REAL-TIME PREFETCHING ON SHARED-MEMORY MULTI-CORE SYSTEMS

JAMIE GARSIDE

DOCTOR OF PHILOSOPHY

UNIVERSITY OF YORK

COMPUTER SCIENCE

JULY 2015

Abstract

In recent years, there has been a growing trend towards using multi-core processors in real-time systems to cope with the rising computation requirements of real-time tasks. Coupled with this, the rising memory requirements of these tasks pushes demand beyond what can be provided by small, private on-chip caches, requiring the use of larger, slower off-chip memories such as DRAM. Due to the cost, power requirements and complexity of these memories, they are typically shared between all of the tasks within the system.

In order for the execution time of these tasks to be bounded, the response time of the memory and the interference from other tasks also needs to be bounded. While there is a great amount of current research on bounding this interference, one popular method is to effectively partition the available memory bandwidth between the processors in the system. Of course, as the number of processors increases, so does the worst-case blocking, and worst-case blocking times quickly increase with the number of processors.

It is difficult to further optimise the arbitration scheme; instead, this scaling problem needs to be approached from another angle. Prefetching has previously been shown to improve the execution time of tasks by speculatively issuing memory accesses ahead of time for items which may be useful in the near future, although these prefetchers are typically not used in real-time systems due to their unpredictable nature. Instead, this work presents a framework by which a prefetcher can be safely used alongside a composable memory arbiter, a predictable prefetching scheme, and finally a method by which this predictable prefetcher can be used to improve the worst-case execution time of a running task.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

Acknowledgements

Firstly, I'd like to thank my supervisor Neil Audsley for his help and guidance over the last four years, and for the opportunity to work on interesting projects and attend numerous conferences throughout; your help and support has been invaluable while doing my PhD. I'd also like to thank my assessors, Leandro Indrusiak, Stratis Viglas and Wim Vanderbauwhede for an enjoyable and interesting viva.

Huge thanks also go out to my friends and colleagues in CSE/120 over the past few years - Ian Gray, Gary Plumbridge, Jack Whitham, David George, Gareth Lloyd and Russell Joyce. Thanks for making a fun and welcoming, albeit at times unproductive, work environment and for your help over the course of this PhD. Thanks also to the rest of my friends in the Computer Science department and further afield throughout my time at York.

Finally, thanks go out to my family and to my partner Ione for their love and support throughout the highs, and especially the lows of the past four years - I could not have done it without you.

Declaration

I declare that all work contained within this thesis is a result of my own investigations, except where explicit attribution has been given. The content of some of the chapters has already been published within the following publications:

- Jamie Garside and Neil C. Audsley: Prefetching Across a Shared Memory Tree within a Network-on-Chip Architecture. *International Symposium on System-on-Chip, 2013*.
- Jamie Garside and Neil C. Audsley: Investigating Shared Memory Tree Prefetching within Multimedia NoC Architectures. *Memory Architecture and Organisation Workshop, 2013*.
- Jamie Garside and Neil C. Audsley: WCET Preserving Hardware Prefetch for Many-Core Real-Time Systems. *22nd International Conference on Real-Time Networks and Systems, 2014*.

This work has not previously been presented for an award at this, or any other, University.

1 INTRODUCTION

Within recent years, the breakdown of Dennard scaling [13] has limited the ability for processor designers to improve system performance by increasing the overall system clock speed. Instead, to maintain expected year-on-year performance increases, system designers have instead turned to scaling the *number* of cores on a chip rather than the core speed. Today, eight-core processor designs are becoming commonplace, and multi-core scaling is set only to continue; EZChip [14] are currently creating 72-core designs, Parallela are creating 64-core designs which can be connected together for a maximum of 256 cores and Intel are aiming for over 60 cores on the Knight's Landing platform [15].

While these systems are allowing apparent processor performance to meet expected year-on-year improvements, they are beginning to run into an important problem. The "memory gap" has been known for the last couple of decades [16], that is, that the memory subsystem simply cannot cope with the memory bandwidth requirements of modern tasks. As the number of cores accessing a single memory increase, this effect is only going to worsen, and memory latencies are only going to increase.

This causes significant problems within the field of real-time systems, which by their very nature need to know the worst-case response time of all the components within a system. This rising memory latency causes the estimate for the worst-case time to execute an instruction which accesses memory to significantly increase and hence, the worst-case time to execute the task as a whole increases. As the number of cores sharing memory increases, this effect will only worsen further to potentially unacceptable levels.

There are many techniques by which some of this latency can be alleviated, one of which being prefetching. This attempts to utilise spare bandwidth to speculatively fetch data ahead of time, although it is generally not used within real-time systems because the behaviour of the prefetcher is difficult to predict and may actually cause the worst-case execution time of a task to *worsen*. That said, a controlled and predictable form of prefetching is an attractive prospect to try and reduce this ever-increasing memory delay by using any "spare" system bandwidth to try and reduce the execution time of other tasks.

This thesis will attempt to investigate the effects of prefetching on real-time multi-core systems, and attempt to create a prefetching scheme under which the worst-case execution time of a task can be improved. Firstly though, the remainder of this chapter will explain the background behind the move to multi-core systems

and the effects this has on real-time systems to provide the intuition behind the work in the remainder of this thesis. It will then provide a research hypothesis which this thesis will attempt to prove, before closing with a plan for the remainder of this thesis.

1.1 BACKGROUND

Up until the last decade, two observations managed to adequately capture the trends within computer architectures. Moore's Law [17] states that, in effect, the number of transistors on an integrated circuit for which the cost per transistor is optimal doubles around every 18 months. Dennard Scaling [13] then states that as the feature size of transistors falls, the power *density* of transistors remains constant and thus the power required for two identically sized dies should be the same, regardless of feature size. In effect, Moore's Law provides extra transistors for new features, and Dennard Scaling combined with improved manufacturing processors allow the transistors to be used within a similar power budget.

While Moore's observations are still relevant today, Dennard Scaling has begun to break down [18]. To maintain the same power density as the number of transistors in a unit area increases, the power utilised *per transistor* naturally must decrease, for which one realisation is to reduce the operating voltage of the transistor. Within recent years, this operating voltage has been lowered so far that sub-threshold leakage[1] becomes a real issue and begins to limit the voltage scaling of transistors. As transistors are made smaller, the power density can no longer remain constant and hence heat dissipation begins to become a real problem within systems.

As clock frequency increases also require more power, this led to clock frequency scaling of microprocessors to plateau around 2003 [20], mainly because it is difficult to dissipate the heat generated by both more transistors and faster transistors. This problem is then combined with increasing wire delays caused by using smaller wires [21]; as wire diameter decreases, the resistance of the wire increases and hence the time taken to switch the state of the wire between "off" and "on" increases because of the capacitor time constant $\tau = RC$. The combination of these factors, amongst others, makes it difficult to increase the clock speed further. Instead, to use the transistors provided by Moore's Law and to provide the expected year-on-year performance increases, system designers instead turned to using multiple cores on a single die.

1 Field Effect Transistors (FETs) used within computer circuits are effectively used as digital switches. When there is a sufficient positive voltage between the "gate" and "source" pins of the FET (called the threshold voltage), it will allow current to flow between the "source" and the "drain" pins. Sub-threshold leakage is the current which can flow when this condition has not yet been met. This leakage is inversely proportional to the threshold voltage [19], which Dennard Scaling assumes can be lowered with transistor size in order to maintain constant power density.

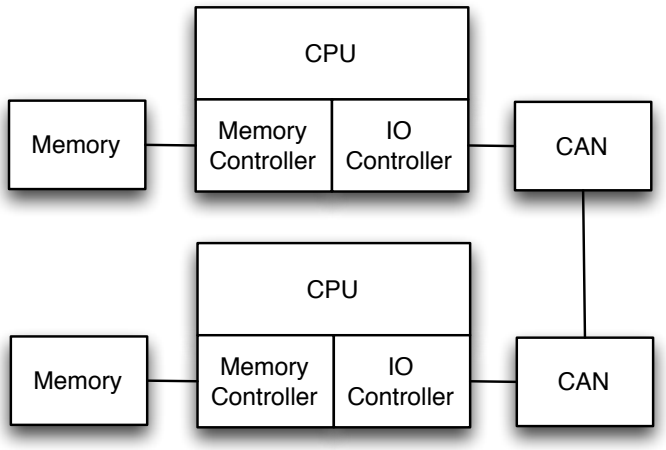


Figure 1.1: Example multi-CPU system, comprised of two independent CPUs with private memories.

While increasing the number of cores on a single chip allows for good performance increases, it does cause a new problem. Single-core processors typically contained their own memory controller, and were connected to their own memory. Any communication between processors then had to take place using a given system bus, for example, Ethernet or CAN. An example of such a system is shown in Figure 1.1, where there are two single-core processors with exclusive access to their attached memory. If data must be shared between these processors, it must be done explicitly through the attached CAN interface.

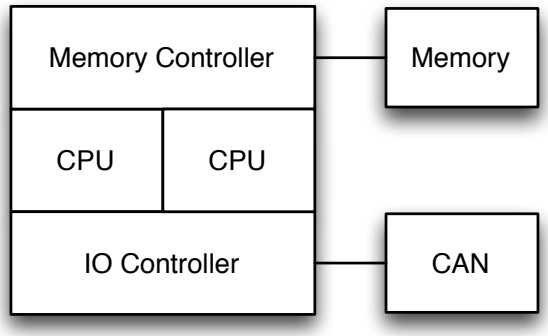


Figure 1.2: Example multi-core system, comprised of a dual-core CPU with a single shared memory.

Due to many factors, allowing each core to retain its own private, software managed memory is infeasible in multi-core processors. Constraints such as die area, power utilisation and the number of package pins [16] limit the number of memory modules that can be connected to a chip and hence make it impossible for each processing core to have its own large, private memory. Instead, each core must

share the global system resources, creating a system such as that shown in Figure 1.2. Within this system, both cores share a single common memory controller and I/O controller. For the aforementioned reasons, these controllers are rarely multi-channel and hence if both cores want to access memory at the same time, there will be contention and one of the cores will be granted access to memory, causing the other to have to wait.

This contention causes significant issues within the field of real-time systems. A real-time task is typically characterised by the fact that it must respond to an input stimulus with a set period of time for the output to be valid. There are various examples of such systems; flight control systems must process the pilot's inputs quickly and update the state of the aircraft's control surfaces quickly enough to be responsive, video decoders must emit a new frame with a set deadline to ensure smooth video playback. To make such guarantees, a real-time task can be evaluated against a model of the system to ascertain a worst-case execution time for a portion of the task's code. Given that the execution time of a block of code can be bounded, it is then possible to assert that a task can respond to a given input in a given period of time.

To construct this system model, it must be possible to determine how long each component in the system will take to respond in the worst-case. For a task which uses external memory, it must therefore be possible to ascertain the worst-case time to access memory in order to create a worst-case bound on the task as a whole. For a system using single-core processors such as that in Figure 1.1, the worst-case response time of the memory is simply the worst-case time for the memory controller to operate. In a multi-*core* system such as that in Figure 1.2 with a shared memory controller, however, the problem is much more complex. Not only does the response time of the memory controller need to be known, but it must also be possible to ascertain what the other processors are doing at that point in time to determine if any of their memory accesses will block any given memory access.

Analysing how these tasks will interfere is possible [22, 23], but difficult; if the tasks running on each processor can be started and stopped at will, the blocking caused between each set of tasks which may execute together must be considered, a problem which is exponential in nature [24] and hence soon becomes infeasible as the number of tasks grows.

A method by which this complexity can be alleviated is to assign each task a partition of the available system bandwidth which describes the portion of the system bandwidth which can be given to a task, and the maximum latency between issuing a request and the request being given service. These constraints are then enforced using a hardware arbiter sitting between the processors and memory, which ensures that no processor can cause any other processor to receive less than its defined bandwidth. A revised version of the system design with an arbiter is

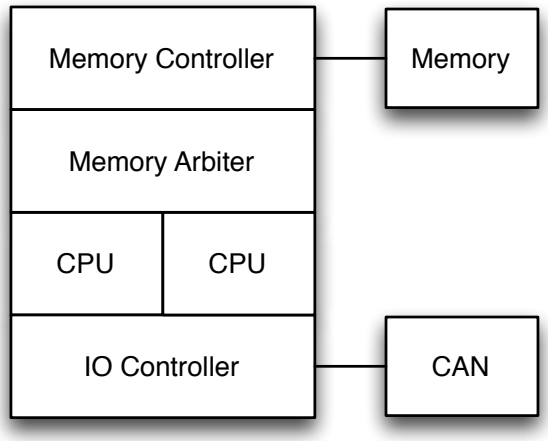


Figure 1.3: Example multi-core system, comprised of a dual-core CPU with a single arbitrated shared memory.

shown in Figure 1.3. By splitting the system bandwidth like this, the system model can use the given constraints of the assigned partition when determining the worst-case response time for a memory access. Because the bandwidth partitioning is enforced by the arbiter, the interference caused by other processors need not be considered when analysing the execution time of a task, since the arbiter will ensure that the task receives the level of service defined by the partition.

While this approach makes the worst-case analysis of the system simpler, and can provide additional safety guarantees, it does have its problems. This approach effectively places a static bound on the amount of blocking that any memory request can experience, and thus assumes that all other processors are fully utilising their bandwidth bound (or even that all available system bandwidth has been allocated). If this is not the case, then there will be "spare" system bandwidth available. Moreover, because the estimate for the worst-case response time of a memory access assumes a request will be blocked by many other processors issuing memory accesses, the worst-case response time of a memory access increases as the number of processors increases.

This problem is being further exaggerated by another issue. While processors have seen bountiful year-on-year performance increases, seeing year-on-year performance increases of 1.25x to 1.52x, the memory side of things has not been so lucky. Memory performance scaling has only been growing year-on-year at a rate of around 1.07x [1], leading to the so called "memory gap" as seen in Figure 1.4. As processor speeds increase and the degree of memory sharing increases, as will the apparent cost of accessing memory.

There has been some improvements in memory technology to alleviate this performance gap; modern DDR memories contain multiple "banks" of memory which can be accessed in parallel to attempt to alleviate contention issues when sharing

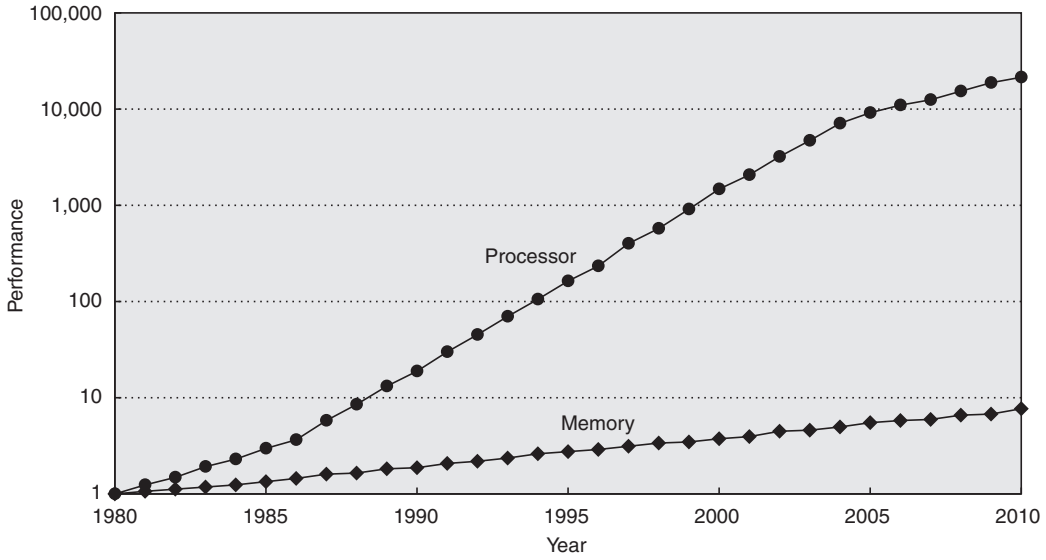


Figure 1.4: The growing gap between CPU and memory performance [1]

a single DDR memory. As an example, Micron's 1Gb DDR3 modules are arranged as eight banks [11], with newer DDR4 memories containing up to sixteen banks of memory [25]. Of course though, each bank can only be accessed by one requester at once, hence the system designer must ensure that all banks are being utilised in a uniform manner by all requesters to ensure optimal system performance. While this reduces the interference which a task may experience when accessing memory, there is still the very real problem of contention within the banks, and the actual rising latency of a single memory request to deal with.

Of course, this rising memory latency has spawned much work to attempt to mask these latencies. One of these methods is caching [26], which attempts to store recently used data close to the processor to attempt to drastically reduce the latency of accessing frequently used data. Examples of where this is useful are loops, where the same code will be executed many times, or for a few data items which are frequently accessed (e.g. counters or state variables). While this technique can mask the latency of frequently accessed items, it does nothing to attempt to mask the latency of the "initial" load of a data item. Caching therefore does not have much of a positive effect on a task which accesses each member of a data set only once, or has very large, straight blocks of code since every memory access will cause a cache miss (i.e. the required data will never be resident in cache).

As mentioned earlier, systems which use partitioned bandwidth may not fully utilise the available bandwidth, leading to some "spare" bandwidth within the system. Another method to alleviate the rising worst-case cost that can also use some of this spare bandwidth is prefetching. A prefetcher is a functional unit within the system which attempts to predict what a requester will soon require,

and issue a memory request on behalf of that requester ahead of time. As an example, if a requester accesses addresses A, $A+1$ and $A+2$ in sequence, it will probably require the data at address $A+3$ in the near future. This is a rather naïve metric; other, smarter metrics are explored further within Section 2.4. If the prefetcher was accurate, it will reduce the latency of a memory access vastly, because the required data will already have been delivered to the processor ahead of time. This method can also be combined with caching; caching can reduce the latency of repeatedly accessed data, and prefetching can attempt to fetch the "next" data into cache ahead of time to reduce the cost when data has not been previously accessed.

Ultimately, however, both techniques cause issues when used within real-time systems as they modify the *context* under which a task is being executed. In order to determine the latency of a memory access when a cache is used, the system model must also consider all other memory accesses which have previously taken place to determine whether the required data is resident in cache. While this is simple for single path tasks, it is generally undecidable if a task's control flow depends on its input data [27]. As the memory access pattern then depends upon the path taken through the program, a static analysis tool cannot accurately determine what data has been fetched and hence what is stored in cache unless the input data is known a priori. This leads to an estimate for the worst-case execution time which is *lower* due to the inclusion of the cache, but pessimistic because the behaviour of the cache cannot be predicted with perfect accuracy.

Prefetching also suffers from a similar problem: the state of the prefetcher is context sensitive depending on what accesses the prefetcher has observed where again, the run-time behaviour of a task based upon its inputs will change the behaviour of the prefetcher. Moreover, the accesses which the prefetcher observes are based upon the behaviour of the cache. If the system analysis cannot determine for definite whether a requested data item is resident in cache, then it cannot also determine whether the prefetcher observes said data access and hence cannot make any guarantees about the state of the prefetcher. Because the cache analysis is undecidable, the prefetcher's state is hence undecidable and thus the set of memory accesses issued by the prefetcher are undecidable. On top of this, many prefetchers fetch directly into the processor's cache rather than any form of holding buffer, hence causing further unpredictability regarding the contents of the processor's cache; the prefetcher may have fetched useful data, but could well also have displaced useful data already resident in cache, hence invalidating much of the existing cache analysis.

Despite these problems, something needs to be done to slow down the rising memory latencies in real-time systems. This work explores using prefetching within the context of a real-time system to attempt to improve the time which an executing task is blocked waiting for memory accesses to complete. This chap-

ter will first further explore the reasons for moving to shared-memory multi-core systems, before using this information to form a research hypothesis for the remainder of this thesis and finally providing a structure for the remainder of this thesis.

