1147/sj.92.0078.

[128] Nimrod Megiddo and Modha Dharmendra S. “ARC: A Self-Tuning, Low Overhead Replacement Cache”. en. In: *Proceedings of FAST ’03: 2nd USENIX Conference on File and Storage Technologies*. Vol. 3. San Francisco: USENIX Association, Apr. 2003, pp. 115–130.

[129] Nagi N Mekhiel. “Multi-Level Cache With Most Frequently Used Policy: A New Concept In Cache Design”. en. In: *Proceedings of the ISCA 8th International Conference*. Honolulu, Hawaii, Nov. 29, p. 5. ISBN: 1-880843-14-5.

[130] Jhonny Mertz and Ingrid Nunes. “On the Practical Feasibility of Software Monitoring: A Framework for Low-Impact Execution Tracing”. In: *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. May 2019, pp. 169–180. DOI: 10.1109/SEAMS.2019.00030.

[131] Edson T. Midorikawa, Ricardo L. Piantola and Hugo H. Cassettari. “On Adaptive Replacement Based on LRU with Working Area Restriction Algorithm”. In: *SIGOPS Oper. Syst. Rev.* 42.6 (Oct. 2008), pp. 81–92. ISSN: 0163-5980. DOI: 10.1145/1453775.1453790.

[132] A. Modgil and V. K. Sehgal. “Improving the Performance of Chip Multiprocessor by Delayed Write Drain and Prefetcher Based Memory Scheduler”. In: *2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*. Mar. 2018, pp. 1864–1869. DOI: 10.1109/ICECA.2018.8474846.

[133] Todd C. Mowry, Monica S. Lam and Anoop Gupta. “Design and Evaluation of a Compiler Algorithm for Prefetching”. In: *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS V. Boston, Massachusetts, USA: Association for Computing Machinery, Sept. 1992, pp. 62–73. ISBN: 978-0-89791-534-2. DOI: `10.1145/143365.143488`.

[134] O. Mutlu et al. “Runahead Execution: An Alternative to Very Large Instruction Windows for out-of-Order Processors”. In: *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* Feb. 2003, pp. 129–140. DOI: `10.1109/HPCA.2003.1183532`.

[135] O. Mutlu et al. “Runahead Execution: An Effective Alternative to Large Instruction Windows”. In: *IEEE Micro* 23.6 (Nov. 2003), pp. 20–25. ISSN: 1937-4143. DOI: `10.1109/MM.2003.1261383`.

[136] Osvaldo Navarro and Michael Hübner. “An Adaptive Victim Cache Scheme”. In: *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*. Dec. 2014, pp. 1–4. DOI: `10.1109/ReConFig.2014.7032496`.

[137] H. Neefs, H. Vandierendonck and K. De Bosschere. “A Technique for High Bandwidth and Deterministic Low Latency Load/Store Accesses to Multiple Cache Banks”. In: *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*. Jan. 2000, pp. 313–324. DOI: `10.1109/HPCA.2000.824360`.

[138] K.J. Nesbit, A.S. Dhodapkar and J.E. Smith. “AC/DC: An Adaptive Data Cache Prefetcher”. In: *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* Oct. 2004, pp. 135–145. DOI: `10.1109/PACT.2004.1342548`.

[139] K.J. Nesbit and J.E. Smith. “Data Cache Prefetching Using a Global History Buffer”. In: *10th International Symposium on High Performance Computer Architecture (HPCA’04)*. Feb. 2004, pp. 96–96. DOI: `10.1109/HPCA.2004.10030`.

[140] *Nios II Processor Reference Guide*. July 2019.

[141] Elizabeth J. O’Neil, Patrick E. O’Neil and Gerhard Weikum. “The LRU-K Page Replacement Algorithm for Database Disk Buffering”. en. In: ACM Press, 1993, pp. 297–306. ISBN: 978-0-89791-592-2. DOI: `10.1145/170035.170081`.

[142] K. Olukotun, T. N. Mudge and R. B. Brown. “Multilevel Optimization of Pipelined Caches”. In: *IEEE Transactions on Computers* 46.10 (Oct. 1997), pp. 1093–1102. DOI: `10.1109/12.628394`.