1.2 RESEARCH HYPOTHESIS

As introduced in Section 1.1, caching is a good way to hide some of this memory latency. The problem is, caching does not actually reduce memory latency, only hides the latency involved with fetching repeated data, which it does by moving recently accessed data closer to the processor into faster storage. Caches also typically fetch data on the granularity of an entire cache line (of the order of 16-32 bytes of data), hence also hide the latency of accessing some subsequent memory locations to the one originally accessed. While this optimises the number of memory accesses that take place though, it does not actually optimise *how* these memory accesses take place; even if some memory accesses no longer take place, those that do will still incur large latency costs.

It is clear that if memory delays are to increase along this trajectory for the foreseeable future, the memory system must be optimised as a whole. Moreover, something needs to be done about the growing pessimism within real-time system analysis brought about by the memory subsystem. Prefetching is an attractive optimisation to use for this; by pushing data out from memory to the processors speculatively, the costs involved with memory access can be reduced, or even removed entirely.

Of course though, this causes problems for real-time systems. A prefetch unit within the system speculatively requesting data from memory can cause extra interference for running tasks, and since prefetchers normally deliver data directly into the cache of the target processor, it may also invalidate any cache analysis. For the most part however, these limitations are simply because prefetching, as of yet, has not been considered under the constraints of a real-time system. If the prefetcher can be constrained in a way such that it is possible to predict *what* it will do and *when*, there is no real reason why it cannot be used in a real-time system.

This thesis will therefore explore the following hypothesis:

> A prefetcher can be constructed so that it can be used within a real-time system in a predictable way, such that it does not cause any detriment to the worst-case execution time of the tasks running within that system. Furthermore, it is possible under certain circumstances to predict when the prefetcher will operate, allowing it to utilise "spare" or unallocated bandwidth within the system to actively improve the worst-case execution time estimate of a task running in a real-time system.

1.3 THESIS STRUCTURE

From this, the remainder of the thesis will be structured as follows:

CHAPTER 2 will detail the related literature in the subjects of multi-core systems, including how to predict the behaviour of DRAM, how to arbitrate multiple requesters fairly for DRAM, and current work related to prefetching on both single and multi-core systems.

CHAPTER 3 will, given the related literature, further concrete the chosen research avenue and provide a research hypothesis

CHAPTER 4 demonstrates the effects brought about by using a traditional, non-arbitrated prefetcher and the potential system analysis issues it brings about.

CHAPTER 5 will provide a framework by which a prefetcher can be integrated into a real-time system without affecting the worst-case execution time of the system.

CHAPTER 6 will then provide a method by which a prefetcher can be used in order to improve the calculated worst-case execution time.

CHAPTER 7 will then, finally, provide the major conclusions identified and outline the future research opportunities from this project.

2 BACKGROUND & RELATED WORK

This chapter will provide a concise overview of the related literature within the fields of work associated with real-time prefetching. This will first give an overview on the operation of memory within systems, the problems it poses within a real-time system, and how these issues are handled within the current state of the art. It will then provide an overview of current caching and prefetching schemes, before detailing how shared resources are arbitrated in both monolithic and distributed fashions. It will then give a brief overview of scheduling theory and how predictability is asserted, before summing up by outlining the methods by which a program can be analysed and a worst-case bound ascertained.

2.1 PREDICTABILITY

There are many variations on the definition of a real-time system. The common definition is effectively that "a system which must, by a defined time period, be able to react to an external stimulus from the environment". An external stimulus may be anything from a user pressing a button on the system to a video transcoder receiving video to a nuclear reactor reaching the point of no return before meltdown.

Of course, each of these scenarios are vastly different in terms of scale. A user will only be slightly inconvenienced if a system takes a while to respond to the play button being pushed. If a video encoder fails, there might be some frame skip. If a reactor goes critical or an aircraft fails to respond to input, it could lead to loss of life. For this reason, the real-time community tends to split up the definition of real time between hard real-time, firm real-time and soft real-time (HRT, FRT and SRT, respectively). These are defined by Burns and Wellings [28] as follows:

HARD REAL-TIME Where it is absolutely imperative that the system reacts within its given time frame, else there may be disastrous consequences.

FIRM REAL-TIME Where the deadline may be missed occasionally, but there is no benefit from it being late.

SOFT REAL-TIME Where the deadline may be missed occasionally, or that a service can occasionally be delivered late.

In the case of both FRT and SRT tasks, there may be an upper limit on the number of times there can be a deadline miss. To quote the video encoder example again, a couple of late frames is not *too* bad; it just leads to a slightly poor experience. Many late frames within a short interval, on the other hand, renders the video almost unwatchable. Additionally, as stated also by Burns and Wellings, a system may have both HRT and SRT requirements, for example, a warning system may have a SRT deadline of 50ms for optimal response, while a HRT deadline of 200ms also exists to prevent damage.

For this task, it is vital to determine the range of times in which a task may execute for in order to prove that a task may meet these constraints. As noted by Wilhelm et al. in their review of worst-case execution time prediction methods [2], this is generally done by observing the execution of a task under a variety of test cases and recording the monitored best and worst case execution times. Of course, this will overstate the best case and understate the worst case, as an exhaustive analysis can be difficult to perform and as such, is not applicable for HRT tasks. Standard techniques in industry for predicting the real worst case execution time (WCET) may place constraints on the activities that can be performed within a program (e.g. forbidding recursion).

Of course, there are a vast number of ways to arrive at a WCET estimate or bound, and the remainder of this section shall attempt to provide an overview of the techniques which can be used. Again, this is to attempt to bring the reader up to speed with the terminology and techniques used, nor explain each method in great depth and does not attempt to be an exhaustive search through the literature surrounding time predictability, which would be another review in itself. If desired, there are many reviews in existence, such as the 2000 review by Puschner and Burns [29] and a review from 2008 by Wilhelm et al. [2].

2.1.1 Deriving Predictability Estimates and Bounds

There are two main classes of approaches used to derive a predictability bound and/or estimate. In this context, a bound *cannot* be exceeded, else the analysis is wrong. An estimate, on the other hand, is not 100% guaranteed to be correct, but should be accompanied with an error margin.

Static Analysis

Static analysis is typically an offline technqiue which attempts to ascertain a worst-case bound of a given section of a task by using a model of the target architecture [30]. This is performed in a number of steps [31]:

FLOW ANALYSIS, using the source and the final output, reconstructs all the possible paths through a program. For this, certain constructs, e.g. recursion, may not be permitted due to their ambiguity in analysis, or may have to be manually constrainted by the programmer to assert facts about them, for example, maximum loop iterations or recursion depth.

GLOBAL LOW-LEVEL ANALYSIS computes the factors affecting a task on a certain machine from the machine's global constructs, for example, cache.

LOCAL LOW-LEVEL ANALYSIS computes the same factors as above, but localised to a single task and code segment, for example, pipeline stalls and branch prediction.

The end result is then calculated from these three components. There may also be other inputs in order to make the task easier, for example, value analysis, as detailed by Ferdinand et al. [32] approximates the state that each processor register can be in at run time. This can then be used to compute the range of possible memory addresses that can occur at run time and thus aid in cache analysis.

Of course though, in order for this technique to be sound, it requires an extremely accurate model of the processor. This is simple for a basic architecture (for example, Z80); each instruction takes a fixed number of cycles and all memory accesses are single-cycle. Of course, modern architectures are much more complex than this. For example, in order to ascertain what the effect of a cache on the worst-case execution time is, the analysis must attempt ascertain what the contents of cache will be at any point in time. Persistence analysis is typically used for this [33], which for each memory access classifies if it is definitely not in cache (e.g. for the cases where analysis cannot definitely ascertain that the same cache line has already been loaded and not evicted), or those which *may* reside in cache (e.g. an earlier access in the flow analysis accessed the same line).

As will be explored further in Section 2.2, the unpredictability of memory also adds difficulty to the static analysis of a given task. The access time to a given location in memory depends upon the previous pattern of accesses; if an access is accessing the same DRAM line as the previous access, the memory access will be much faster than for a different row. Moreover, as multi-core systems become more prevalent, static analysis must ascertain how long an access to a shared memory resource takes given the interference from other tasks in the system. Techniques to derive this will be explored further in Section 2.3.

These factors lead to problems with pessimism in static analysis. As it is difficult to soundly assert conditions on the state of the system, many static analysis techniques must assert worst-case conditions on each access. As an example, worst-case memory access latencies are normally assumed and worst-case initial cache conditions must be assumed. Moreover, in order to make the problem feasible to solve, many analysis tools will limit the number of blocks that the state of caches is

asserted over, and hence may assume that cache blocks are missing when they will definitely reside in cache. Despite this pessimism, static analysis should always be able to find the worst-case path, assuming that the model of the processor is sound and is hence typically regarded as safer than measurement-based approaches.

Measurement-based Analysis

Rather on relying upon an accurate model of the processor, measurement-based analysis instead uses the behaviour of the processor itself to model the execution times [34], after all, "the best model of the processor is the processor itself". As with static analysis, measurement based techniques begin by re-constructing a flow graph of the program, where each block in the graph typically corresponds to a single-entry single-exit block of instructions which are executed sequentially, typically known as a basic block. The tool then executes the program a number of times under a set of inputs and environmental conditions, then times how long each of these blocks takes to execute to derive a distribution of probabilities for each block.

After this has taken place for each block, the execution times for each block can be combined according to the flow graph. If blocks are connected in a sequential fashion, then the execution times of each are summed, if they appear in parallel (e.g. in an if/then/else construct), then the maximum execution time of all of the parallel blocks is selected. The combination of all of these basic blocks then forms an estimate of the worst-case execution time of the entire task.

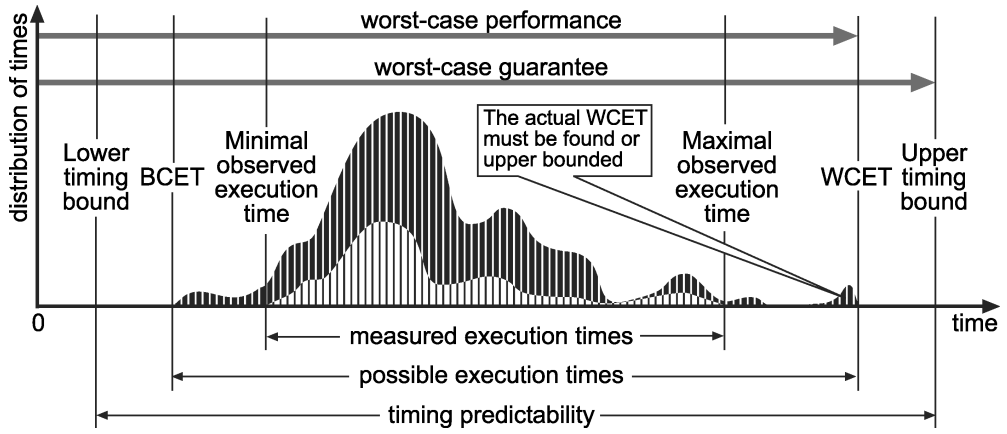


Figure 2.1: Graphical view of the executon times of a task, along with the relevant bounds [2].

In order to be sound, this approach must be able to assert that it has actually observed the worst-case path through the task and the worst-case conditions of the system. An example of this is shown in Figure 2.1, where the meaning of each item is explained as follows:

MEASURED EXECUTION TIMES: Observed maximum/minimum execution times for the task. These may not be accurate, as the actual maximum/minimum execution times may not have been observed.

BCET/WCET: Actual best and worst-case execution times. These may occur very infrequently and hence were never observed.

UPPER/LOWER TIMING BOUND: Best and worst case execution times with a safety margin added.

MEASURED EXECUTION TIMES: The set of all execution times observed when evaluating the worst-case behaviour of the task.

POSSIBLE EXECUTION TIMES: The set of all possible execution times for all different program paths, inputs and initial hardware conditions.

TIMING PREDICTABILITY: The possible range of execution times after the safety margin has been added.

In this example, the actual highest execution time may have been observed as "maximal observed execution time", but the actual worst-case under a different set of conditions may be much higher at the "WCET" line. The first solution is the observe the flow graph in order to assert that full program coverage has been achieved. If this cannot be asserted, then it is possible that a block of code has been missed with an extremely high actual execution time. The second solution is to exhaustively test all possible inputs, and execute the code with a sufficient number of iterations such that all possible system states have *probably* taken place.

While simpler than static analysis techniques, it is much more difficult to assert that a measurement-based analysis is sound because of the unpredictability of system compontents such as caches and shared memory. While dynamic anaysis does work on multi-core system, it does have to be constrained such that the memory arbiter is also operating in worst-case conditions in order for the actual worst-case execution time to be observed, which may again be difficult without modifying the arbiter itself.

2.1.2 Response Time

The predictability of a system is also not only dependant upon the worst case execution time of a single task. In general, the WCET of a task assumes a sole task running with no interference. Clearly, on a system, interrupts can arrive, a process can be pre-empted to make way for a higher priority task, or a required resource may not be available as another task it currently using it. For this reason, the deadline of a task is normally much longer than the actual WCET of a task. In general, the worst case response time R of a task i is denoted as $R_i = B_i + C_i + I_i$

as outlined by Burns and Wellings [28]. C_i is the WCET of process i and I_i is the interference process i receives, be it pre-emption, locked resources, or anything else influenced by *other* processes within the system. B_i is the blocking factor which the maximum time spent on acquiring shared resources and critical sections. This analysis goes on to detail the blocking factor I as follows, given fixed priority scheduling:

Interference I_i

Given all tasks in the system with a higher priority, $hp(i)$, the maximum amount of interference is as follows, given that T_i is the period of task i.

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

This equation can be extended to include the time taken to perform a context switch [28], the time to overcome a polluted cache from the pre-empting process [35], and generally the time to recover the state of the system at the time the process was pre-empted. An example extension to this is as follows:

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j + \gamma_j)$$

Where CS^1 and CS^2 are the times taken by the operating system to context switch away from and back to a task, respectively. γ_j is then the delay imposed on task i by the cache pollution introduced by task j.

Blocking B_i

The blocking factor B is a much simpler factor to compute. Assuming a priority inheritance protocol [36], where a task utilising a shared resource assumes the priority of the highest consumer of that resource when using it, the blocking factor is the sum of the time taken in all critical sections where said critical section is used by a process with lower priority than process i, and also by a process with equal or higher priority to i.

This blocking term exists solely because it is possible for a lower priority task to lock a resource used by a higher priority task (as the normal execution time with the lock is accounted for by I_i. Since priority inheritance ensures that each critical section can only be blocked by a lower priority task once for each critical section, for the duration of that critical section [36]. Hence, given K critical sections, that $usage(k, i)$ is 1 if the previously detailed conditions hold for critical section k and

task priority i (and 0 otherwise), and that $C(k)$ is the WCET of the critical section k:

$$B_i = \sum_{k=1}^{K} usage(k, i)C(k)$$

2.1.3 Priority Ceiling Protocol

The model detailed previously assumes a priority inheritance protocol is used, where a user of resource assumes the priority of the job it blocks with the highest priority. This is to prevent "priority inversion" from occurring, where a low priority task a obtains a shared resource, then is preempted. If a higher priority task b attempts to use the shared resource *without* a priority inheritance protocol, it will be blocked until task a has finished with the resource. If there are also other tasks with their priorities in between tasks a and b, task a will still have to wait for those tasks to complete (or block), and as such, have to wait much longer to execute its critical section, thus further blocking task b.

Another protocol is the Priority Ceiling Protocol, also suggested by Sha et al [36]. This was designed to address some of the flaws in priority inheritance, for example, that it is possible for transitive blocking to occur (high priority task c blocked by b blocked by low priority task a), and that deadlock is possible (e.g. a high priority task c wanting to lock a resource owned by a, which in turn wants to lock a resource owned by c). This takes two forms: the original ceiling priority protocol and the simpler immediate ceiling priority protocol.

From Burns and Wellings [28], the original priority ceiling protocol takes the following form.

1. Each task has a static default priority
2. Each resource has a static ceiling value defined, which is the maximum priority of the tasks which use it.
3. Each task has a dynamic priority, which is the higher of its own priority, and any inherited from blocking higher priority tasks (same as priority inheritance).
4. A task can only lock a resource if its dynamic priority is *higher* than the ceiling of any locked resource (unless those resources belong to that task).

Clause 4 prevents some of the properties listed above. Taking the deadlock example, when the higher priority task attempts to lock something which the lower priority task would require, since its dynamic priority is not strictly greater than

the current system ceiling. It would then not be allowed to access the resource and would be blocked by the lower priority task until said lower priority task had finished with the shared resource, and the system ceiling was again lowered.

The immediate ceiling priority protocol is similar, but much easier to implement. In this case, when a task acquires a shared resource, its priority is raised to the priority ceiling of the shared resource (that is, immediately assumes the priority of the highest priority task which could utilise the resource).

Within both priority ceiling protocols, the blocking factor B_i is changed to [28]:

$$B_i = \max_{k=1}^{K} usage(k, i)C(k)$$

This is, simply enough, the length of the longest critical section in any lower priority task which is shared with the current task. This holds because when a low priority task obtains a lock on a resource which could be used by a high priority task, no other tasks may lock any resource which can be used by the high priority task. Given that, a high priority task may be blocked by a single lower priority task. Assuming nested locks, a task may be blocked multiple times, but it may only be blocked by a single task.

2.2 MEMORY

Of course, in order for a processor to be able to perform any useful computation, it needs access to some form of memory. Traditionally, this would have been some form of static-RAM (SRAM) device, and is effectively implemented as a flip-flop connected to two transistors to control access to the cell. In a typical implementation, this utilises approxamately six transistors, as can be seen in Figure 2.2. In order to access SRAM, *W* line of the word to read is asserted by an address decoder based upon the input address, then the data stored in the flip-flop is sensed through the B lines.

This design leads to a chip which consumes reasonably little power and is fast. More importantly, however, is the fact that it is predictable; since the device is effectively a set of flip-flops connected to a multiplexer, the access time is related to the propogation delay of the internal logic, and both reads and writes typically take the same amount of time to complete.

While SRAM performs well and is simple to predict, the design of each cell makes it infeasible to use for large-scale shared memories typically required on modern systems. The number of components in each cell makes each cell large in silicon, hence it suffers from poor density and as a result it is expensive to create large SRAM chips. In a world dictated by cost, this makes DRAM an attractive

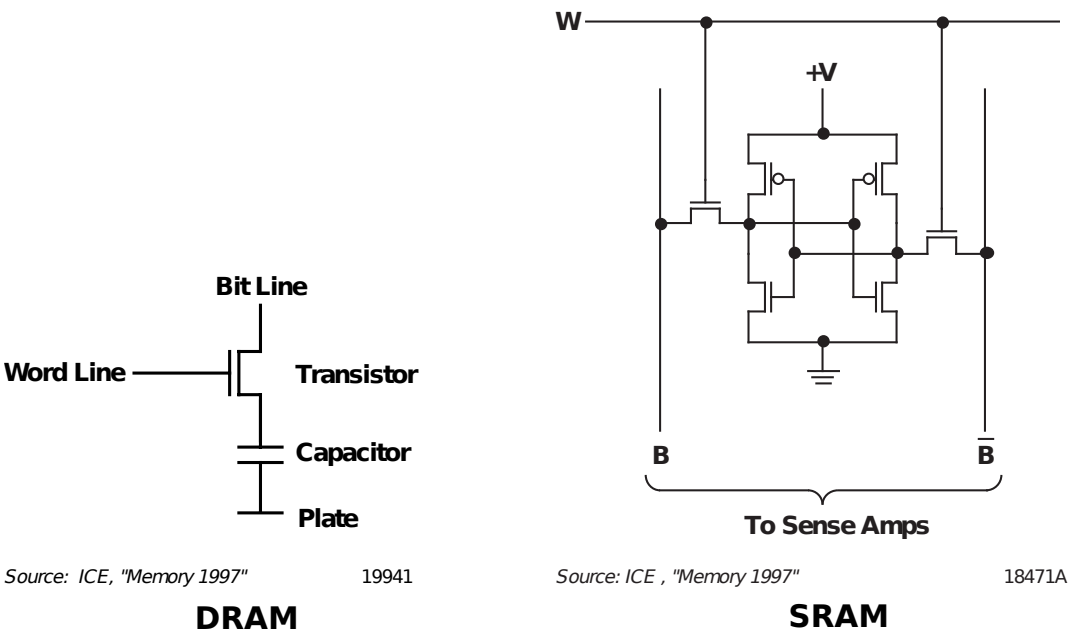


Figure 2.2: Comparison of SRAM and DRAM designs, adapted from [3]

.

alternative. Each DRAM cell is implemented using a capacitor connected to a single access transistor, as can be seen in the first half of Figure 2.2, giving each DRAM cell a much smaller footprint than the SRAM equivalent and hence allowing devices to have much higher storage densities. This makes DRAM attractive to any situation where large amounts of storage are required (e.g. external memory), leaving SRAM to be used for smaller amounts of faster storage, such as processor caches and scratchpads.