[143] Kunle Olukotun, Trevor Mudge and Richard Brown. "Performance Optimization of Pipelined Primary Cache". en. In: *ACM SIGARCH Computer Architecture News* 20.2 (June 1992), pp. 181–190. ISSN: 01635964. DOI: 10.1145/146628.139726.

[144] *Open Source Hardware Association - Homepage*. https://www.oshwa.org/.

[145] Noritaka Osawa, Toshitsugu Yuba and Katsuya Hakozaki. "Generational Replacement Schemes for a WWW Caching Proxy Server". en. In: *High-Performance Computing and Networking*. Ed. by Gerhard Goos et al. Vol. 1225. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 940–949. ISBN: 978-3-540-69041-2. DOI: 10.1007/BFb0031665.

[146] T. Ozawa, Y. Kimura and S. Nishizaki. "Cache Miss Heuristics and Preloading Techniques for General-Purpose Programs". In: *Proceedings of the 28th Annual International Symposium on Microarchitecture*. Nov. 1995, pp. 243–248. DOI: 10.1109/MICRO.1995.476832.

[147] Peter S. Pacheco. "Parallel Hardware and Parallel Software". en. In: *An Introduction to Parallel Programming*. Elsevier, 2011, pp. 15–81. ISBN: 978-0-12-374260-5. DOI: 10.1016/B978-0-12-374260-5.00002-6.

[148] S. Palacharla and R. E. Kessler. "Evaluating Stream Buffers as a Secondary Cache Replacement". In: *ACM SIGARCH Computer Architecture News* 22.2 (Apr. 1994), pp. 24–33. ISSN: 0163-5964. DOI: 10.1145/192007.192014.

[149] Y. Pan and T. Zhang. "Improving VLIW Processor Performance Using Three-Dimensional (3D) DRAM Stacking". In: *2009 20th IEEE International Conference on Application-Specific Systems, Architectures and Processors*. July 2009, pp. 38–45. DOI: 10.1109/ASAP.2009.11.

[150] B. Panda and S. Balachandran. "Expert Prefetch Prediction: An Expert Predicting the Usefulness of Hardware Prefetchers". In: *IEEE Computer Architecture Letters* 15.1 (Jan. 2016), pp. 13–16. ISSN: 1556-6056. DOI: 10.1109/LCA.2015.2428703.

[151] Parag Panda, Geeta Patil and Biju Raveendran. "A Survey on Replacement Strategies in Cache Memory for Embedded Systems". en. In: *2016 IEEE Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER)*. Mangalore, India: IEEE, Aug. 2016, pp. 12–17. ISBN: 978-1-5090-1623-5. DOI: 10.1109/DISCOVER.2016.7806218.

[152] Preeti Ranjan Panda, Nikil D. Dutt and Alexandru Nicolau. "Memory Data Organization for Improved Cache Performance in Embedded Processor Applications". en. In: *ACM Transactions on Design Automation of Electronic Sys-*

tems (TODAES) 2.4 (Oct. 1997), pp. 384–409. ISSN: 1084-4309, 1557-7309. DOI: 10.1145/268424.268464.

[153] Gajinder Panesar and Iain Robertson, eds. *RISC-V Processor Trace*. Mar. 2020.

[154] D. Patterson et al. “A Case for Intelligent RAM”. In: *IEEE Micro* 17.2 (1997), pp. 34–44. ISSN: 02721732. DOI: 10.1109/40.592312.

[155] *PG172 - Integrated Logic Analyzer v6.2 - LogiCORE IP Product Guide*. Oct. 2016.

[156] *PG267 - AXI Verification IP v1.1 - LogiCORE IP Product Guide*. Apr. 2017.

[157] James E Pitkow and Margaret M Recker. “A Simple Yet Robust Caching Algorithm Based on Dynamic Access Patterns”. en. In: *Proceedings of the Second International WWW Conference*. Oct. 1994, p. 8.

[158] Stefan Podlipnig and Laszlo Böszörmenyi. “A Survey of Web Cache Replacement Strategies”. In: *ACM Comput. Surv.* 35.4 (Dec. 2003), pp. 374–398. ISSN: 0360-0300. DOI: 10.1145/954339.954341.