These density gains are not without any cost, however; DRAM has many flaws stemming from the usage of capacitors as a storage medium [5]. The first of these is that capacitors leak charge over time. In order to mitigate this issue, the DRAM controller must periodically refresh the contents of these capacitors in order to maintain the charge stored within and prevent bit flips from occuring. This takes a period of time within which the memory cannot accept any other memory requests, as the refresh cycle effectively reads an entire DRAM row out and writes it back again. As an example, a modern 1Gb Micron DDR4 module requires a refresh which takes 260ns every 7.8µs [25]. This both slows down the potential access time, and makes DRAM less predictable; the total access time now depends on whether a refresh cycle needs to take place first.

As RAM sizes increase, it is not practical to simply carry on adding more and more rows to the RAM chip; the resulting die would be extremely long and very narrow. To alleviate this, denser SRAM and DRAM cells add more DRAM rows, but also make each row longer; as an example, a single 256Mb DDR3 module has a row size of 2Kb [11]. Due to physical and electrical constraints, it is clearly impossible to transmit all 2048 of these bits from the package at once (to contrast,

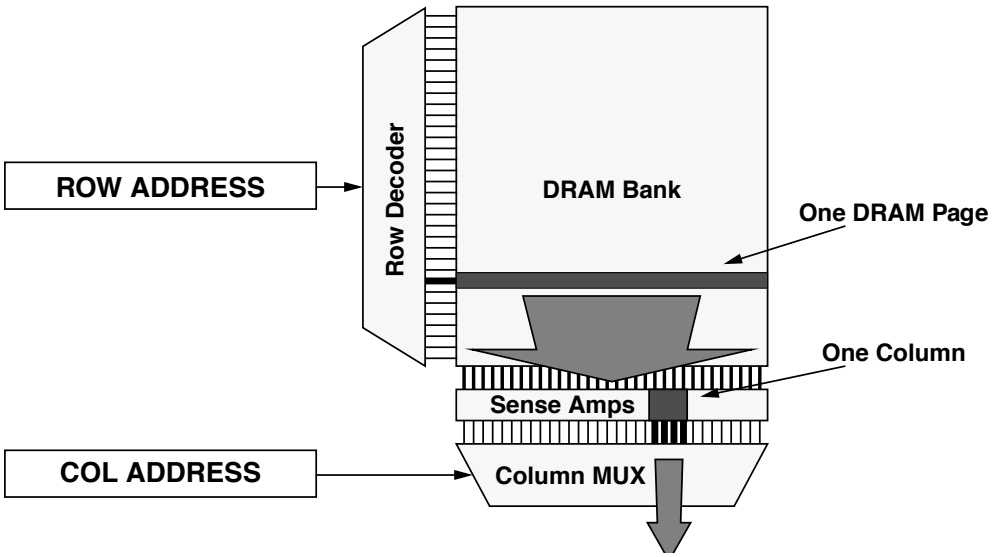


Figure 2.3: Internal organisation of DRAM cells, from [4]

.

the package has only 78 pins). Instead, a column decoder is also employed in order to select which bits within a row are required, and the RAM is addressed using a row/column scheme where part of the requested address form the row to read from, and the remainder select the respective column.

While SRAM cells can typically be addressed directly using this matrix scheme, DRAM is more difficult. Firstly, each bit is stored using a small capacitor (of the order of 20-30 fF [37]), hence there is not enough charge stored within to drive standard logic pins. Instead, when a row is selected, the cells in the addressed row drive a set of sense amplifiers, which detect the small amounts of charge (or lack thereof) and drive a logic '0' or '1'. Due to the capacitor time constant $T = RC$, this takes time, and hence there is a required time interval between selecting the row and being able to select the column. Due to this restriction, and to reduce device pin counts, the row and column are typically selected as two separate commands, denoted as "RAS" and "CAS".

There is also additional time as reading a DRAM cell is a destructive operation, since charge is drained from the capacitors to drive the sense amplifiers. Because of this, the sense amplifiers must write the data read back into the DRAM cells again in order to prevent data loss. After data has been read or written, the sense amplifiers must be reset to a known state again. This is done by driving them to a voltage level in-between logic '0' and '1' such that the small amount of charge stored in the capacitor (or lack thereof) will drive the charge stored in the sense amplifier towards the respective logic level through a process called "precharging". Both the writing back of read data and precharging of course, takes additional latency.

The net result of this is that there is a set of latencies that need to be adhered to in order to be able to utilise a DRAM module, as follows:

t_{RCD}: RAS-to-CAS latency. The time between issuing a row selection (RAS) and column selection (CAS)

t_{CAS}: CAS latency. The time between issuing a column selection and the selected data being available.

t_{RP}: Precharge delay. The time between a precharge being issued and being able to select a new row.

t_{RAS}: Active to precharge delay. The minimum time between opening a page (by issuing RAS) and closing it again (by issuing a precharge).

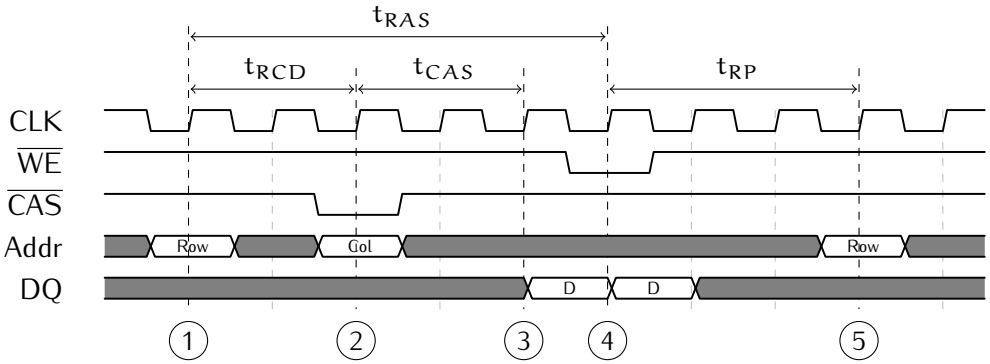


Figure 2.4: Example timing diagram for a DRAM read cycle, adapted from [5]. It is assumed that all commands are latched on the rising clock edge.

The relationships between these timing constraints can be found in Figure 2.4, where the major stages are described as follows:

1. A row address is placed on the DRAM's address lines, and $\overline{\text{RAS}}$ is asserted to signal the DRAM to latch the address.

2. After t_{RCD} cycles, the controller must now place the column address on the DRAM address lines and assert $\overline{\text{CAS}}$ to latch the address.

3. The DRAM module makes the requested data available after another t_{CAS} cycles. In this case, the module outputs two words of data to the controller.

4. After a minimum of t_{RAS} cycles after the first $\overline{\text{RAS}}$ has been issued, the controller can choose to issue a precharge command if it wishes to access another row, typically done by asserting both $\overline{\text{RAS}}$ and $\overline{\text{WE}}$ simultaneously. This command can overlap with the data output stage as shown, as long as the t_{RP} constraint holds.

5. t_{RP} cycles after issuing the precharge command, the memory module can then accept the next row address, and the process starts afresh.

These specified latencies typically depend upon both the target data rate of the device, and the speed grade of the individual part in use and are typically expressed as a number of clock cycles. Example figures for these latencies, along with the associated clock frequency can be found in Table 2.1.

	DDR3 Speed (MT/s)					
Parameter	800	1066	1333	1600	1866	2133
t_{CK} (ns)	2.5	1.875	1.5	1.25	1.07	0.938
t_{RCD} (cycles)	6	8	10	11	13	14
t_{CAS} (cycles)	6	8	10	11	13	14
t_{RP} (cycles)	6	8	10	11	13	14
t_{RAS} (cycles)	15	20	24	28	32	36

Table 2.1: Example values for t_{RCD}, t_{CAS}, t_{RP} and t_{RAS} for a variety of DDR3 devices. Adapted from [11].

Because of the latency required between selecting a row and being able to select a column, and the time taken to precharge the row again, many memory controllers attempt to keep the currently selected row open in the hope that the next access will hit the same row [4]. While this tends to yield good performance benefits for tasks which access sequential data, it further harms the predictability of DRAM; the timing of an access now also depends upon the addresses of the accesses which came before it. Moreover, a period of time must be waited after switching from a read to a write, or vice versa, due to the time it takes to reverse the direction of the data bus, further harming predictability.

The net result of these issues is that DRAM is difficult to predict compared with the simpler SRAM design since the latency of an access depends upon the previous accesses and whether the memory controller scheduled a refresh cycle at that point in time. Of course, it is possible to assume the worst-case timing on every single access (i.e. each access causes read/write switching and a precharge to take place), but this causes the efficiency of the memory subsystem to suffer (to around 40% efficiency on DDR2 devices [38] and falling further for DDR3 devices). This is, of course, unacceptable for modern systems. Much work has gone into improving the predictability of DDR systems while attempting to retain some of the efficiency, which will be explored throughout the remainder of this section.

2.2.1 Predictable DRAM Access

AMC

Paolieri et al. present a new memory controller called AMC (Analysable Memory Controller) [39]. It is designed such that each task can be independently analysed;

no task can affect the timing behaviour of another. It also utilises a pseudo-priority system where hard real-time tasks are serviced before any non hard real-time tasks. This then means that, while memory accesses cannot be preempted, each access for a hard real-time task has a known upper bound.

Another method they have used in order to improve the timing behaviour is to distribute the data across all banks. This has the effect that all RAM commands can be effectively pipelined, hence, a closed-page auto-recharge policy is also used, meaning there is a precharge issued after every access, each memory access will happen in a fixed time, however, the granularity of memory accesses must be quite large. Given four banks, each bursting eight 32-bit words, 128 bytes of memory must be read or written for every memory access. This does mean though, that if there are the correct number of banks in the memory system, the precharge command can have been issued and completed in time for the last bank outputting its data, so no cycles are lost waiting for a precharge.

Despite this, there are still some issues. Firstly, the worst case timing can differ based on the previous action; read to write and vice versa can introduce more delay. On top of this, in the four-bank system, there is still some delay between consecutive accesses in order to allow the memory to precharge and for the RAS and CAS to be re-issued. This issue could be alleviated by adding yet more banks, but increasing the bank count means even coarser memory granularity, so yet more wasted data, and subsequent accesses are waiting for yet longer for the previous access to complete.

The memory requests of each hard real-time task are then serviced in a round-robin fashion. In addition to this, each hard real-time task also has its own access queue. This means tasks cannot interfere with themselves, and equally well, a task issuing many accesses cannot interfere with a task only issuing a few accesses. To this extent, the worst case timing under this system is defined as follows, where N_{HRTs} is the number of hard real-time tasks in the system, and $t_{ILWORST}$ is the worst case time for a single access:

$$WCRT = N_{HRTs} \cdot t_{ILWORST} - 1$$

The rationale here is simply that the worst case time is, obviously, the worst case access time multiplied by the number of other HRT tasks. Typically, this would be $N_{HRTs} - 1$, as a system with one HRT task should have no interference, but a NHRT task may have started the cycle before, which cannot be preempted and hence must be considered. This also explains the -1 in the equation, which accounts for this clock cycle where there were no other outstanding requests from an HRT task, and hence an NHRT request was issued. Given this worst-case response time, a worst-case execution time for the whole task can then be ascertained.

There is another consideration, which is when the refresh cycles are issued in the system, which are issued every t_{REFI} cycles. In order to accurately predict how many refresh cycles will take place over the execution of the task, the authors propose starting each hard real-time task at the same time as a refresh cycle is initiated such that it is known that a refresh will definitely happen after another t_{REFI} cycles. If the task is initiated *just* after a refresh has been issued, it will therefore have to wait for the next refresh, and hence the final WCET for the task is as follows:

$$WCET_{TOTAL} = WCET + t_{REFI} - 1$$

This, in the general case, allows for full predictability for each memory access, although as the number of tasks in the system grows, so does the WCET. In a system with say, 100 nodes, all of which have HRT requirements, this will simply not scale, as the WCET bound will be huge. This work does, however, produce a good way of alleviating the cost of a precharge and the timing difficulties generally found with a closed-page system. Moreover, it also has an in-depth analysis of the timing issues found with a DRAM system.

Predictable Access Patterns

Akesson et al. have considered the generation of predictable memory access patterns [40, 41] which are then scheduled at run time in the memory controller [38, 42]. The first implementation of this technique, Predator [38] defines three memory access "groups", a memory read, a memory write, and a memory refresh. These then correspond directly to a sequence of SDRAM commands which have a known time bound. These groups of commands are then scheduled dynamically according to a CCSP arbiter [43], which will be explored further in Section 2.1.

Similar to the analysis done by Paolieri et al. [39], each access group (except for the refresh group) accesses an entire burst across all banks. Again, all accesses are accessed using auto-precharge, so the activity of one task cannot affect that of another. Again, they impose the constraint that there must be a minimum amount of cycles between accesses to the same bank (tRC cycles). In any case, they arrive at a useful figure which is, for a read and write group, the data efficiency, which is 82.6%, assuming all requests are aligned (i.e. they can be read in a single burst for each bank). They also arrive at the figure 60.3% for a system with a burst length of 4 (i.e. less data transferred), assumably as subsequent pipelined requests will need to wait longer for the tRC time constraint to be satisfied, however, as less is now read, the overall efficiency may be improved as less data is wasted, and thus, the CPU spends less time stalled receiving useless data.

Memory Access Scheduling

Rixner et al. [44] propose a method to schedule memory accesses, but without any regard for the timing predictability or WCET of the memory accesses. When applied to synthetic benchmarks, this scheme can show a memory bandwidth improvement of up to 144% compared with a naïve system (i.e. activate, read, precharge on every access).

Rixner's method adds a precharge manager and a row arbiter to every bank. For each access, these decide when the currently open row should be precharged and which row should be opened, respectively. There is then a column arbiter, which, for each request, decides which should be able to read, or rather, which can have access to the shared data bus. Finally, these connect to an address arbiter which chooses which access to actually schedule based on the current state of the memory system. Each of these arbiters can then use one of many different arbitration techniques, as follows:

IN-ORDER Classical DRAM - only service a request if it is the oldest one.

PRIORITY Each reference has a priority: service the highest priority one first. There are then sub-techniques, such as increasing the priority of old requests or loads being given higher priority than stores.

OPEN Only precharge if there are pending references to other rows in the bank and there are no references to the currently active row.

CLOSED Precharge rows with no more accesses pending. This is different to "open" in that open waits until another access arrives targeting a different bank.

MOST PENDING The row with the most accesses is serviced first. This is then combined with either "open" or "closed" to handle precharging. It is not clear if all accesses to that row should be flushed before switching to a new row, or switch to a new row just as another row has more pending accesses.

FEWEST PENDING Service the row with the fewest pending accesses. The theory is, by eliminating this, more accesses will arrive targeting a row which already has many accesses, thus increasing performance as the row will get more use before needing to be precharged.

These schemes were then evaluated against a number of benchmarks. Firstly, a comparison to "first ready" was made, where access is given to the oldest request which does not violate timing (allowing bank accesses to be interleaved) show an improvement of up to 125% in memory bandwidth compared with the naïve model, although real applications only showed 17% improvement (compared with 70% for microbenchmarks and 40% for application memory traces). More aggressive re-ordering (such as targeting accesses to the currently open row) showed

much greater improvements, 27-30% for applications, 106-144% for microbenchmarks and 85-93% for application traces, although of course, the optimal strategy is completely dependant upon the actual workload in use. Nethertheless, changing between such strategies at run time in a soft or non real-time system could yield some interesting results.

2.2.2 Summary

As can be seen, the complexity of accessing a region of DRAM is not as simple as with classic SRAM chips where an access consisted of placing the required address on the inputs and asserting the "read" line. This increased complexity ultimately means that the response time of a memory module can depend on the accesses that came before it; if the previous access targetted the same row as required, an access might be doubled in speed. Similarly, the need to periodically refresh the memory modules, rendering main memory inaccessible, adds further complexity to the response time analysis of memory.

This is simplified by the introduction of memory access patterns [40, 41]. Since the amount of data transferred per pattern is known (as the burst size is set in the controller), the system is now easier to analyse, as only the patterns for transfer need to be considered rather than every block fetched from memory. These patterns are then scheduled according to a credit-controlled arbiter, enforcing a bandwidth quota onto each task. This is also better than the approach suggested by Paolieri et al [39], since their approach uses a round-robin based arbiter, causing a much more pessimistic estimate of WCET, especially as the number of cores is increased.

Moreover, the predictable patterns will restore the RAM to a known state after an access has taken place. This also does not have such a large overhead due to the patterns used. Since a memory access *must* access all banks (due to the burst size), it is possible to begin precharging a bank while a later bank is being read from. If long enough bursts are used, this overhead is effectively eliminated.

The downside of this approach is that it also enforces constraints on the rest of the system. The approach always reads whole bursts out of every DDR bank in the system for efficiency reasons. Of course, if the rest of the data is not required by the running task, this creates significant overheads. Moreover, this imposes restrictions on the minimum transfer size over the interconnect, and in the case of a cache, a minimum line size (else the additional data can overwrite potentially useful data in other cache lines). The additional data *can* be thrown away and ignored, but this reduces potential performance and wastes cycles while handling this data.

As with many other methods though, a combination of methods may be advantageous for a mixed HRT/SRT/NRT system. As shown by Rixner et al [44],

memory accesses can be scheduled based upon the previous access stream. This then allows the memory controller to exploit aspects of the previous stream, such as bypassing the precharge and RAS stage if accesses lie on the same row. Of course, this can severely damage real-time analysis and WCET bounds, but for soft and non real-time tasks, a great speedup can be observed. It may therefore be possible to use such a method for these soft real-time tasks, while using the predictable arbiter for hard real-time tasks. This not only allows for potentially faster SRT memory accesses, but simplifies analysis of the higher priority HRT tasks.

2.3 THE MOVE TO MULTI-CORE

Due to the breakdown of Dennard scaling, it is difficult to further increase the clock speed of processors. Moreover, as feature sizes become smaller, as do the interconnecting wires. Since the resistance of a wire is inversely proportional to its size, and the capacitance proportional to its length, the capacitor time constant $T = RC$ begins to dictate the maximum wire length as wires are made thinner, and hence the area of the chip which is reachable in a single clock cycle [45] which quickly makes large circuits at a high clock speed infeasible.

Instead, system designers are looking towards utilising multiple, smaller cores in order to meet the demand for processing performance. This allows system designers to utilise these extra transistors, as predicted by Moore's Law, and to maintain the typical performance scaling trends without significantly increasing the clock speed of the processor.

Of course, processors need to be able to communicate with both peripherals and main memory. On embedded systems, this communication is normally failitated using a shared bus such as AHB [46]. Classical shared buses such as these suffer when scaled further than a few cores; because a processor locks the whole bus during an access, other processors cannot initiate a request, even if the destination of the request is a completely different processor or peripheral. Moreover, there are still problems with long wire lengths since the bus must visit each processor and peripheral.