[159] Mounika Ponugoti and Aleksandar Milenkovic. “Enabling On-the-Fly Hardware Tracing of Data Reads in Multicores”. en. In: *ACM Transactions on Embedded Computing Systems* 18.4 (June 2019), pp. 1–27. ISSN: 15399087. DOI: 10.1145/3322642.

[160] S. Przybylski, M. Horowitz and J. Hennessy. “Characteristics Of Performance-Optimal Multi-Level Cache Hierarchies”. In: *The 16th Annual International Symposium on Computer Architecture*. May 1989, pp. 114–121. DOI: 10.1109/ISCA.1989.714545.

[161] S. Przybylski, M. Horowitz and J. Hennessy. “Performance Tradeoffs in Cache Design”. In: *[1988] The 15th Annual International Symposium on Computer Architecture. Conference Proceedings*. May 1988, pp. 290–298. DOI: 10.1109/ISCA.1988.5239.

[162] Steven A. Przybylski. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*. en. San Mateo, Calif: Morgan Kaufmann Publishers, 1990. ISBN: 978-1-55860-136-9.

[163] PULP Foundation. *Trace Debugger for RISC-V Core*. June 2018.

[164] A. Qazi et al. “Optimization of Access Latency in DRAM”. In: *2016 International Conference on Computing, Electronic and Electrical Engineering (ICE Cube)*. Apr. 2016, pp. 163–168. DOI: 10.1109/ICECUBE.2016.7495216.

[165] M. K. Qureshi et al. “Set-Dueling-Controlled Adaptive Insertion for High-Performance Caching”. In: *IEEE Micro* 28.1 (Jan. 2008), pp. 91–98. DOI: 10.1109/MM.2008.14.

[166] M.K. Qureshi, D. Thompson and Y.N. Patt. “The V-Way Cache: Demand-Based Associativity via Global Replacement”. In: *32nd International Symposium on Computer Architecture (ISCA'05)*. June 2005, pp. 544–555. DOI: 10.1109/ISCA.2005.52.

[167] Moinuddin K. Qureshi et al. “A Case for MLP-Aware Cache Replacement”. In: *ACM SIGARCH Computer Architecture News* 34.2 (May 2006), pp. 167–178. ISSN: 0163-5964. DOI: 10.1145/1150019.1136501.

[168] Moinuddin K. Qureshi et al. “Adaptive Insertion Policies for High Performance Caching”. en. In: *ACM SIGARCH Computer Architecture News* 35.2 (June 2007), p. 381. ISSN: 01635964. DOI: 10.1145/1273440.1250709.

[169] Jonathan Rainer. *Experimental Data (Including Hardware Variants) Measuring Memory Activity for Trace Assisted Caching*. en. Sept. 2020. DOI: 10.5281/ZENODO.4042892.

[170] Jonathan Rainer. *Experimental Data (Including Hardware Variants) Measuring Runtime for Trace Assisted Caching*. en. Sept. 2020. DOI: 10.5281/ZENODO.4040337.

[171] Jonathan Rainer. *Jonathanrainer/Ichijou: Initial Release*. Zenodo. Sept. 2020. DOI: 10.5281/ZENODO.4045225.

[172] Jonathan Rainer. *Jonathanrainer/Kuuga: Initial Release*. Zenodo. Sept. 2020. DOI: 10.5281/ZENODO.4045227.

[173] Jonathan Rainer. *Jonathanrainer/Sawatari: Initial Release*. Zenodo. Sept. 2020. DOI: 10.5281/ZENODO.4045229.

[174] A. Ramirez, J.Ll. Larriba-Pey and M. Valero. “Trace Cache Redundancy: Red and Blue Traces”. In: *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*. Jan. 2000, pp. 325–333. DOI: 10.1109/HPCA.2000.824361.

[175] Mike Reddy and Graham P. Fletcher. “Intelligent Web Caching Using Document Life Histories: A Comparison with Existing Cache Management Techniques”. In: 1998.

[176] Jeff Reilly. “A brief introduction to the SPEC CPU95 benchmarks”. In: *SPEC Newsletter, Sep* (1995).