Advancements to this include crossbar interconnects. These instead connect every processor to every other processor or peripheral through a set of switchboxes and using dedicated links instead of utilising a shared bus (for example, AXI [47]). This provides great improvements to perfomance, as multiple transactions can now take place simultaneously, but of course, large switches are required in order to provide this interconnection. While crossbars may also solve the problems of long wires found with buses, they can incur a large logic overhead to implement these switches which may damage the maximum possible clock frequency.

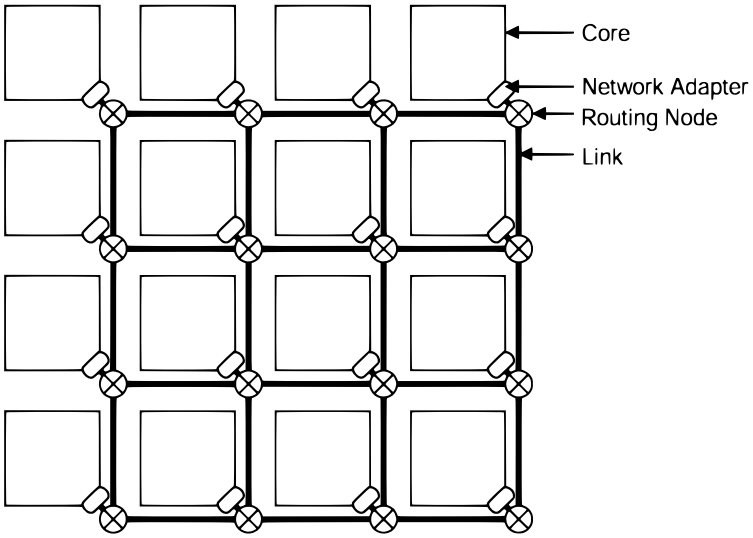


Figure 2.5: An example of a Manhattan-grid based network-on-chip [6].

These problems with scaling have led to many system designers moving to network-on-chip based approaches [48]. These instead attach each processor to a small network router, then each message is encapsulated into a network packet and routed over the network based upon some routing scheme. A common example of such a network can be seen in Figure 2.5. Here, packets can be routed in any direction at each router to their destination. Commonly, they will be routed in an X-Y fashion, where they will first be routed to the correct column within the network, then either up or down the column to their target.

There are numerous examples of using this form of interconnect in the literature [6, 49, 50]. Within the context of scheduling memory accesses, this research is largely orthogonal; it focuses on the many-to-many communication paradigm and, in cases, enforcing bandwidth guarantees on the communication between cores in the system [50]. Attempts to schedule an 8-core system Æthereal based system with inter-process communication and memory access in a realatively large FPGA failed due to the amount of buffers required at each router and due to the size of the TDM schedules required [51].

Many current architectures make this distinction between memory traffic and inter-process communication traffic. The Parallella Epiphany multi-core does not itself integrate a memory controller, but provides three different communication networks; one for inter-process read, one for inter-process write and one for off-chip traffic (e.g. to a memory controller) [52]. The Tilera line of multi-core processor contain many different communication networks, including a network for I/O devices, a network for memory accesses and a network for inter-process communicaion [53]. Intel's Knights Corner multi-processors only contain a single ring network using QPI [54] as can be seen in Figure 2.6, but these devices only support the shared memory paradigm and do not expose core-to-core communications to the programmer, hence only a memory network is provided.

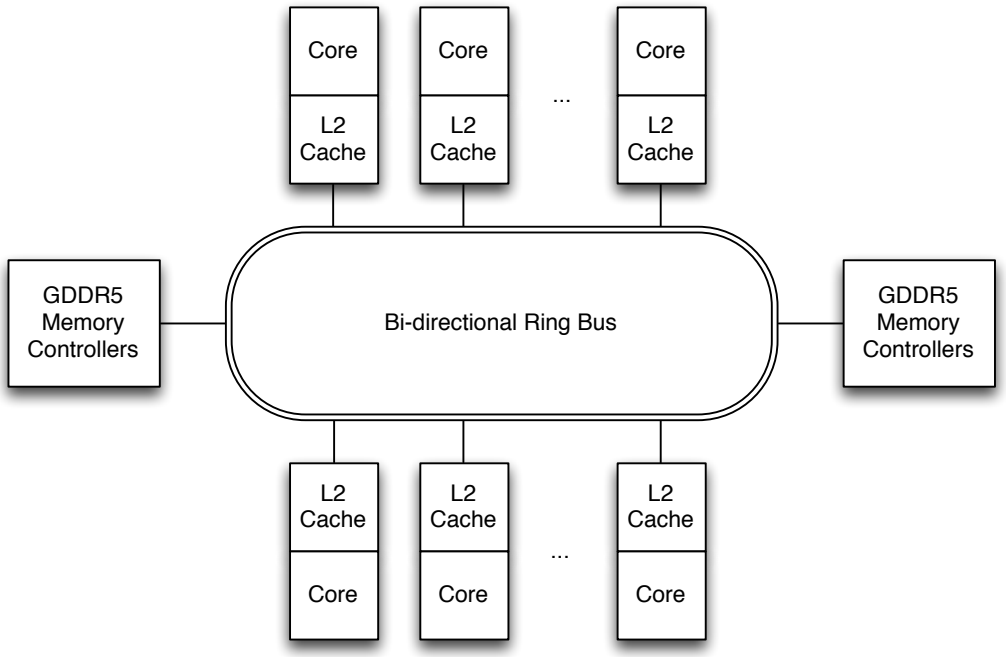


Figure 2.6: Example block diagram of Intel's Knight's Corner platform. Adapted from [7].

None of these multi-core processors contains any arbitration over the interconnection network and hence are difficult to predict in terms of timing behaviour, and instead would have to utilise an arbitration scheme, as will be discussed in Section 2.3.1. As an example, the Predator DRAM controller [38] instead connects directly to the virtual channels of an Æthereal router, then performs memory arbitration from there.

2.3.1 Memory Arbitration

As discussed previously, while many commercial and research multi-core systems utilise a network-on-chip to connect their processors to shared memory, the access across the network is rarely arbitrated and hence, it is difficult to obtain a worst-case response time for a memory access. In systems where this interconnect *is* arbitrated (e.g. Æthereal), the disconnect between the communication requirements of a task and the memory requirements makes the arbitration scheme of the network unsuitable for rate limiting access to shared memory.

For this reason, many multi-core systems also utilise some form of arbitration at the memory controller in order to provide timing guarantees to the tasks within the system. These arbiters are responsible for rate limiting the accesses to shared memory according to some arbitration scheme. Given a predictable form of arbiration, and a predictable memory controller (e.g. from Section 2.2), an upper bound can then be ascertained for a memory transaction.

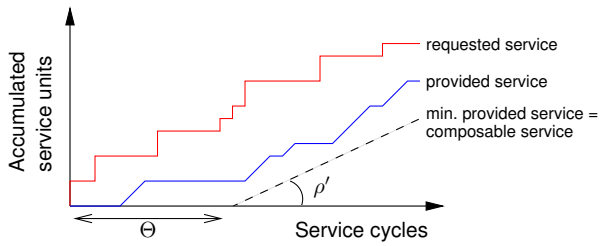


Figure 2.7: Graphical representation of the service provided by an $\mathcal{LR}$ arbiter [8].

Many of these arbiters fall into the class of *Latency-Rate* ($\mathcal{LR}$) arbiters [55]. An arbiter in this class can be characterised by two things: it's latency, or the maximal waiting time in which a set of requests may be blocked, and its rate, or the average rate at which service will be provided to a shared resource. A graphical model of this can be seen in Figure 2.7. Here, the red line denotes the service requested through the arbiter, and the blue line the provided service. The arbiter will provide an average rate of service ρ after a delay Θ. The system is then schedulable assuming that the sum of all rates does not sum to more than the available bandwidth, i.e. $\sum_{r \in R} \rho_r <= 1$, where R is the set of all requesters. In this case, the provided service will never fall below the minimum service level (the dotted line) while there is still some work outstanding.

Being able to derive the latency and rate of a memory transaction in this way is good for system analysis. Without such an arbiter, the system as a whole must be analysed at once, and the behaviour of each task with regards to shared memory must be accounted for together which is difficult and time consuming. By splitting the available system bandwidth in this way, each task only needs to be analysed against its assigned partition of the bandwidth, hence simplifying the analysis. Furthermore, this simplifies task switching; given each task has assigned bandwidth bounds, a new task can be scheduled on a processor as long as the amount of bandwidth it requires does not exceed the remaining "spare" bandwidth in the system. The combination of tasks with their own partitions like this is typically called "composable" system construction [56].

Of course, the behaviour of an arbiter depends upon the arbitration scheme in use. Some of the most commonly used schemes are detailed throughout the remainder of this section.

Static Priority

The most basic arbitration scheme to use is static priority. Within this scheme, each input port on the arbiter is assigned a static priority, and on each scheduling interval (i.e. the period of time at which the arbiter will schedule a new request), the request which originated from the highest priority input port is given service. If the set of running tasks is known ahead of time, along with the exact memory

locations they will access, when they will access them and exactly how long each access will take, this approach is one of the simplest to analyse and does not require much overhead in hardware to keep track of state.

Of course though, static priority does not allow for any dynamism in the system; it is difficult to change the set of running tasks in a safe manner (unless all possible sets of running tasks have been analysed ahead of time), and nor can static priority deal with much deviation from what was expected; if a memory access could not take place when the analysed schedule assumed that it could, then it may cause the schedule to shift by one or more time periods and hence may cause tasks to miss their deadlines. Moreover, static priority is not safe; if a task initiates more memory accesses to what was expected, either through a bug in the task or an error in the system analysis, it can negatively affect other tasks and again cause them to miss their deadlines. In the worst case, it is possible for a task in a static-priority system to flood the memory controller with reqeusts and effectively starve all other tasks.

It is possible for statically scheduled systems to undergo "mode changes" [57] in order to change the priorities of tasks or the set of running tasks to add some form of dynamism to the system. Ultimately though, this simply assigns a set of tasks to a "mode", then analyses the interactions of each, allowing the system to jump between a few pre-determined set of tasks.

This inability to deal with unexpected conditions makes static priority a poor fit for a system which also uses prefetching; if the prefetcher initiates a prefetch at a poor time, it may cause excess interference in the system and cause tasks to miss their deadlines, as the system analysis did not take this into account. Even if the prefetch is safe, it may displace useful data out of the processor's cache and again, cause it to perform an unexpected fetch, causing excess interference and cause tasks to miss their deadlines. While not good on its own, static priority can be very good when combined with other arbitration schemes; both CCSP and FBSP use static priority to determine an ordering when there are multiple requests which can be scheduled in the same scheduling interval. This, combined with another arbitration scheme, allows some flexibility within the system as high-priority requesters can issue a request without experiencing much latency, whereas requesters which can deal with additional latency can be assigned a lower static priority while still guaranteeing an upper bound on the blocking they will experience.

Round-Robin/Time Division Multiplexing

Arguably the most basic *fair* arbitration scheme is round-robin arbitration [58]. This scheme simply cycles through the input ports to the arbiter in sequence, accessing the request from the next slot on each scheduling interval. Because of the simplicity of this arbiter, it is trivial to derive latency-rate constraints for it; as-

suming a scheduling interval of t_{sched} and N inputs ports, the latency is the time taken if a request has "just missed" its interval, that is, the time taken to elapse the window it just missed, and the windows of all other requestors, hence can be derived as $(t_{sched} \times N) - 1$. The rate is equally simple to derive, since bandwidth is equally shared between requestors, and is hence $\frac{1}{N}$.

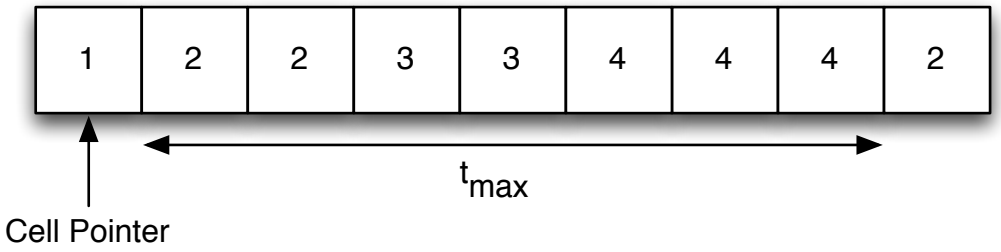


Figure 2.8: Example 9-slot TDM schedule for four requestors.

Of course, round-robin assumes that all requestors will require the same rate of service. This is relaxed when using time-division multiplexing instead. This method uses a table, where each slot denotes which requestor is given service on each cycle, of which an example can be seen in Figure 2.8. On each scheduling interval, the arbiter reads which requestor should be given service from the current table cell, then increments the cell pointer. This can then be used to accomodate the case where uniform bandwidth is not required, but some requestors may require a slightly high bandwidth share than others.

While slightly more complicated, this arbiter does still fit into the framework for an $\mathcal{LR}$ arbiter. Here, it must be assumed that the request just missed its slot again, and that it will have to wait the maximal amount of time possible until it can be scheduled again. This behaviour depends upon the current TDM schedule; for the schedule in Figure 2.8, the maximal waiting for requestor number 2 is t_{max}, or in this case, $(7 \times t_{sched}) - 1$. The average rate is then the proportion of slots which a requestor has from the whole schedule, or rather, $\frac{nSlots}{N}$.

Frame-Based Static Priority

So far, all of the arbitration schemes suffer in the fact that they do not allow for any jitter in the release of requests coming from a requestor; if they miss the start of their window by a cycle, they must wait the maximal delay for their time slot to appear again. Frame-based static priority attempts to fix this by moving to a model whereby a requestor is permitted to make a number of requests within a given time window [59]. Of course, there still needs to be a mechanism to arbitrate which requestor in the set of requestors which *can* make a request can have service, which is done through static priority.

At least for memory arbitration, this is advantageous over systems such as TDM because it does not require the task to always have the same access pattern; it may be "burstier" over some windows, or may be constant over others. Instead, it only enforces an average rate. On the other hand, it may cause a great amount of slack to accumulate for low-priority requestors; if the high-priority requestors are not utilising their entire bound, lower-priority requestors will be able to gain service much faster than their estimated latency. Moreover, the hardware overhead of such an arbiter is larger than the table-based systems.

The latency and rate of the arbiter can be derived as follows: the latency is the worst-case situation where all higher priority requestors must consume their entire bound. If n_i is the number of requests that a requestor $i \in R$ may have serviced over the frame interval t_{frame} and $hp(i)$ is the set of requesters with a higher priority than i, the worst-case latency Θ_i is defined as follows:

$$\Theta_i = \sum_{k \in hp(i)} n_k \times t_{sched}$$

The rate is then the proportion of the total number of requests which can be serviced in the frame time, hence:

$$\rho_i = \frac{n_i \times t_{sched}}{t_{frame}}$$

Credit-Controlled Static Priority

While Frame-Based Static Priority goes some way to allow for non-uniform access patterns, it still imposes scheduling within a given frame time. Credit-Controlled Static Priority (CCSP) instead attempts to emulate the the characteristics of an $\mathcal{LR}$ server, rather than fitting an $\mathcal{LR}$ server model to the arbiter [9].

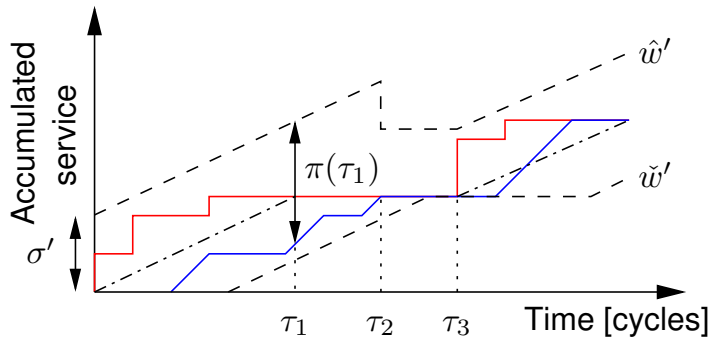


Figure 2.9: Example of accounting when using a CCSP arbiter [9].

In effect, this model creates an upper bound on the service that a requestor can receive in a period of time, based upon the latency parameter in an $\mathcal{LR}$ server,

ρ. In order to account for the request size of a requestor, this upper bound is initially a "burstiness" offset, σ, and increases while the requestor is "active" on every cycle by ρ. The difference between this upper bound and the service that has been provided is then the potential, where a requestor can be scheduled if it has a non-zero potential. These concepts can be seen in Figure 2.9; the red line is the requested service, the blue line the provided service, $\hat{\omega}'$ the upper bound on provided service and π the potential.

Given this rate limiting, assuming that the system is schedulable (i.e. $\sum_{r \in R} \rho_r <= 1$), each requestor can receive a minimum level of service $\check{\omega}'$, each after a latency Θ_r. Of course, CCSP assumes that it is possible to perform continuous accounting and that requests are pre-emptable. In the case of shared memory, memory transactions are non-preemptable, in which case the arbiter must wait until the requestor has sufficient potential to encapsulate the worst-case timing of the memory transaction before it is permitted to be scheduled. Again, similar to the case of FBSP, many requstors may become "valid" at any one time. In this case, a requestor from this set is selected by means of static priority arbitration.

Work Conservation

One issue with most arbitration schemes is that they enforce that the requestors connected to them should be requesting data through the arbiter at a given rate. While some tasks may fit into this model, many do not; they may request large amounts of data at some points during their execution, or sit idle at other points. Moreover, if the system is being dynamically scheduled, the total amount of available system bandwidth may not be assigned to tasks within the system. In these cases, the memory controller would normally be sat idle. This is not good; an idle memory controller is *always* useless, instead it *could* be trying to do something useful.

This situation is the motivation behind "work-conservation". In effect, if the arbiter cannot schedule anything because all tasks have exceeded their bound, then it will instead pick some work from one of the tasks which has exceeded its bound. As an example, both CCSP and FBSP can just pick the highest priority requestor which has some work outstanding if not requestors with some remaining potential have any work to do. Round-robin and TDM can look ahead in the slot table for the next requestor with work to do (note that round-robin and TDM *must not* move the slot table pointer; if a requestor is relying upon the fact that it will be serviced every x cycles, and hence only generates a memory request every x cycles, moving the slot table pointer may skip the requestor and hence reduce the actual allocated bandwidth).

While work-conservation can be used to improve the average-case timing for a task, it cannot be used to improve the worst-case. Many work-conservation schemes are unpredictable, althouh even if they were predictable, the behaviour

of tasks under work-conservation depends on the behaviour of all other tasks (in order to be able to ascertain whether a task will miss its slot) and hence breaks one of the main reasons for performing bandwidth partitioning.

2.3.2 Distributed Memory Arbitration

One problem that has been identified when using an arbiter to control access to memory is that a large, monolithic arbiter like is typically used with the above schemes cannot scale to the number of requestors in a modern system. If the arbiter is connected to each processor in the place of a memory bus, the system will still have long wires to connect many processors which are typically spanning the die to a centralised memory arbiter. Connecting the memory arbiter to a NoC can solve this, since there is only one entry point, but since each processor, or worse, each task requires its own bandwidth bounds, this requires the memory controller to separate these requests into separate input queues (through a large de-multiplexer), then consider each of these input queues. While this (potentially) solves the issue of long wires, the logic overhead caused by attempting to arbitrate so many input queues leads to extremely long logic delays, and hence a low clock speed in the arbiter.

As outlined in Section 2.3, many systems split out the memory requirements of a task and the processor-to-processor communication requirements. Instead of attempting to route packets to the memory controller then arbitrate from these, state-of-the-art approaches instead take inspiration from network-on-chip systems used for inter-process communication and attempt to apply them to memory arbitration. By splitting this arbitration over a specialised memory network, these distributed approaches can scale to many cores, while still enforcing bandwidth guarantees required by real-time systems.

Distributed TDM

The first of these is a distributed TDM implementation by Schoeberl et al [10]. This uses a set of network interfaces, which because memory traffic is a many-to-one operation rather than a many-to-many, are then connected to a set of two-into-one multiplexers which are connected together in a tree structure. These multiplexers are bi-directional; they select one of their inputs which has data on it to relay the request towards memory, then also relay requested data back to the processors again. A conceptual view of this can be seen in Figure 2.10; here, the nodes labelled *NI* are the network interfaces, and the *MI* node maps memory requests onto the memory controller.

Since the worst-case response time of the memory controller can be ascertained and bounded (using the techniques from Section 2.2.1), a TDM schedule can be derived such that two requestors can never request in the same instant, and that a

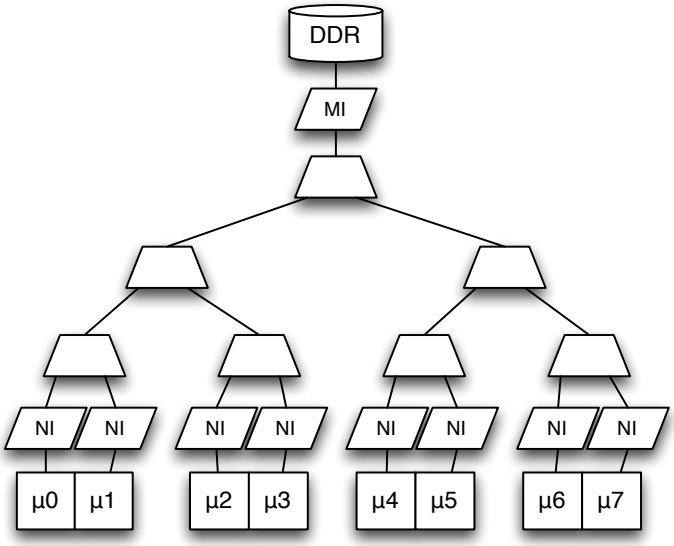


Figure 2.10: Conceptual view of Distributed TDM [10]

memory request will never be blocked at the top of the tree. This TDM schedule is then enforced within the network interfaces next to the processors. For this reason, the multiplexers do not require any intelligence whatsoever; they can just map whichever input has data onto the output since the TDM schedule ensures that the network is collision-free.

This improves the behaviour of a standard monolithic arbiter as now the distribution of the arbiter can span the whole device, rather than just being located in a section of the device close to the memory controller. This solves the wire length issue, and since each multiplexer is two-into-one and has no internal intelligence, the logic overhead of this approach is relatively low. On the other hand, in order for this approach to work, all of the network interfaces need to be clock-synchronised, since this approach effectively replicates the TDM slot table into each NI, and each NI must move through the slot table at the same time. Moreover, this approach cannot support work-conservation since the arbiters have no way to communicate and hence can never reach agreement on which requestor should be able to dispatch a request in the work-conserving case.

GSMT

GSMT (Generic, Scalable and globally arbitrated Memory Tree) [12] is another distributed arbitration scheme which performs some admittance control at the bottom of the tree, next to processors, then uses a tree of multiplexers to connect to memory. In contrast with Distributed TDM, however, the admittance control is performed by both the network interfaces *and* the multiplexer tree. Conceptually, the layout of the system is identical to that for Distributed TDM, as seen in Figure 2.10.

The initial admittance control is performed by the network interfaces. In effect, these are credit-based arbiters and can be used to implement many different arbitration schemes. These network interfaces contain a credit counter with an initial value InCr. On every active period, this credit counter is incremented by an amount (Nr), and decremented by an amount every time the requestor has been scheduled (Dr). In order to support frame-based arbitration schemes, this credit counter is replenished to a given amount when an entire frame has elapsed. Simply enough, if the current amount of credits that a requestor has lies between an upper and lower bound, the requestor can be scheduled. This scheme has been shown to work across many different schemes, where the authors give examples for TDM, FBSP and CCSP.

Of course, for FBSP and CCSP there many be many potential requestors within each scheduling interval. This is handled by assigning each requestor a static priority, as is typical with these schemes. On each interval, each requestor which can be scheduled (i.e. the credit counter falls between the upper and lower bound) sends its request onto the tree along with its priority. Each multiplexer then chooses the request with the highest priority, discarding the other request. The memory interface then receives the request with the highest priority, and relays a message back to this network interface to notify it that its request was scheduled, while all other requestors should retry in the next scheduling interval.

This priority scheme also allows the arbitration system to implement work conservation. This is done by biasing the priority of a work-conserving access such that its priority is lower than any non-work-conserving access. If the requestor's credit counter does not fall between the given bounds, then it can still send a request, just with this lower priority, and the rest of the arbitration scheme works as usual.

Compared with Distributed TDM, this approach is more flexible, but as each multiplexer must now inspect each request for its priority, its overhead is greater and hence the potential maximum frequency is lower. Moreover, the network interfaces themselves must be larger and slower to implement the counters. This approach still also suffers from potential issues with clock synchronisation, as request intervals all must start at the same instance, but also, each multiplexer must be synchronised to ensure that requests from sub-trees all move up the tree at the same time.

Bluetree

Bluetree [60, 61] is another distributed approach which uses a tree of multiplexers. This differs from distributed TDM however in that it does not perform any admission control at the bottom of the tree, next to the processors. Instead, each multiplexer has intelligence and separately performs its own arbitration. Moreover,

while Distributed TDM and GSMT rely upon some notion of a service interval, Bluetree is a fully demand-based system.

This demand-based scheduling is implemented by including buffers at the input to each multiplexer, which include some flow-control to the multiplexers located below them. Whenever the multiplexer further up the tree has space in its input buffer, the multiplexer picks a request from one of its inputs based upon a static priority scheme. By convention, the left-hand input has priority over the right-hand side of the multiplexer. Responses from the memory controller are then routed back to the requestors by the multiplexers.

In order for the multiplexer to be timing predictable, it contains a "blocking counter". This records how many packets have been relayed by the high-priority side of the multiplexer while a packet has been waiting at the low-priority side. When this becomes equal to a predefined constant m, a single request is admitted from the low-priority side, then the counter reset. Because a low-priority packet is only blocked when there is a high-priority packet pending, then this arbitration scheme is work-conserving, since the low-priority packet could be relayed otherwise.

As an example, if this counter is set as $m = 3$, then when a request arrives at the low-priority side of a multiplexer, then the maximal waiting is the time taken for three high-priority requests to cross the multiplexer before it may itself cross. Of course, for a packet arriving on the high-priority side, it may have to wait for a single low-priority packet to cross the multiplexer, assuming that the request has been preceeded by three other requests in quick succession.

Compared with Distributed TDM and GSMT, this approach does not require full clock synchronisation across the tree, however, the system analysis is much more complex and leads to a higher worst-case execution time than the other schemes, and the logic overhead at each multiplexer is higher than other approaches due to the blocking counters. As the network inherently supports requests being blocked at each stage in the tree though, this approach allows for much simpler pipelining of memory requests, and since there is no concept of a service interval, this approach allows for potentially much higher bandwidth than both Distributed TDM and GSMT.

2.3.3 Summary

After the demise of Dennard Scaling, system designers have used multi-core approaches in order to provided the expected year-on-year performance scaling that has been apparent over the last few decades. While this parallelism provides a good way to scale the performance of a system, it provides issues to the designers of real-time systems; the analysis of a running set of tasks now needs to not only

analyse the behaviour of the task under inspection, but also of all other running tasks.

Typically, this issue is sidestepped by providing a hardware arbiter which enforces an upper bound on how many requests a single processor may issue within a given period of time, effectively partitioning the available memory bandwidth across a set of processors. A task can then be analysed within the parameters of this partition, rather than attempting to analyse the timing behaviour of it with all other running tasks. This both simplifies the analysis, and is safer since a runaway task will be throttled by the arbiter.

Traditionally, this arbiter is implemented as a single, monolithic block with many inputs (from the processors) and a single output (to the memory controller). As the number of processors within the system grows, it becomes fast impossible to scale the arbiter to the required number of processors. This restriction is imposed by the fact that these arbiters typically select a requestor from a set of input buffers, then issues this request to memory. In order to do this lookup in a single cycle, huge multiplexers are required, and the logic overhead soon becomes massive. Moreover, as the size of the arbiter grows, as does the internal wire length, which cannot grow much further since the capacitance of the wire (which increases with length) soon limits the switching frequency, and hence the maximum clock speed of the arbiter.

	Area (mm^2)			f_{max} (MHz)		
# Clients	TDM	CCSP	GSMT	TDM	CCSP	GSMT
4	0.016	0.020	0.017	588	526	1250
8	0.029	0.036	0.035	500	435	1250
16	0.061	0.077	0.070	435	357	1250
32	0.107	0.172	0.141	333	333	1250
64	0.203	0.417	0.282	333	303	1250

Table 2.2: Area and maximum frequency for a distributed arbiter (GSMT) versus monolithic TDM and CCSP [12].

To solve this problem, much of the state-of-the-art research in system arbitration is turning to using distributed arbitration schemes. These typically replicate and distribute parts of the arbiter out to all requestors, then use a set of multiplexers to route the packet to memory in a pipeline. Because the multiplexers are simple two-to-one multiplexers, which are then pipelined, the logic overhead is typically less than for a monolithic arbiter, and the shorter wire length and pipelining allows for a much greater maximum frequency which, more importantly, is not significantly impacted by the number of requestors. As an example, the results of scaling GSMT versus monolithic TDM and CCSP are shown in Table 2.2.

The usage of this arbiter then allows the worst-case response time of a memory request to be bounded. Moreover, as each task uses a partition of the system, it is trivial to assert that the system is schedulable; if the sum of the bandwidths of

all tasks is not greater than the available system bandwidth, then the system is schedulable. The assertion that the tasks cannot cause other tasks to miss their deadlines by the use of a hardware arbiter then allows the system to be analysed and a worst-case execution time estimate to be ascertained for each task, as detailed in Section 2.1.

2.4 PREFETCHING

A useful technique to use to attempt to reduce the latencies associated with accessing main memory is prefetching. This is a technique which fundamentally attempts to speculatively issue memory requests for data which will be required by a processor in the near future. Given the rising memory delays outlined in Section 2.2, made worse by needing to ensure fair access to this main memory from multiple processors as outlined in Section 2.3, something needs to be done to attempt to reduce the overheads brought about by accessing shared memory.

There are two main methods by which prefetching can be implemented. The first of these is pure hardware prefetching, where the processor contains a prefetch unit which can speculatively issue memory accesses on behalf of the processor. Hardware prefetches are almost ubiquitous in high performance processors; the POWER4 architecture [62] contains both instruction side and data side stream prefetches, and many Intel processors contain both stream and stride prefetchers [63]. While hardware prefetching is effectively transparent to the system, it is not without its limitations. Because the prefetcher can typically only observe what addresses are being fetched, it cannot usually prefetch addresses which do not follow an obvious pattern (for example, dynamically allocated linked lists), or simply non-linear access to an array.

Instead, software prefetching [64] can be used to predict such data access patterns. This is a technique which adds a "prefetch" instruction to a processor's instruction set which effectively acts as an asynchronous load. While this can improve the prefetching accuracy of a task which fetches more "random" data, it does have the downside that prefetching is now managed by the programmer. The programmer must now decide when to issue a prefetch in order for it to arrive in a timely manner, but not too early, and for what data. While this decision may be easily possible for single-core machines, it becomes much more complex on multi-core systems where the optimum time to initiate a prefetch, or how far ahead of the stream the prefetch should be issued now depends on the current memory load, as the prefetch may be blocked by other cores requesting data from memory. Modern compilers can also automatically insert prefetch instructions on behalf of the programmer [65], but of course, the compiler must still decide how early a prefetch should be initiated in order to be optimal. Moreover, software

prefetching and hardware prefetching are not mutually exclusive; the combination of the two can still yield performance improvements where both are used for different data patterns, or the software prefetcher can be used to train the hardware prefetcher [65].

While software prefetching can be more accurate (due to the executing program having better knowledge of the data it will soon require), it does have its pitfalls. Inserting such instructions into the program stream of course increases the overall size of the program, which may be problematic for embedded systems. Moreover, it is difficult to place these instructions in an optimal fashion; they must appear early enough in the program such that the data to be prefetched has been fetched by the time it is required by the program, but not too early or too late. On single-core systems, the "optimal" locations for these may be determined through profiling or in-depth knowledge of the architecture, but of course on a multi-core system, the latency of a prefetch may depend upon the behaviour of all other cores in the system, and hence placing these instructions in an "optimal" place may be difficult for multi-core systems.

Applying software prefetching to a real-time system also brings about its own set of issues. Firstly, the inclusion of these prefetch instructions fundamentally changes the timing behaviour of the task, and will require it to fetch more data from main memory, causing more cache misses. Equally, this increase in task size may also cause the task to no longer fit in cache, again causing yet more cache misses. As an example, Lee et al. [65] demonstrate that the code size of some benchmarks increase by 50-100% when software prefetch is used. This overhead may increase both the worst-case execution time of the task (especially if assuming that all prefetches miss) and may increase the complexity of the system analysis. Finally, assuming that the prefetcher is a shared resource, the interference caused by all tasks communicating with the prefetcher must be factored into the system analysis to see if a software prefetch will even reach the prefetcher by the time that the task requires the prefetched data.

Because of this additional complexity of using software prefetching in a real-time multi-core system, this thesis only concerns itself with hardware prefetching methods for the time being. Even within pure hardware prefetching, there are numerous different prefetching schemes which are detailed throught the remainder of this section.

Stream Buffers

The simplest approach uses the concept of sequential streams of data, as proposed by Jouppi et al. [66]. Simply enough, this makes the assumption that if the processor has previously required blocks A, $A+1$ and $A+2$ from memory, then it will probably soon require block $A+3$. If that block was deemed as useful, then block $A+4$ can also be prefetched. This is implemented by utilising a small table inside

the prefetcher which encodes the state of n access streams. When the processor makes a memory request, if the address requested is the next in any of the stored streams, then the stream information is updated and optionally, a prefetch initiated, otherwise the access overwrites an existing stream using some update policy (e.g. least recently used or round-robin).

The cache then records which blocks are stored as the result of a prefetch. If the processor requests a cache block which has been prefetched, then the cache notifies the prefetcher of this event. Using the stream information, the next line in the stream can then be prefetched, and the information updated. The performance of this technique depends upon the number of stream buffers employed. Using a single stream buffer (i.e. the prefetcher can only identify one stream at once), a performance increase of 7% was observed; with four, this figure increased to 60%.

Stride Prefetching

While the code for a task is typically accessed in a serial fashion, the data is not always in such a predictable scheme. Tasks may not access every block in sequence, but may access with a stride. Stride prefetching [67] attempts to extend stream prefetching to discover cases where the processor is predictably accessing the pattern of A, $A + d$, $A + 2d$ etc. It does this by using a similar table-based scheme, but indexing the table on the program counter of the instruction which issued the load. On each load, the instruction's entry can be recalled, and the current load address compared to the last one to try and discover a pattern. If one can be reliably ascertained (e.g. the prefetcher has observed more than m accesses in this pattern), then the prefetcher can begin prefetching data as before.

While this can detect more diverse patterns than stream-buffer based techniques, it does suffer from the fact that the prediction tables need to be much larger than before since the tables must be indexed on the program counter rather than the last load address. While these will be good at detecting a stride in say, a loop with a single load instruction within, they may suffer when there are multiple load instructions, each of which load from sequential memory addresses (e.g. if the previously described loop was unrolled). Fu et al showed that program counter based approaches such as these require a table of around 256 entries in order to be effective [68].

Markov Prediction

Of course, both stream and stride prefetching rely upon a constant difference between load addresses to be observed. While this may be the case for code which steps through the elements of an array, or for serial instructions, it is not always the case. Joseph and Grunwald [69] attempt to prefetch these other cases by using Markov predictors. This attempts to encode a graph of accessed memory locations where the nodes are memory addresses and the edges encode the "next" address to

be fetched, with a probability. When a memory access is observed, the prefetcher can load this information and prefetch the most likely child nodes.

Of course, this graph will likely be huge and hence difficult to traverse quickly. Nesbit et al. propose a generalisation of this which instead encodes the *difference* between the memory addresses [70], where each node in the graph is the difference between the last memory fetch and the current one, and the arcs are again the probability. This can provide similar performance to a Markov prefetcher at a fraction of the space required, albeit with potentially more false prefetches being issued. Such an approach can be good for loops within which the body accesses data in a structured, but not constant fashion (e.g. accessing elements in a structure out-of-order).

Pointer Chasing

Another method to handle the prefetching of seemingly random data is to annotate the data to be fetched with a set of pointers to the next data in the chain [71]. The primary motivation of this is to prefetch linked data structures, such as linked lists, which may also contain pointers to a separate data payload. Because this data is annotated by either the programmer or the compiler, this does not actually require any prediction and hence is much more accurate than Markov-based approaches. After these pointers have been identifed, a prefetcher can then use them to fetch the "next" data items into target cache, ready, and also any supplimental data associated with each data item (i.e. data pointed to from a generic linked list implementation).

Another approach attempts to identify such pointers automatically based upon the observed *data* stream. Cooksey et al. utilise a method which insepcts data returned from memory then, if any of it *looks like* a pointer, it can traverse down that pointer and fetch additional data [72]. If any of that also could be a pointer, the process can be repeated. In many cases, the heuristic for "looks like a pointer" typically inspects the data returned from memory and if it could be a memory address which falls in the heap segment of a program, then the technique assumes it is probably a pointer to dynamically allocated data. Of course, this technique can provide a great deal of false positives. There is also work on training this technique based upon whether the processor accessed the prefetched data, and how many prefetched blocks must be accessed for the prefetcher to restart prefetching data.

A final approach again attempts to detect pointer loads by inspecting returned memory contents, but instead stores the mapping between the location of the pointer and the data it points to in a cache [73]. The advantages of this are twofold; firstly, on a repeated load of the pointer, the data pointed to can be loaded directly from the pointer cache instead of main memory. In addition, if a pointer has been retrieved from this cache, the prefetcher can start loading blocks from the pointed to memory location. This potentially has less pollution than Cooksey's method,

but again assumes that pointers can be accurately identified. In addition, it is unsuitable for multi-core systems, as while the local core can update the pointer cache if a pointer is changed, multi-core systems would require some form of snooping to implement this, adding to the overhead.

2.4.1 Adaptive Techniques

Prefetchers can typically be tuned on two main critera, the *distance* and the *degree*. The prefetch distance is used to set how far ahead of the current stream the prefetcher will start fetching from. Incrementing this value is useful in situations where the prefetcher is blocked by standard memory traffic from processors. If the prefetcher is blocked, it will not be able to initate a prefetch in a timely fashion and hence the prefetch may be late; in the best case, it will only be able to mask a portion of the memory latency, and in the worst case will not be able to be initiated at all.

The prefetch degree controls how much data is fetched at once. As briefly mentioned in Section 2.2, requesting successive memory locations is much, much cheaper than random access, hence the performance penalty to request successive memory blocks in a prefetch is extremely small. Controlling this parameter is something of a tradeoff; fetching more data at a time may cause more, useful data to be fetched in fewer prefetches. If this extra data was not required however, it can cause issues, for example, it may displace useful information from cache and hence actually worsen the performance of a task.

Of course, the actual set of parameters to use depends entirely upon the task being executed and the current state of the system; a task which fetches huge arrays will benefit from a larger prefetch degree than one which only fetches a few successive memory locations. Additionally, if the current load on the memory controller is high, a higher prefetch distance will probably yield much better results than a lower one. In the ideal world, the prefetcher should fetch only the data which is required, just as the processor requires it.

To accommodate this, recent research is concerning itself with tuning these parameters dynamically based upon the current state of the system. One of these is proposed by Srinath et al called "Feedback Directed Prefetching" [74]. This ranks the current behaviour of the prefetcher based upon three critera:

PREFETCH ACCURACY: How accurate the prefetcher is. This is evaluated by calculating how many of the prefetches were actually used by the processor.