[177] Faizal Riaz-ud-Din and Markus Kirchberg. “Acme-DB: An Adaptive Caching Mechanism Using Multiple Experts for Database Buffers”. en. In: *Enterprise Information Systems VI*. Ed. by Isabel Seruca et al. Berlin/Heidelberg: Springer-Verlag, 2006, pp. 72–81. ISBN: 978-1-4020-3674-3. DOI: 10.1007/1-4020-3675-2_9.

[178] J.A. Rivers et al. "On High-Bandwidth Data Cache Design for Multi-Issue Processors". In: *Proceedings of 30th Annual International Symposium on Microarchitecture*. Dec. 1997, pp. 46–56. DOI: `10.1109/MICRO.1997.645796`.

[179] Scott Rixner et al. "Memory Access Scheduling". In: *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ISCA '00. Vancouver, British Columbia, Canada: Association for Computing Machinery, May 2000, pp. 128–138. ISBN: 978-1-58113-232-8. DOI: `10.1145/339647.339668`.

[180] Luigi Rizzo and Lorenzo Vicisano. "Replacement Policies for a Proxy Cache". In: *IEEE/ACM Transactions on Networking* 8.2 (Apr. 2000), pp. 158–170. ISSN: 1063-6692. DOI: `10.1109/90.842139`.

[181] John T. Robinson and Murthy V. Devarakonda. "Data Cache Management Using Frequency-Based Replacement". In: *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '90. New York, NY, USA: ACM, 1990, pp. 134–142. ISBN: 978-0-89791-359-1. DOI: `10.1145/98457.98523`.

[182] E. Rotenberg, S. Bennett and J.E. Smith. "A Trace Cache Microarchitecture and Evaluation". In: *IEEE Transactions on Computers* 48.2 (Feb. 1999), pp. 111–120. ISSN: 2326-3814. DOI: `10.1109/12.752652`.

[183] Amir Roth, Andreas Moshovos and Gurindar S. Sohi. "Dependence Based Prefetching for Linked Data Structures". In: *ACM SIGPLAN Notices* 33.11 (Oct. 1998), pp. 115–126. ISSN: 0362-1340. DOI: `10.1145/291006.291034`.

[184] Daniel Sanchez and Christos Kozyrakis. "The ZCache: Decoupling Ways and Associativity". In: *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. Dec. 2010, pp. 187–198. DOI: `10.1109/MICRO.2010.20`.

[185] T. Scheipel, F. Mauroner and M. Baunach. "System-Aware Performance Monitoring Unit for RISC-V Architectures". In: *2017 Euromicro Conference on Digital System Design (DSD)*. Aug. 2017, pp. 86–93. DOI: `10.1109/DSD.2017.28`.

[186] André Seznec. "A Case for Two-Way Skewed-Associative Caches". In: *Proceedings of the 20th Annual International Symposium on Computer Architecture*. ISCA '93. San Diego, California, USA: Association for Computing Machinery, May 1993, pp. 169–178. ISBN: 978-0-8186-3810-7. DOI: `10.1145/165123.165152`.

[187] André Seznec and Francois Bodin. "Skewed-Associative Caches". en. In: *PARLE '93 Parallel Architectures and Languages Europe*. Ed. by Arndt Bode, Mike Reeve and Gottfried Wolf. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1993, pp. 305–316. ISBN: 978-3-540-47779-2. DOI: `10.1007/3-540-56891-3_24`.

[188] Jun Shao and Brian T. Davis. "A Burst Scheduling Access Reordering Mechanism". en. In: *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Scottsdale, AZ, USA: IEEE, 2007, pp. 285–294. ISBN: 978-1-4244-0804-7. DOI: `10.1109/HPCA.2007.346206`.

[189] W. Shin et al. "DRAM-Latency Optimization Inspired by Relationship between Row-Access Time and Refresh Timing". In: *IEEE Transactions on Computers* 65.10 (Oct. 2016), pp. 3027–3040. ISSN: 0018-9340. DOI: `10.1109/TC.2015.2512863`.

[190] O. Shoukry et al. "Proactive Scheduling for Content Pre-Fetching in Mobile Networks". In: *2014 IEEE International Conference on Communications (ICC)*. June 2014, pp. 2848–2854. DOI: `10.1109/ICC.2014.6883756`.

[191] Amit Kumar Singh et al. "Resource and Throughput Aware Execution Trace Analysis for Efficient Run-Time Mapping on MPSoCs". en. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.1 (Jan. 2016), pp. 72–85. ISSN: 0278-0070, 1937-4151. DOI: `10.1109/TCAD.2015.2446938`.

[192] Yannis Smaragdakis. "General Adaptive Replacement Policies". en. In: ACM Press, 2004, p. 108. ISBN: 978-1-58113-945-7. DOI: `10.1145/1029873.1029887`.

[193] Yannis Smaragdakis, Scott Kaplan and Paul Wilson. "EELRU: Simple and Effective Adaptive Page Replacement". In: *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '99. New York, NY, USA: ACM, 1999, pp. 122–133. ISBN: 978-1-58113-083-6. DOI: `10.1145/301453.301486`.

[194] Alan Jay Smith. "Cache Memories". In: *ACM Comput. Surv.* 14.3 (Sept. 1982), pp. 473–530. ISSN: 0360-0300. DOI: `10.1145/356887.356892`.

[195] Leonid B. Sokolinsky. "LFU-K: An Effective Buffer Management Replacement Algorithm". en. In: *Database Systems for Advanced Applications*. Ed. by YoonJoon Lee et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 670–681. ISBN: 978-3-540-24571-1. DOI: `10.1007/978-3-540-24571-1_60`.

[196] Yan Solihin, Jaejin Lee and Josep Torrellas. "Using a User-Level Memory Thread for Correlation Prefetching". In: *ACM SIGARCH Computer Architecture News* 30.2 (May 2002), pp. 171–182. ISSN: 0163-5964. DOI: `10.1145/545214.545235`.

[197] S. Sreedharan and S. Asokan. "A Cache Replacement Policy Based on Re-Reference Count". In: *2017 International Conference on Inventive Communication and Computational Technologies (ICICCT)*. Mar. 2017, pp. 129–134. DOI: `10.1109/ICICCT.2017.7975173`.

[198] A. Srivastava et al. "190-MHz CMOS 4-Kbyte Pipelined Caches". In: *Proceedings of ISCAS'95 - International Symposium on Circuits and Systems*. Vol. 2. Apr. 1995, 1053–1056 vol.2. DOI: `10.1109/ISCAS.1995.519948`.

[199] D. Stiliadis and A. Varma. “Selective Victim Caching: A Method to Improve the Performance of Direct-Mapped Caches”. In: *IEEE Transactions on Computers* 46.5 (May 1997), pp. 603–610. ISSN: 2326-3814. DOI: `10.1109/12.589235`.

[200] J. Stuecheli et al. “Coordinating DRAM and Last-Level-Cache Policies with the Virtual Write Queue”. In: *IEEE Micro* 31.1 (Jan. 2011), pp. 90–98. ISSN: 0272-1732. DOI: `10.1109/MM.2010.102`.

[201] S. Subha. “An Architecture for Victim Cache”. In: *2016 2nd International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB)*. Feb. 2016, pp. 255–258. DOI: `10.1109/AEEICB.2016.7538284`.

[202] Ranjith Subramanian, Yannis Smaragdakis and Gabriel H. Loh. “Adaptive Caches: Effective Shaping of Cache Behavior to Workloads”. In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 385–396. ISBN: 978-0-7695-2732-1. DOI: `10.1109/MICRO.2006.7`.

[203] Jubee Tada. “A Cache Replacement Policy with Considering Global Fluctuations of Priority Values”. en. In: *International Journal of Networking and Computing* 9.2 (July 2019), pp. 161–170. ISSN: 2185-2847.

[204] Ju-Ho Tang and Kimming So. “Performance and Design Choices of Level-Two Caches”. In: *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*. Vol. 1. Jan. 1994, pp. 422–430. DOI: `10.1109/HICSS.1994.323143`.

[205] Geng Tian and Michael Liebelt. “An Effectiveness-Based Adaptive Cache Replacement Policy”. In: *Microprocessors and Microsystems* 38.1 (Feb. 2014), pp. 98–111. ISSN: 0141-9331. DOI: `10.1016/j.micpro.2013.11.011`.

[206] Andreas Traber and Micheal Gautschi. *PULPino: Datasheet*. June 2017.