PREFETCH LATENESS: The timeliess of the prefetches. This is evaluated by recording how many prefetches were delivered to the processor by the time it required them. Even if they were not fully delivered, they may be coaleasced

with the actual prefetch by the prefetcher, masking *some* of the time associated with memory access.

CACHE POLLUTION: How many prefetches caused "useful" cache data to be displaced. This is evaluated by using a bloom filter to record which cache blocks were emitted due to a prefetch.

Based upon these critera, the prefetcher's parameters can be modified. As an example, if the prefetcher is accurate, but late, the prefetch distance can be increased to help timeliness. If the prefetcher is accurate but on time, the prefetch degree can be increased to fetch more useful data. If the prefetcher is causing cache pollution, the prefetch degree can be lowered to prevent extra, un-needed data from being fetched.

APOGEE [75] is another technique which attempts to tune the distance of a prefetch on a GPU for the purposes of energy reduction. The authors make the observation that while prefetching on each thread on a GPU is inefficient, a set of threads typically operate together in a SIMD (single instruction, multiple data) format. Moreover, these threads will typically exploit a similar access pattern, with a fixed stride between the threads. By observing the accesses of each of these threads, the stride between the threads can be ascertained, and prefetches dispatched. On top of this, the prefetcher evaluates the late-ness of each prefetch, and if they are late, increments the distance accordingly.

Another technique is "Adaptive Stream Detection" [76, 77]. This technique attempts to build a histogram of "stream length" at run-time, then uses this information to ascertain whether prefetches should be initiated. In the training period, this technique uses a table which holds the current streams, with the last address accessed in each one and a "lifetime" field. If a memory access cannot be corrlated with an existing stream, a new entry is added with a pre-determined lifetime, otherwise, the entry is updated and the lifetime reset. On each processor cycle, this lifetime is decremented, and if it becomes zero, the entry is removed from the table and the observed stream length is added to the histogram.

After the training period, the prefetcher then uses this table to track the currently observed prefetch streams in a similar way to stream prefetching. When a new access is observed, the prefetcher inspects the histogram, then will prefetch the next k consecutive blocks from memory if the inequality $\mathtt{lht}(i) < 2 \times \mathtt{lht}(i+k)$ is satisfied, where $\mathtt{lht}(i)$ is the number of memory accesses which occured which were part of a stream of length i.

2.4.2 Memory-Side Prefetching

So far, most of the techniques described in this section concern themselves with the local behaviour of a single task, and does not concern itself with the *global* state

of the system (with the exception of APOGEE [75], although arguably a parallel SIMD workload *is* the same task).

By moving the prefetcher into the memory controller, it is potentially possible for the prefetcher to issue requests based upon the current state of the system. Yedlapalli et al propose a technique called "Meeting Midway" [78] which attempts to create prefetches based upon the observed access stream *and* the current state of the memory controller. Again, as discussed in Section 2.2, performing a memory access to an already active memory row is extremely cheap. "Meeting Midway" therefore only prefetches data from a currently active DDR row by effectively also requesting the next few blocks when a DDR access occurs.

The technique also performs extra optimisations for multi-core systems. Firstly, it will not issue a prefetch if there are a large number of waiting demand accesses to prevent a detriment on the performance of other tasks in the system. It also only fetches the data into a buffer in the prefetcher to prevent cache pollution. For this reason, it can be used alongside a conventional core-side prefetcher. Finally, due to its position next to memory, it does not cause bus locking when performing a prefetch and hence does not block other tasks un-necessarially.

This technique is also used within other prefetchers. For similar reasons, the prefetcher used for "Adaptive Stream Detection" [76] was also designed to reside within the memory controller. Yang and Lebeck [79] also propose another prefetcher which is implemented within the memory controller which implements a similar pointer chasing routine to those shown earlier in this section. By placing the prefetcher inside the memory controller, prefetches could be issued much faster, as the prefetch coming from the processor did not have to cross the interconnect and other functional units inside of the processor.

2.4.3 Multi-core Prefetch

Much of the work presented within this section can be used within a multi-core context without modification. Also, some of the techniques (e.g. APOGEE [75] and "Meeting Midway" [78]) are designed to be used with a multi-core system. While these techniques exist, there is little work which evaluated the performance of a prefetcher on a multi-core system to observe the overall performance impact or the scaling problems which may arise.

Dahlgren et al [80] show the study of an adaptive prefetcher in a cache-coherent multi-core system, with shared memory at each node within the system. While the authors do model contention at each memory, they assume that the interconnect has infinite bandwidth and also assume an infinite cache in most experiments. Despite this, they do find that prefetching does improve the execution time and reduce the number of cache misses. While the authors also find a slightly increased

traffic consumption across the interconnection network, they do not provide any study on the amount of memory contention that takes place.

Tullsen et al [81] provide a similar study by assigning a fixed memory delay of 100 cycles, then varying how many of these cycles can be executed in parallel with other memory requests, and how many must be serialised over all processors. From this, the authors executed a number of benchmarks with a number of different prefetching schemes, and found that in almost all cases, prefetching worked well where the contended part of the memory bus latency was small, and hence prefetches could be completed quickly without significantly blocking other tasks. As the latency increased, however, prefetching tended to cause a performance detriment in all cases.

Other work by Ebrahimi et al [82] attempt to extended previous feedback-directed prefetching approaches [74] to multiple cores in order to try and throttle multiple core-side prefetchers and prevent prefetchers from negatively impacting the execution times of other processors. The target system is a multi-core system with a shared last-level cache, with a prefetcher on each core. The local prefetching characteristics (i.e. distance and degree) can be changed as with previous techniques, but a further function unit also monitors the *global* behaviour of the prefetcher and may override the control decisions made by the local prefetcher.

As an example, if a prefetcher is not operating in an accurate fashion and moreover, there is a great demand on shared memory by other prefetchers, then the global functional unit can cause that prefetcher to throttle down. Moreover, if the accuracy of a prefetcher is low, and it is also polluting the data stored in last-level cache belonging to *other* processors, then the global functional unit can throttle the local prefetcher. The authors found that this technique can typically improve the performance of the system further than standard adaptive prefetching, with lower bus bandwidths when applied to a multi-core system due to the lower amount of interference. Moreover, the authors also found that the gains due to the prefetcher actually *increased* with the number of cores in the system.

Finally, Liu and Solihin [83] explored the utilisation of prefetching on systems which already had bandwidth partitioning, comparing the speedups found when executing a set of tasks on a dual or quad-core system compared with serially on a single-core. As found in previous research, utilising bandwidth partitioning in itself can cause an improvement in the execution time versus a standard non-arbitrated system, but in all cases, the prefetcher could either cause a performance gain or a detriment, depending on the tasks being executed. From this, the authors derived a metric which decides whether each core's prefetcher should be enabled. From this, it can be shown that only enabling certain prefetchers is never any worse than the case without prefetching, and can typically yield performance gains.

2.4.4 Summary

Prefetching is a technique which is frequently used to try and hide some of the memory latency by attempting to predict what the processor will require next and attempting to fetch this before it is required, effectively attempting to speculatively overlap memory access with the computation time of a task. Of course, the effectiveness of this depends on many factors; the access pattern of a task should be predictable enough for an external unit to be able to determine the pattern, and there should be sufficient time in between memory accesses to be able to dispatch another prefetch.

While prefetching has been shown to be a useful technique on single-core systems, and on multi-core to an extent, there has been no work which attempts to evaluated its effects on a time-predictable system, or to be able to incorporate the prefetcher into the worst-case execution time calculations. Moreover, there are no current prefetching techniques which attempt to fit a prefetcher in alongside a hardware arbitration scheme in a safe manner; Liu and Solihin [83] are close, but the prefetcher used is still a standard prefetcher with no knowledge of the arbiter, which is simply then disabled if it is found to cause a performacne detriment ahead of time.

Given that more and more real-time systems are moving to multi-core systems in order to fulfill their performance requirements, a system like prefetching will soon be required in order to prevent memory latencies from becoming huge and extremely pessimistic, and causing large amounts of slack within a system. Methods such as memory-side prefetching could be useful for this, as their location next to the memory controller can allow them to gain an insight into the state of the entire memory subsystem, and hence only prefetch when it is "safe" to do so. Moreover, by combining them with a distributed arbitration approach such as those shown in Section 2.3.1, it may be possible to prefetch for a subset of cores to improve the execution time for a set of soft real-time tasks, while allowing hard real-time tasks to request at a high priority and still maintain a low deadline without much pessimism.

2.5 SUMMARY

This chapter has outlined much of the background work which both motivates the contributions of this thesis, and which is used by this thesis when exploring the hypothesis presented in Section 1.2. The major work detailed within this chapter can be summarised as follows:

Section 2.1 first details what it means for a task to be "predictable", and ultimately details what must be met in order to assert that the prefetching scheme described within this thesis is predictable. In order to meet the goal that the

prefetcher should be able to improve the worst-case execution time, the entire prefetching and arbitration scheme must be able to be modelled with a reasonably simple system model such that a static analysis tool can ascertain what the behaviour of the prefetcher will be at any time in the system in the worst-case. This can then be combined with a model of the processor, arbitration scheme and memory controller in order to arrive at an estimate for the worst-case execution time of a task in the system.

Section 2.2 then goes on to detail why modern, large memories cause issues for predictability in real-time systems. Both AMC [39] and Predator [38] attempt to distribute data across DDR banks in order to optimise bank efficiency, operate a closed-page policy and attempt restrict each access to a fixed "pattern" for which the worst-case response time can be ascertained. Both of these techniques provide a method by which the response time of a single memory transaction can be bounded and scheduled by an arbiter, and can hence be used by this thesis and thus provides predictability to the memory subsystem.

The rationale behind moving to multi-core architectures, and how multi-core systems can be made to be predictable with respect to a shared resource is then outlined in Section 2.3. While there are methods by which the access to a shared resource can be analysed without using any extra hardware, they are both expensive to analyse and are potentially unsafe in the case of a runaway task. In order to resolve this, Section 2.3.1 details a number of schemes by which multiple processors can access memory in a safe and fair manner. Such arbitration schemes can provide an upper bound on the number of requests which may block a request from a given processor, and hence can be combined with a predictable memory controller from Section 2.2 in order to determine the worst-case response time of a single memory access in a shared-memory system.

Finally, Section 2.4 details many of the current techniques for prefetching data for tasks and gives an overview of the recent work within multi-core prefetching. While many of these techniques can work well to improve the average-case execution time of a given task, none currently lay any focus on the *worst-case* timing of a task, or on being predictable. As detailed within Section 2.1, each of the components in the system must be predictable in order to ascertain a worst-case execution time for a task. If the prefetcher is not predictable, then the memory hierarchy as a whole is not predictable, making it difficult to make any assertions about the worst-case timing behaviour of the system.

This thesis will therefore build upon the work detailed in Sections 2.2, 2.3 and 2.4 in order to fill in some of this gap and to create a "predictable" prefetching scheme. In order to fulfil the hypothesis outlined in Section 1.2, it will then use some of the system analysis techniques described in Section 2.1 in order to analyse the worst-case behaviour of the prefetcher and attempt to integrate it with the worst-case execution time analysis of a task.

3 REAL-TIME PREFETCHING

Moving towards multi-core processing with an external shared memory causes significant issues within the field of real-time systems. As explored within Chapters 1 and 2, there is the requirement that the execution time of tasks within a real-time system can be bounded, such that it is possible to assert that they will complete their processing within a given time interval. To do so requires bounding the worst-case timing behaviour of everything within a system which a task uses, be it shared resources, memories within the system or even the functional units within the processor upon which the task is running.

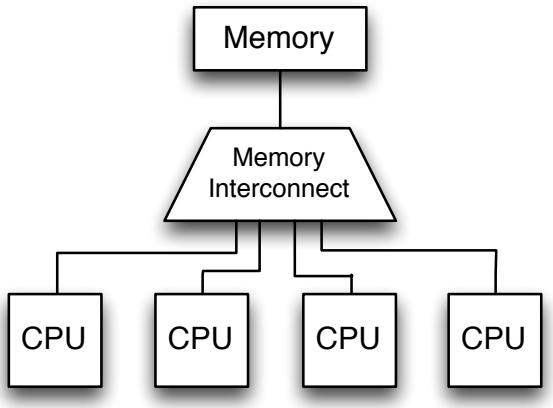


Figure 3.1: Graphical view of a system with four processors and a hardware memory arbiter.

Figure 3.1 shows a system model for a fairly simple multi-core system with shared memory. These are typically made up of a set of processors, which through some memory interconnect are then connected to a memory controller which issues commands to the underlying memory. There has already been extensive work on making each of these blocks predictable; Section 2.2.1 details many methods by which the memory controller can make the underlying DDR memory accessible in a predictable fashion, and there are many processor designs which are designed to be predictable from the outset [84, 85].

There are also many ways by which the memory interconnect can be made to be predictable. A few techniques use a standard, non-predictable interconnect, then attempt to analyse the behaviour of all tasks accessing it [86, 23]. While these techniques can be used on "standard" memory interconnects, they are computationally expensive; they must analyse the behaviour and interactions of each and

every task on the system together to determine the interference that they will cause each other, a problem which has effectively $O(N^2)$ complexity with respect to the number of tasks.

Some systems instead opt to utilise an arbiter in the place of the memory interconnect. An arbiter is a device which can fairly split the available memory bandwidth across a set of requesters, of which many techniques can be found in Section 2.3.1. As the amount of assigned bandwidth and the maximum latency of a memory request can be determined from this partitioned system is known ahead of time, a task can instead be evaluated against its own bandwidth bound. Because each task is analysed against its allocated bound, this technique far simplifies the system analysis, as now each task can be analysed in isolation against its own worst-case bound, and need not take other tasks into consideration. Because each task is analysed in isolation, the complexity of this technique is simply $O(N)$ with respect to the number of tasks.

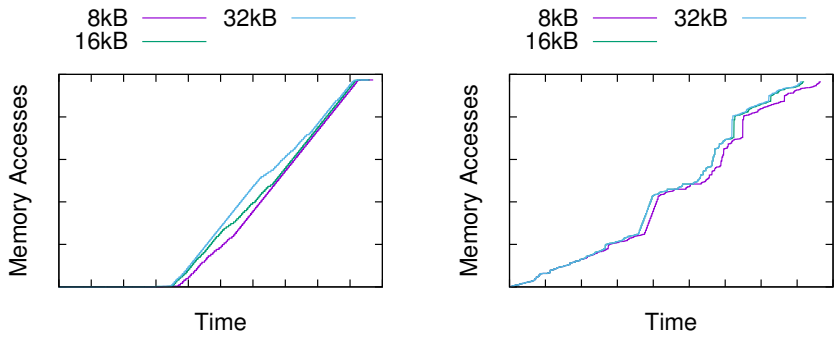
This is typically known as "composable" system construction [87], since a set of tasks can be combined together while still being able to provide the required level of service to all tasks, assuming that the required service is less than or equal to the provided service of the system. In order for an arbiter to be composable, it must provide a minimum defined service level to a given task, which cannot be negatively impacted by other tasks within the system. This means that tasks must be analysed in isolation and must not attempt to reclaim "spare" time from other tasks.

3.1 MEMORY ARBITRATION

Many arbiters which can typically be used to control access to shared resources such as memory fall into the Latency-Rate ($\mathcal{LR}$) class of arbiters [55]. These arbiters provide an average rate of service ρ to their clients after a maximum latency σ. As an example, assuming that a requester occupies a single slot in a four-slot TDM schedule, with an active period of t_{period}, the requester will be allocated a quarter of the available bandwidth (hence $\rho = 0.25$), and in the worst case must wait for the whole schedule to cycle before being able to make a request (hence $\sigma = t_{period} \times 4 - 1$).

In order to be used with an $\mathcal{LR}$ arbiter, the average request rate must be ascertained for a task. This may be simple for some tasks; a task which works on serial data in a simple loop will access the data storage in a fairly predictable pattern. A task which carries out many different functions, on the other hand, is far more difficult to predict since the different paths through the task may yield different request rates. Ascertaining this rate is then made even more complicated when extra hardware features are used, such as caches and prefetchers. Moreover, this

request rate may differ based upon the current control flow of the task; certain paths may require more data, faster from memory and hence the system designer must consider the average-case request rate, across all paths, and the worst-case rate, which may only be rarely invoked.



(a) sha **(b)** binarysearch

Figure 3.2: Access timings for benchmarks with varying cache sizes.

Examples of a task's request rate can be seen graphically in Figure 3.2, which plot the cumulative number of memory accesses over time for the *sha* and *binarysearch* benchmarks. Here, the *sha* benchmark computes the SHA-1 hash for a serial block of data and has a relatively linear access pattern. The *binarysearch* benchmark accesses data much more randomly and exhibits "bursty" behaviour. The access pattern depends heavily on the input data and cache behaviour of the task; the block size of the cache means that any required data may have already been fetched into cache by a previous access.

The end result of this is that it is not always possible to be able to ascertain a static bandwidth bound which accurately captures the behaviour of the task. For example, for the *binarysearch* benchmark, the system designer must choose to provide a large portion of the available memory bandwidth, and optimise the worst-case execution time (WCET) with the trade-off that the task will not always fully utilise its bandwidth bound, or provide a smaller portion of the bandwidth and ensure that no bandwidth is wasted at the cost of the WCET. Finally, the assigned partition must be able to cover the worst-case path of a program. If this path is not frequently taken, then the bandwidth bound may again be an over-estimate.

Over-estimating the required bandwidth in this way can create some "spare" time within the system as the arbiter will be providing the requester with service, when it has no requests to service. The conventional way to deal with this spare time is to use a technique called work conservation. This attempts to ensure that the shared resource is never idle by scheduling the work generated by another task whenever spare time would be created by the arbiter. For example, in a TDM system, if the currently active task has no requests outstanding, a request from another task can be taken instead. Note that this would not necessarily advance

the schedule, as this can change the timing behaviour of the arbiter, it simply chooses a request from a different requester to schedule in the "spare" time.

While this keeps the arbiter busy, the work-conserving behaviour of many arbiters is typically not well defined without breaking composability, since this would require knowledge of the request patterns of the other tasks. As an example, both the arbiter used in [12] and CCSP [9] fall back to a static priority scheme in the work-conserving mode, hence the access behaviour of all other tasks in the system would need to be known in order to improve the WCET using work-conservation (except for the highest priority task).

Finally though, work conservation only attempts to re-distribute spare bandwidth amongst other requesters. Within the context of the memory subsystem in a real-time task, this partially ignores the issue of rising memory latencies. Even if the the work-conserving behaviour of the arbiter could be analysed and incorporated into the WCET analysis, the potential gains are still bounded by memory latency. By using different approaches to consume this slack time, the memory latency could be eliminated entirely in certain cases.

3.2 PREFETCHING

Prefetching is one technique which arose to try and cope with these rising memory latencies. It attempts to do this by speculatively fetching blocks from memory directly into the cache of the target processor based upon the pattern of memory addresses it has already accessed. As an example, if a processor has accessed addresses A, $A+1$ and $A+2$, it is quite likely to soon require the data at address $A+3$. This is, of course, a simple heuristic; many other approaches have been discussed in detail within Section 2.4.

One other motivator for prefetching is that, even in single-core systems, main memory is not typically fully utilised. The cost of a memory request crossing interconnects, crossing the memory controller, and the response being delivered back to the processor poses a significant overhead. Caches are another technique to cope with this rising latency which work by eliminating redundant memory accesses, but adding a cache to the system further reduces the utilisation figure for main memory. Figure 3.3 shows this effect that a cache can have on memory utilisation and plots the memory use for a number of benchmarks for varying cache sizes. These experiments use a single Microblaze [88] processor running at 100MHz with a standard DDR3 memory controller running at 200MHz. On top of the delays inherent to the processor and memory system, the interconnect causes there to be 18 cycles of latency for each memory transaction.