[207] P. Tsai, N. Beckmann and D. Sanchez. “Jenga: Software-Defined Cache Hierarchies”. In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. June 2017, pp. 652–665. DOI: `10.1145/3079856.3080214`.

[208] R. F. Tsao, L. W. Comeau and B. H. Margolin. “A Multi-Factor Paging Experiment: I. The Experiment and Conclusions”. en. In: *Statistical Computer Performance Evaluation*. Ed. by Walter Freiberger. Academic Press, Jan. 1972, pp. 103–134. ISBN: 978-0-12-266950-7. DOI: `10.1016/B978-0-12-266950-7.50012-8`.

[209] James Tuck, Luis Ceze and Josep Torrellas. "Scalable Cache Miss Handling for High Memory-Level Parallelism". In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. Dec. 2006, pp. 409–422. doi: 10.1109/MICRO.2006.44.

[210] Rollins Turner and Henry Levy. "Segmented FIFO Page Replacement". en. In: ACM Press, 1981, pp. 48–51. isbn: 978-0-89791-051-4. doi: 10.1145/800189.805473.

[211] *UG885 - VC707 Evaluation Board for the Virtex-7 FPGA User Guide*. en. Feb. 2019.

[212] *UG974 - UltraScale Architecture Libraries Guide*. en. Apr. 2018.

[213] Vladimir Uzelac and Aleksandar Milenković. "Hardware-Based Load Value Trace Filtering for On-the-Fly Debugging". In: *ACM Trans. Embed. Comput. Syst.* 12.2s (May 2013), 97:1–97:18. issn: 1539-9087. doi: 10.1145/2465787.2465799.

[214] A. I. Vakali. "LRU-Based Algorithms for Web Cache Replacement". en. In: *Electronic Commerce and Web Technologies*. Ed. by Kurt Bauknecht, Sanjay Kumar Madria and Günther Pernul. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2000, pp. 409–418. isbn: 978-3-540-44463-3. doi: 10.1007/3-540-44463-7_36.

[215] S.P. Vander Wiel and D.J. Lilja. "A Compiler-Assisted Data Prefetch Controller". In: *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No.99CB37040)*. Oct. 1999, pp. 372–377. doi: 10.1109/ICCD.1999.808569.

[216] A. V. Veidenbaum and K. A. Gallivan. "Decoupled Access DRAM Architecture". In: *Proceedings Innovative Architecture for Future Generation High-Performance Processors and Systems*. Oct. 1997, pp. 94–103. doi: 10.1109/IWIA.1997.670415.

[217] R. Wang, Y. Gao and G. Zhang. "Real Time Cache Performance Analyzing for Multi-Core Parallel Programs". In: *2013 International Conference on Cloud and Service Computing*. Nov. 2013, pp. 16–23. doi: 10.1109/CSC.2013.11.

[218] Zhenlin Wang et al. "Using the Compiler to Improve Cache Replacement Decisions". In: *Proceedings.International Conference on Parallel Architectures and Compilation Techniques*. Sept. 2002, pp. 199–208. doi: 10.1109/PACT.2002.1106018.

[219] Tripti S. Warrier, Kanakagiri Raghavendra and Madhu Mutyam. "SkipCache: Application Aware Cache Management for Chip Multi-Processors". In: *IET Computers Digital Techniques* 9.6 (2015), pp. 293–299. issn: 1751-861X. doi: 10.1049/iet-cdt.2014.0150.

[220] S. Wasly and R. Pellizzoni. "Hiding Memory Latency Using Fixed Priority Scheduling". In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Apr. 2014, pp. 75–86. DOI: 10.1109/RTAS.2014.6925992.

[221] Andrew Waterman and Krste Asanovic, eds. *The RISC-V Instruction Set Manual*. en. June 2019.

[222] Wei-Che Tseng et al. "Optimal Scheduling to Minimize Non-Volatile Memory Access Time with Hardware Cache". In: *2010 18th IEEE/IFIP International Conference on VLSI and System-on-Chip*. Sept. 2010, pp. 131–136. DOI: 10.1109/VLSISOC.2010.5642609.

[223] Wei-Che Tseng et al. "PRR: A Low-Overhead Cache Replacement Algorithm for Embedded Processors". In: *17th Asia and South Pacific Design Automation Conference*. Jan. 2012, pp. 35–40. DOI: 10.1109/ASPDAC.2012.6164972.

[224] W. Wei et al. "Exploiting Program Semantics to Place Data in Hybrid Memory". In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. Oct. 2015, pp. 163–173. DOI: 10.1109/PACT.2015.10.

[225] Jack Whitham and Neil Audsley. "Time-Predictable Out-of-Order Execution for Hard Real-Time Systems". en. In: *IEEE Transactions on Computers* 59.9 (Sept. 2010), pp. 1210–1223. ISSN: 0018-9340. DOI: 10.1109/TC.2010.109.

[226] M. V. Wilkes. "Slave Memories and Dynamic Storage Allocation". en. In: *IEEE Transactions on Electronic Computers* EC-14.2 (Apr. 1965), pp. 270–271. ISSN: 0367-7508. DOI: 10.1109/PGEC.1965.264263.

[227] Maurice V. Wilkes. "The Memory Gap and the Future of High Performance Memories". en. In: *ACM SIGARCH Computer Architecture News* 29.1 (Mar. 2001), pp. 2–7. ISSN: 01635964. DOI: 10.1145/373574.373576.

[228] W.A. Wong and J.-L. Baer. "Modified LRU Policies for Improving Second-Level Cache Behavior". In: *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*. Jan. 2000, pp. 49–60. DOI: 10.1109/HPCA.2000.824338.

[229] Christopher Wood, Eduardo B. Fernandez and Tomas Lang. "Minimization of Demand Paging for the LRU Stack Model of Program Behavior". en. In: *Information Processing Letters* 16.2 (Feb. 1983), pp. 99–104. ISSN: 0020-0190. DOI: 10.1016/0020-0190(83)90034-0.

[230] Xiaoxia Wu et al. "Hybrid Cache Architecture with Disparate Memory Technologies". In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA '09. Austin, TX, USA: Association for Computing Machinery, June 2009, pp. 34–45. ISBN: 978-1-60558-526-0. DOI: 10.1145/1555754.1555761.

[231] Zhen Yang et al. “Overlapping Dependent Loads with Addressless Preload”. In: *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Sept. 2006, pp. 275–284.

[232] X. Yu et al. “IMP: Indirect Memory Prefetcher”. In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Dec. 2015, pp. 178–190. DOI: 10.1145/2830772.2830807.

[233] Chenxi Zhang, Xiaodong Zhang and Yong Yan. “Multi-Column Implementations for Cache Associativity”. In: *Proceedings International Conference on Computer Design VLSI in Computers and Processors*. Oct. 1997, pp. 504–509. DOI: 10.1109/ICCD.1997.628915.

[234] Chuanjun Zhang and Bing Xue. “Divide-and-Conquer: A Bubble Replacement for Low Level Caches”. In: *Proceedings of the 23rd International Conference on Supercomputing*. ICS ’09. New York, NY, USA: ACM, 2009, pp. 80–89. ISBN: 978-1-60558-498-0. DOI: 10.1145/1542275.1542291.

[235] Zheng Zhang and Josep Torrellas. “Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching”. In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. ISCA ’95. S. Margherita Ligure, Italy: Association for Computing Machinery, May 1995, pp. 188–199. ISBN: 978-0-89791-698-1. DOI: 10.1145/223982.224423.

[236] Li Zhao et al. “Exploring DRAM Cache Architectures for CMP Server Platforms”. In: *2007 25th International Conference on Computer Design*. Oct. 2007, pp. 55–62. DOI: 10.1109/ICCD.2007.4601880.

[237] Yuanyuan Zhou, James Philbin and Kai Li. “The Multi-Queue Replacement Algorithm for Second Level Buffer Caches”. In: *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*. USA: USENIX Association, June 2001, pp. 91–104. ISBN: 978-1-880446-09-6.

[238] Hussein Al-Zoubi, Aleksandar Milenkovic and Milena Milenkovic. “Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite”. en. In: *Proceedings of the 42nd Annual Southeast Regional Conference on - ACM-SE 42*. Huntsville, Alabama: ACM Press, 2004, p. 267. ISBN: 978-1-58113-870-2. DOI: 10.1145/986537.986601.