Of course, the memory controller sitting idle is bad; the memory controller could be doing some useful work instead. In these cases, prefetching attempts to utilise

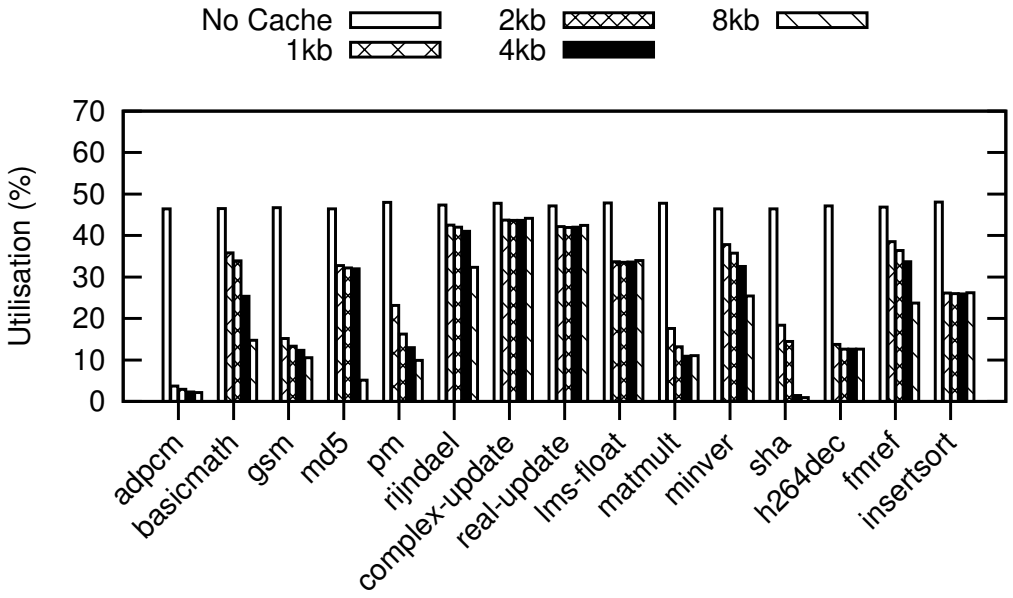


Figure 3.3: Memory utilisation for a number of benchmarks and cache sizes.

this spare bandwidth to speculatively issue a memory access for something which is *potentially* useful to the processor. Figure 3.4 shows this performance boost when used in an identical system to that in Figure 3.3. This example uses a simple stream prefetcher, hence yields good results for benchmarks which access serial data, e.g. *md5* and *insertsort*. It also compliments benchmarks with large amounts of straight-line code well, for example *basicmath* and *rijndael*. Some benchmarks show little improvement, however. *matmult* is a small benchmark which executes over a number of iterations. In this case, the benchmark itself can fit entirely in cache, from which it then executes until it completes without accessing memory, hence there is nothing useful to prefetch.

While prefetching has been shown to be a valuable technique for single-core systems, there is little research on its impact on a multi-core system. That which does exist typically concerns itself with cache-coherent systems and the impact which the prefetcher has on the false sharing of data, or just concerns itself with the average case. There has been work on evaluating a prefetcher on a bandwidth partitioned multi-core system [83], but this effectively only presents a methodology to disable the prefetcher when it can be shown to give a performance detriment ahead of time. There is also little work on the effects of a prefetcher on a real-time system; Lee et al. [89] show that automatically prefetching the instructions on the worst-case path can improve performance over standard prefetching, but this work ignores any bus contention and interference that prefetching can cause, and only attempts to improve the worst case path through the program, which may only be taken very rarely.

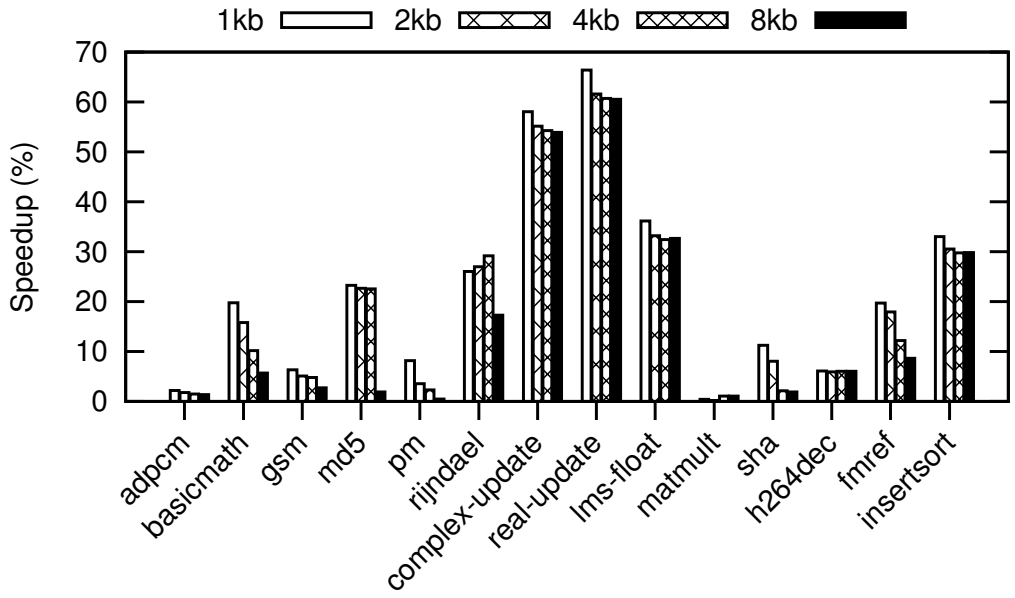


Figure 3.4: Performance increase when using basic prefetching on various benchmarks

Prefetching is not typically used within real-time multi-core systems for a simple reason: a prefetcher is generally a separate functional unit within the system which can issue memory requests as it pleases. This typically causes the system analysis to become infeasible; if the amount of blocking is not known or bounded, then a worst-case execution time cannot be ascertained for the system as a whole. Even if the prefetcher does have deterministic behaviour, the potential interference that it will cause will cause a great amount of pessimism within the analysis.

As an example, the prefetch system presented by Dahlgren et al [80] fetches the next K cache blocks after a cache miss, if they do not already reside in cache. In the worst case, this of course causes another K memory accesses to occur after performing a memory access. Any system analysis must now assume that each memory access may be blocked by K memory accesses from the previous load, hence the cost of each memory access must now be multiplied by $K + 1$ causing the amount of pessimism in the system analysis to greatly increase. Some prefetchers sidestep this issue by imposing a priority ordering between prefetches and demand miss accesses, only emitting prefetches when there are no demand misses waiting for service, like the scheme proposed by Hur and Lin [76]. While this will prevent the blocking issue, it does have the problem that in a system with sufficiently high load, the prefetch will be blocked by demand accesses for a long period of time, potentially rendering the prefetch useless and preventing further prefetches from being initiated.

Moreover, the unpredictability of prefetchers causes issues for systems which utilise cache. Typical prefetchers prefetch directly into the cache of the target processor, but any cache analysis which has taken place within the system analysis

will assume that the contents of the cache can only be affected by the currently running task. By fetching directly into the cache of the target processor the prefetcher may displace data which the processor required in the near future, hence harming performance and invalidating any cache analysis which may have already taken place, since the analysis tool can no longer make any concrete assumptions about the contents of cache.

Some techniques instead move the prefetcher into the memory controller itself, rather than the processors [77, 90, 78]. Inserting the prefetcher here in the memory hierarchy is interesting because it is now possible for the prefetcher to ascertain the state of the system and hence only prefetch when there is available bandwidth. Such an approach is also global; the bandwidth which is left unused by one processor can be used to prefetch data for another processor instead. It may also be able to select data to prefetch based on the *global* access stream, rather than just that for a single processor. While this is an interesting concept, it does not actually fix the issues that have already been presented within the global scope of the system. A work-conserving arbiter will still attempt to schedule requests whenever there is "spare" time, hence blocking prefetches.

These issues typically prevent prefetchers from being used within real-time systems due to both their unpredictability and the amount of bandwidth they can potentially consume. For the most part though, both of these issues arise simply because the prefetcher doesn't have any knowledge of the system; to be used in a real-time system, the prefetcher needs to be integrated into the system such that it can reliably determine when a prefetch can be dispatched.

3.3 REAL-TIME PREFETCHING

As previously discussed within Section 3.2, prefetching is an attractive method by which the rising memory latencies found within modern systems can be masked. Despite this, prefetching is typically not used within a real-time system because of the many pitfalls detailed previously; the prefetcher can operate at random intervals, causing unpredictable blocking, and can displace useful contents from a processor's cache, causing the processor to initiate spurious memory requests.

These pitfalls exist solely because the prefetcher is not controlled in any meaningful way with respect to the system as a whole; without any knowledge of the scheduling of the system, the prefetcher will not be able to throttled to prevent it causing harm. As presented within Section 1.2, this thesis will attempt to prove the hypothesis that by integrating the prefetcher into the global arbitration scheme, it can be used safely within a real-time system, without causing any harm to the existing worst-case execution time.

In order to evaluate existing prefetching schemes and build upon them in a meaningful way, a general model of the real-time system must be decided upon. This model will then be evolved throughout the thesis, with a hardware realisation at each step to evaluate the effectiveness of the prefetching scheme in the real world. The remainder of this Section will take the existing literature from Chapter 2 and use this literature to construct a model under which these investigations can take place.

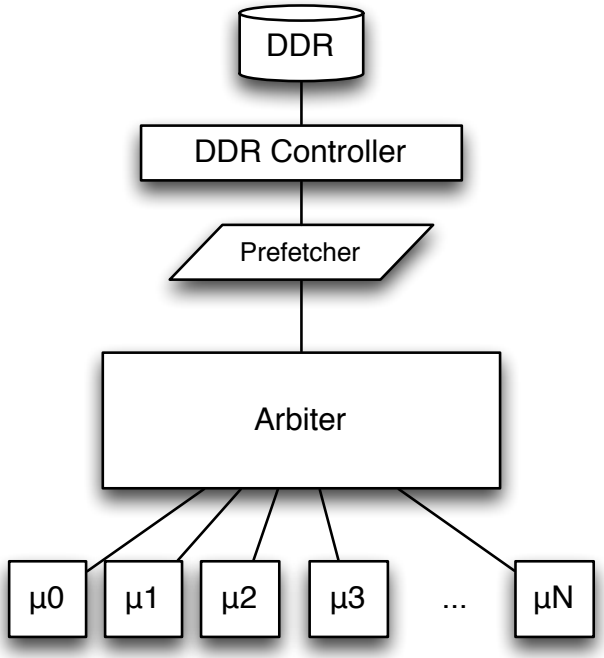


Figure 3.5: Block diagram of the system model.

Conceptually, the system is comprised of a number of requesters, each of which initiate memory requests which are then passed through a fair, composable arbitration scheme. After this, they are then issued to the memory controller, which will then either respond with the requested data or an acknowledge packet. Finally, there is a prefetcher integrated into the system which, as stated before, can issue memory requests on behalf of the requesters in the system. A block diagram of this model can be found in Figure 3.5.

At a high level, this system is comprised of a set of requesters $\mathbb{C}$. Each requester $r \in \mathbb{C}$ issues a set of requests $W(r) = \{\omega_1^r, \omega_2^r, ..., \omega_n^r\}$, which are memory requests in an address space $\mathbb{A}$, where the set of all possible requests is described with $\mathbb{C}$. Each of these requests are dispatched at time $t_d(\omega_n^r)$, and a response arrives back at the initiator at time $t_a(\omega_n^r)$. Each access operates on a single address $A(\omega_n^r) \in \mathbb{A}$ and is either a read or a write. Each of these requesters may have real-time requirements, such that each request must complete in a known and bounded

amount of time $t_{wc}(r)$. While t_{wc} may be different for each requester, it must be the same for each request initiated by a given requester, hence:

$$\forall r \in \mathbb{C}, \omega_n^r \in W(r) : t_a(\omega_n^r) - t_d(\omega_n^r) \leqslant t_{wc}(r) \tag{3.1}$$

In order to satisfy this constraint, the response time of each component within the block diagram in Figure 3.5 must be known and bounded. The remainder of this section will detail the assumptions made about each of these blocks such that a worst-case response time can be ascertained, and therefore present a general system model under which prefetching experiments can take place.

3.3.1 Requesters

As stated previously, the requesters are a set of hardware components $r \in \mathbb{C}$ connected to the inputs of the arbiter and issue a set of requests $W(r) = \{\omega_1^r, \omega_2^r, ..., \omega_n^r\}$ to shared memory. These requesters may be processors running real-time tasks, or may be hardware peripherals which require access to shared memory. Irrespective of the implementation details of each requester, it is assumed that each of their memory requests must complete within a bounded period of time, $t_{wc}(r)$. The actual hardware implementation of these requesters is orthogonal to this work, since this work only intends to ensure that the memory subsystem is predictable; it is assumed that the behaviour of the requester can be analysed given an analysable memory system given that $t_{wc}(r)$ can be derived.

Each of these requesters may issue a request $\omega_n^r \in W(r)$ to read or write data from address $A(\omega_n^r) \in \mathbb{A}$, on the granularity of B bytes at a time. When a requester initiates a read request, it is expected that it will receive a response with the B bytes at the specified address within $t_{wc}(r)$ cycles, and that similarly, when a requester initiates a write request, the data contained within the request will be written to memory, and an acknowledgement delivered back to the requester within $t_{wc}(r)$ cycles.

In order to support a prefetcher, the requesters must also be able to receive data "pushed" to them from the memory subsystem. In order to distinguish it from other traffic (e.g. to prevent "pushed" data being mistaken as a read response), the data will be tagged as "prefetched" to allow the requester to handle it accordingly. In order for the requester to be analysable within the system as a whole, any data which is pushed into the requester must not cause it to initiate any spurious memory requests, for example, Section 3.2 observed that prefetching directly into the target cache of a processor may displace useful data and hence cause repeated requests for the same data. The requester must ensure that this scenario does not occur.

3.3.2 Arbiter

The arbiter is a many-to-one component which is responsible for scheduling requests from the requesters $\mathbb{C}$, and relaying them to the next component in the chain in a fair and predictable manner. While forwarding these requests "up" through the arbiter, the arbiter must not modify the request in any way, except to tag each request with the requester ID $r \in \mathbb{C}$ so that a response can be routed to the correct requester.

The arbiter must also support routing responses back "down" through the arbiter, from the memory controller (or prefetcher), back down to the requesters again. Any responses being routed in this manner must be tagged with the destination requester $r \in \mathbb{C}$. The requester indices must maintain the same mapping in both directions (i.e. a a request from index x and a response to index x must refer to the same requester).

How a request is routed from the requesters to the next stage is irrelevant to this system model, however, there are a few constraints. It must be possible to ascertain a worst-case bound on the time taken to cross "up" the arbiter for a given requester, $t_{wc}^{arb\uparrow}(r)$. This worst-case must only depend on the requester index, and must not depend upon any other details of the request. In order to optimise throughput, the arbiter may accept multiple pipelined requests from any given input, and be able to provide the worst-case time between two requests being accepted $\delta^{arb}(r)$, this creates two constraints on the times at which a given request is accepted by the arbiter, $t_a^{arb\uparrow}(\omega_n^r)$ and dispatched from the top of the arbiter $t_d^{arb\uparrow}(\omega_n^r)$:

$$\forall r \in \mathbb{C}, \forall \omega_n^r \in W(r) : t_d^{arb\uparrow}(\omega_n^r) - t_a^{arb\uparrow}(\omega_n^r) \leqslant t_{wc}^{arb\uparrow}(r) \tag{3.2}$$

$$\forall r \in \mathbb{C}, \forall \omega_n^r, \omega_k^r \in W(r) : k > n \implies t_a^{arb\uparrow}(\omega_k^r) \geqslant t_a^{arb\uparrow}(\omega_n^r) + \delta^{arb}(r) \tag{3.3}$$

The time taken to route a packet downwards, $t_{wc}^{arb\downarrow}(r)$ should also be bounded for each requester. Since the "down" path back to the requesters has only a single logical master, there should be no contention for responses. For this reason, all responses should be delivered in a constant time, and not depend upon the current system load or access pattern, but may change based upon the requester index.

As a simplification for later stages of the system model, both the arbiter and the set of all requesters can be treated as a single master which issues a superset of all transactions from all requesters. This superset W^* is therefore the set of all requests from all requesters such that $\forall \omega_n^r \in W^* : \omega_n^r \in W(r)$.

3.3.3 Memory & Memory Controller

As described in Section 2.2, DRAM is unpredictable, but generally used where a large memory is required because of its high density and low cost. While some of this unpredictability is caused by potential speedups (e.g. subsequent accesses to an already open row are faster than the first request which opened the row) and can hence be avoided (e.g. by always closing the row after every request), some of it is inherent to DRAM (e.g. the need for refresh cycles, read-to-write switching and vice versa) and cannot be side-stepped by simply returning to a known state between two transactions.

The memory controller is the unit which is actually responsible for scheduling these commands onto the physical DRAM chips. The memory controller must decide when to open and close DRAM rows, when to issue DRAM refresh cycles and so on. Because of the tight coupling required between the memory controller and the memory itself, this system model treats the two as a single functional unit.

The memory controller must respond to two types of request, reads and writes, each for an address $A \in \mathbb{A}$. This address space $\mathbb{A}$ must be identical for all requesters and for both reads and writes. There is no concept of "virtual memory", and all addresses are physical. Each address A corresponds to a location of memory which stores B bytes of data, where no two addresses should correspond to the same memory location (i.e. there is no address aliasing). When the memory controller receives a read request for location A, it should respond to the respective requester with the B bytes of data stored at location A. Similarly, on a write, the memory controller must take the B bytes of data stored within the request, and commit them to location A. For a write request, the requester may also provide a "byte-enable" to write a subset of bytes to prevent having to read the data before modification. After writing, the memory controller must respond with a "write acknowledge" response.

In order to be used within a real-time system, the memory controller must yield these responses within a given period of time, denoted as t_{wc}^{mem}. As with all other units in the system, this time must not depend upon any of the parameters of the request itself, it must be a single time to cover the worst-case of all requests. The memory controller may pipeline requests if required, with an inter-request time of δ^{mem}, but must not reorder its requests in any way[1]. Given that the arrival

1 Given that some memory modules contain many distinct banks of memory, some controllers may reorder transactions to optimise throughput by scheduling two requests which target different banks at the same time, even if there are other requests between them. Other controllers also reorder transactions if there are many requests targetting the same row to reduce the delay associated with precharging the memory and opening the same row. Such behaviour is typically difficult to capture in the system model, and again depends upon the set of requests which came before a given memory access.

time of a request, and dispatch time of a response are denoted by $t_a^{mem}(\omega_n^r)$ and $t_d^{mem}(\omega_n^r)$, the following constraints hold:

$$\forall \omega_n^r \in W* : t_d^{mem}(\omega_n^r) - t_a^{mem}(\omega_n^r) \leqslant t_{wc}^{mem} \tag{3.4}$$

Thus all requests must complete by the given worst-case response time.

$$\forall \omega_n^r, \omega_k^q \in W^* : t_a^{mem}(\omega_n^r) > t_a^{mem}(\omega_k^q) \implies t_a^{mem}(\omega_n^r) \geqslant t_a^{mem}(\omega_k^q) + \delta^{mem} \tag{3.5}$$

Thus all requests have a minimum separation of δ^{mem} cycles.

$$\forall \omega_n^r, \omega_k^q \in W^* : t_a^{mem}(\omega_n^r) > t_a^{mem}(\omega_k^q) \implies t_d^{mem}(\omega_n^r) > t_d^{mem}(\omega_k^q) \tag{3.6}$$

Thus responses must be delivered in the same order which the requests arrived.

3.3.4 Prefetcher

The prefetcher is the unit under investigation which sits between the arbiter and the memory controller. Its purpose is to observe the set of all requests from the arbiter, attempt to ascertain a correlation of the addresses being fetched and if one can be found, speculatively issue an access along the same pattern.

Upon receiving a read request, the prefetcher may update some of its internal state, but must allow the packet to transit the prefetcher without any modification. Optionally, after receiving this read request, the prefetcher may also emit a prefetch based upon its internal state. Upon receiving a write request, the prefetcher must simply allow the request to transit the prefetcher without modification. Because there is no requirement that data must be read before writing in this model, the prefetcher need not act upon write requests, since if the requester is reading before writing, the stream will be correlated from the reads, and if the requester is only writing then initiating a prefetch based upon the writes will only deliver data to the requester which it will never read.

As with all other modules in the system, the prefetcher must operate in a predictable manner in order to operate within a real-time system. As previously, the arrival time of a request at the prefetcher and the time at which the request is issued from the prefetcher to the memory controller is denoted with $t_a^{pf\uparrow}(\omega_n^r)$ and $t_d^{pf\uparrow}(\omega_n^r)$, respectively. Moreover, the time to transit the prefetcher must be bounded and is represented with $t_{wc}^{pf\uparrow}$. As with all other modules, the timing of the "down" path back from the memory controller to the arbiter must also be bounded

and have no blocking, and is represented with $t_{wc}^{pf\downarrow}$, where the arrival time from the memory controller and the dispatch time to the arbiter are represented with $t_a^{pf\downarrow}(\omega_n^r)$ and $t_d^{pf\downarrow}(\omega_n^r)$. It holds therefore that

$$\forall \omega_n^r \in W* : t_d^{pf\uparrow}(\omega_n^r) - t_a^{pf\uparrow}(\omega_n^r) \leqslant t_{wc}^{pf\uparrow} \tag{3.7}$$

$$\forall \omega_n^r \in W* : t_d^{pf\downarrow}(\omega_n^r) - t_a^{pf\downarrow}(\omega_n^r) = t_{wc}^{pf\downarrow} \tag{3.8}$$

There is also an exception to the "the prefetcher must not modify packets" rule. The prefetcher may coalesce a read request with an outstanding prefetch if the two are for the same address and for the same requester. Because the response time of memory is known and bounded, this still works within the realm of a real-time system, the intuition of which will be explained in Section 3.3.5. This mechanism allows an outstanding prefetch to satisfy the read request, hence alleviating some of the latency of the read request.

3.3.5 Discussion

Currently, this section has provided the intuition that each component within the system must have worst-case bounds, but has ignored the system as a whole. The first stipulation on the system as a whole is that there must not be any buffers in-between two functional units; when a request leaves one unit, it must be accepted by the next in the same cycle. It thus follows that:

$$\begin{aligned} \forall r \in \mathbb{C}, \forall \omega_n^r \in W(r) : t_d(\omega_n^r) &= t_a^{arb\uparrow}(\omega_n^r) \\ \wedge t_d^{arb\uparrow}(\omega_n^r) &= t_a^{pf\uparrow}(\omega_n^r) \\ \wedge t_d^{pf\uparrow}(\omega_n^r) &= t_a^{mem}(\omega_n^r) \\ \wedge t_d^{mem}(\omega_n^r) &= t_a^{pf\downarrow}(\omega_n^r) \\ \wedge t_d^{pf\downarrow}(\omega_n^r) &= t_a^{arb\downarrow}(\omega_n^r) \\ \wedge t_d^{arb\downarrow}(\omega_n^r) &= t_a(\omega_n^r) \end{aligned} \tag{3.9}$$

The relationships between these timing characteristics can be seen graphically in Figure 3.6. This timing diagram lists each functional unit in both directions (hence Arbiter $\uparrow$ denotes the request is currently "in" the arbiter, transiting towards memory) and shows how a request moves through these units in the worst-case, and within what time periods. Each time interval listed above the timing diagram (e.g. $t_{wc}^{arb\uparrow}$) denote the worst-case timing bounds for a request to transit each

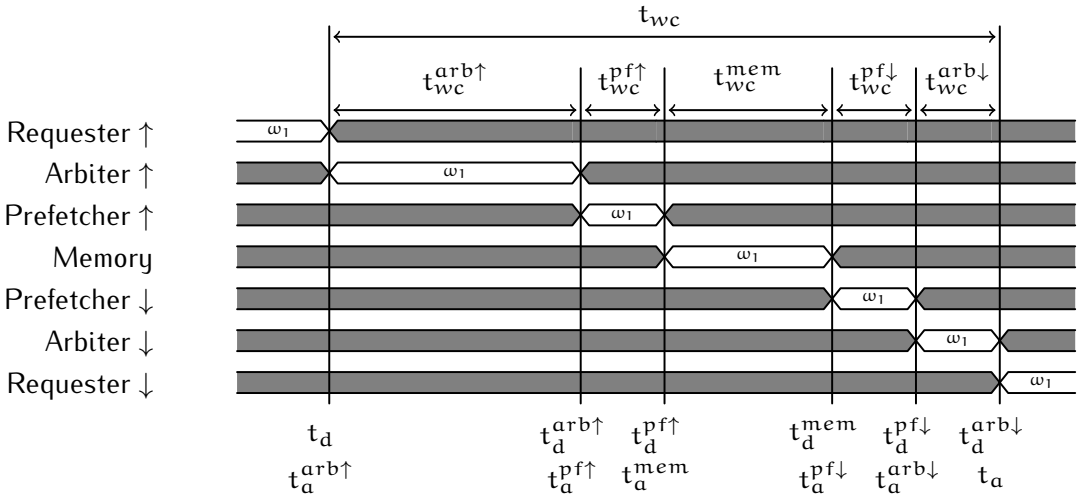


Figure 3.6: Timing diagram showing the time taken for a request ω_1 to cross each functional unit in the system.

functional unit, and each time instant listed below the diagram (e.g. t_d) denote the times at which the packet arrives at or is dispatched from each functional unit.

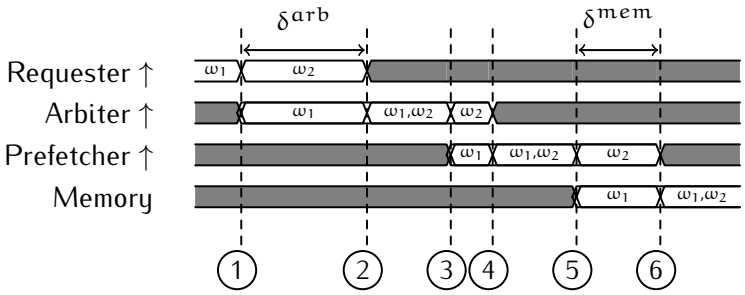


Figure 3.7: The relationships between the inter-request times (e.g. δ^{mem}) for two requests, ω_1 and ω_2 from the same requester.

Figure 3.7 also shows how the definitions of the "inter-request" times (i.e. δ^x) influence how requests are relayed through the system, and shows example worst-case figures for the inter-request times (note that this does *not* demonstate the worst-case crossing times t_{wc}^x). This diagram shows two requests, ω_1 and ω_2 as they transit from the single requester to memory, a process which operates as follows:

1. ω_1 is accepted by the arbiter, and at the same instant the requester attempts to issue another access ω_2.

2. After a period of δ^{arb} cycles (in the worst-case), the arbiter is now able to accept another request from the requester, ω_2. Note that the arbiter is still handling request ω_1 at this point.

3. The arbiter now relays ω_1 on to the prefetcher, and is still currently processing ω_2.

4. In this contrived example, ω_2 did not experience as much blocking as ω_1 and hence emerges from the arbiter and in to the prefetcher. Note that the prefetcher is still also processing ω_1 at this point, but can pipeline both requests.

5. The prefetcher has completed processing ω_1, and the memory controller has space in its input queue and hence ω_1 is relayed to the memory controller.

6. The prefetcher relays ω_2 to the memory controller. The prefetcher may have completed processing ω_2 before this point, but may be blocked by up to δ^{mem} cycles by the memory controller if the memory controller could not accept ω_2 in the meantime.

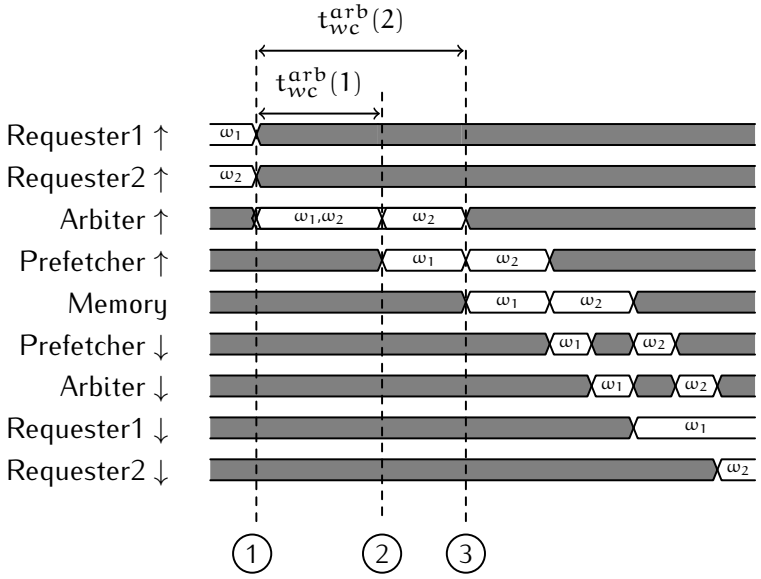


Figure 3.8: Timing diagram showing two requests ω_1 and ω_2 dispatched from two different requesters simultaneously.

Of course, the pipelining can be performed on requests from different requesters, if required. Figure 3.8 shows two requests being initiated simultaneously from two different requesters. The operation of this is similar to that in Figures 3.6 and 3.8, but each requester has different worst-case timings to cross the arbiter. The key points of this diagram are as follows:

1. Both ω_1 and ω_2 are accepted simultaneously by the arbiter on two separate inputs. Both requests will then be processed by the arbiter in subsequent cycles.

2. After a maximum of $t_{wc}^{arb\uparrow}(1)$, the request from requester 1, ω_1 emerges from the arbiter into the prefetcher. It is then processed by the rest of the components in the system as before.

3. After a maximum of $t_{wc}^{arb\uparrow}(2)$, request ω_2 emerges from the arbiter and again, is processed by the remainder of the components within the system.

For simplicity, both requests do not reside in the rest of the system's functional units (e.g. the prefetcher or the memory controller) at the same time in this diagram. This case is, of course, possible, whilst still observing the δ^{arb} and δ^{pf} constraints.

Finally, the set of defined constraints combined with the relationships defined between $t_d(\omega_n^r)$ and $t_a(\omega_n^r)$ for each of the functional units within the system also allows the worst-case response time to be ascertained when using any component as a reference point. $\hat{t}_{wc}^{x}$ is used to denote the worst-case response time of a request, from the point that it is dispatched from functional unit x to the point that a response is received by the same unit. This is logically the worst case of all functional units "above" the current functional unit. As an example, for the prefetcher, $\hat{t}_{wc}^{pf}$ is defined as follows:

$$\hat{t}_{wc}^{pf} = \max^{x} t_a^{pf\downarrow}(\omega) - t_d^{pf\uparrow}(\omega) = x \tag{3.10}$$

Due to Equation (3.9), $t_d^{pf\uparrow}(\omega) = t_a^{mem}(\omega)$ and $t_a^{pf\downarrow}(\omega) = t_d^{mem}(\omega)$. Equation (3.4) then provides an upper bound on the difference between $t_d^{mem}(\omega)$ and $t_a^{mem}(\omega)$ hence:

$$\hat{t}_{wc}^{pf} = t_{wc}^{mem} \tag{3.11}$$

Intuitively, the following equations hold for the response time of the remainder of the system:

$$\hat{t}_{wc}^{arb} = \hat{t}_{wc}^{pf} + t_{wc}^{pf\uparrow} + t_{wc}^{pf\downarrow} \tag{3.12}$$

And that a worst-case for an entire memory request, $\hat{t}_{wc}(\omega_n^r)$ can be ascertained:

$$\hat{t}_{wc}(\omega_n^r) = \hat{t}_{wc}^{arb} + t_{wc}^{arb\uparrow}(r) + t_{wc}^{arb\downarrow}(r) \tag{3.13}$$

This set of equations now allows us to express that the prefetch coalescing system presented within Section 3.3.4 will not harm the worst-case. This mechanism allows an outstanding prefetch to fulfil a read request, assuming that both the prefetch and the demand access are for the same address and for the same requester. Assuming the prefetch is ω_{pf}^r and the demand access ω_n^r, the assumption

is that the prefetch was initiated before the respective demand access would have left the prefetcher, hence:

$$t_d^{pf\uparrow}(\omega_{pf}^r) < t_d^{pf\uparrow}(\omega_n^r)$$

Logically, these requests would complete after $\hat{t}_{wc}^{pf}$ cycles in the worst-case, hence $t_a^{pf\downarrow}(\omega_{pf}^r) = t_d^{pf\uparrow}(\omega_{pf}^r) + \hat{t}_{wc}^{pf}$, and similar for ω_n^r. Because $\hat{t}_{wc}^{pf}$ is constant, it hence follows that $t_a^{pf\downarrow}(\omega_{pf}^r) < t_a^{pf\downarrow}(\omega_n^r)$ and because the time taken to transit "down" the prefetcher, $t_{wc}^{pf\downarrow}$ is constant, it hence follows that the prefetch will always complete before the respective demand miss would have.

Of course, while this section shows that the prefetcher will not cause a detriment to the worst-case response time of a memory access $t_{wc}(\omega)$, it completely ignores that the prefetcher must be able to initiate memory requests of its own in order to be useful. At the moment however, this is almost impossible to fit into the model.

The arbiter currently assumes that the worst-case time between a request leaving the arbiter and a response arriving, $\hat{t}_{wc}^{arb}$ is simply based upon the time to transit the prefetcher and the latency of the memory controller. If the prefetcher issues memory requests of its own, these will have an impact on the definition of $\hat{t}_{wc}^{arb}$ as the arbiter must now wait for a potential prefetch to complete before its request will be admitted. As there is no model of how the prefetcher is currently operating, this is currently impossible. The model of the prefetcher could be changed such that it can only issue a prefetch in response to a memory request arriving, but this simply causes the worst-case delay imposed by the prefetcher $t_{wc}^{pf\uparrow}$ to also include another memory transaction. In effect, this causes the worst-case response time of a memory transaction to double, which is clearly unacceptable.

In order to prove the hypothesis that it is possible to use a prefetcher "safely" within a real-time system, it must be possible to integrate the prefetcher in such a way that the worst-case delay caused by the prefetcher does not increase. In order to do so, this will require that the behaviour of the prefetcher is better defined and importantly, integrated with the rest of the components within the system in order to ensure its safety.

These ideas will now be explored further throughout the remainder of this thesis. Chapter 4 will flesh out the model presented within Section 3.3 with a "standard" prefetch approach in order to investigate the impact of the prefetcher on the worst-case execution time of a real-time system. Chapter 5 will then take these results, and propose modifications to the model to create a feedback system under which a prefetcher can be safely integrated into a real-time system, before also demonstrating this safety with a hardware implementation. Finally, Chapter 6 will build on this system to provide a framework by which the worst-case execution time of a task can be improved through the inclusion of a prefetcher, and Chapter 7 will sum up the contributions and provide potential future research avenues.

4 REAL-TIME PREFETCHING ON MULTICORE

4.1 INTRODUCTION

As explored within Section 2.4.3, there is already some work investigating the effects of prefetching on multi-core systems. While some of this work has explored the effects of prefetching on bus utilisation [81, 80], none have really explored the impact of varying numbers of cores prefetching on both the bus utilisations and execution times of tasks. Moreover, no work has yet investigated the effects of prefetching on a system with real-time requirements. Some work has investigated the problems with prefetching within a bandwidth-partitioned system [83], but this work only really provided a basic metric to determine whether a prefetcher should be used within a partition or not.

Chapter 3 outlines how prefetching is potentially a valuable strategy to mask the rising worst-case access costs for memory accesses within real-time systems. As discussed towards the end of Section 3.3.5, however, there is a critical flaw with current prefetchers; they are not built in an analysable way and hence it is currently impossible to predict what a prefetcher will actually prefetch and when.

In order to provide motivation behind this work, this chapter will investigate the actual performance impact which a prefetcher will have on a real-time system. To do so, it will first provide a hardware realisation of the system model defined within Section 3.3 and, using this system, execute a number of tasks on it to investigate how a prefetcher can cause an improvement or detriment to the execution time of these tasks. This will then provide the intuition behind the subsequent chapters of this thesis which will use the results of this work to propose modifications to the system model in order to perform "safe" prefetching within a real-time system.

4.2 SYSTEM ARCHITECTURE

Firstly, a realisation of the system model defined in Section 3.3 must be constructed in order to allow experiments to be carried out to evaluate the impact of prefetching on a "standard" system. In order to support the flexibility to quickly modify the behaviour of the system components (i.e. the processors and the prefetcher) while still allowing the fast turnaround of experiments, the experimental system was created on a Xilinx VC709 FPGA board [91]. This board contains a large

Virtex 7 FPGA along with 8GB of DDR3 memory, of which 2GB is used by the experimental platform. The remainder of this section will detail the construction of the components used in this evaluation platform, along with any pre-built FPGA IP (e.g. processors and memory controllers) that they use.

4.2.1 Arbitration Scheme

Because this work concerns itself with many-core real-time systems, there are a few design considerations which must be taken into account when designing the memory interconnect. The interconnect itself must firstly be able to scale to a large number of requesters, and present some path for future scaling. As also stated within Section 3.3.2, the arbiter must also provide access to shared memory in a fair and composable way.

The first of these considerations necessitates a move towards a distributed arbitration scheme as large, monolithic arbiters simply cannot scale to the number of requesters in modern real-time systems while still operating at a fast clock speed [12]. As mentioned in Section 2.3.3, as monolithic arbiters increase in size, their maximum clock frequency significantly decreases. The very nature of distributed schemes instead allow them to scale much larger with virtually no decrease to their clock frequency.

The tradeoff of using a distributed approach is a slightly longer WCET due to the increased number of pipeline stages that a memory transaction must cross, although the extra few cycles are insignificant compared with the delay imposed by the memory controller. To hide some of this additional delay, it may be possible to pipeline memory requests, hence masking some of the apparent latency of crossing the memory controller.

The pros and cons of many of these distributed arbitration schemes are explored within Section 2.3.2. To be compatible with the system model, there are a few characteristics which are required of the arbitration scheme:

- The arbiter must tag each request with the requester index from which it originated.
- The arbiter must support routing of a response back to the respective requester based upon the requester index.
- The worst-case delay for crossing "up" the arbiter, $t_{wc}^{arb\uparrow}(r)$ must be known and bounded for a given requester $r \in \mathbb{C}$.
- The delay for crossing "down" the arbiter, $t_{wc}^{arb\downarrow}(r)$ must be known and fixed for a given requester $r \in \mathbb{C}$.
- The delay between two requests for a given requester $r \in \mathbb{C}$ being accepted by the arbiter, $\delta^{arb}(r)$ must be known and bounded.

Of the approaches explored within Section 2.3.2, Distributed TDM [10] can be discounted immediately. Distributed TDM operates by arbitrating each requester at their input to the arbiter and effectively replicates a 1-input TDM scheduler for each requester. Because there can only be one request in-flight per active period, the arbiter does not need to tag each request with the requester index, and broadcasts each response. Each requester can then ascertain whether the response was destined for them simply if they were scheduled in that period. This means that any components "after" the arbiter (i.e. the prefetcher and memory controller) cannot ascertain which requester a request originated from, and the arbiter does not support sporadically pushing data to a specific requester.

Both GSMT [12] and Bluetree [60, 61] tag requests with their originators and hence support routing responses back to a specific requester. Moreover, they have bounded "up" times and constant "down" times, and make it possible to determine the worst-case inter-request time δ^{arb}. Both techniques also allow for work conservation and can fully utilise main memory; Bluetree routes based on backpressure, and by setting the scheduling interval equal to the response time of memory, GSMT can ensure there is always a packet waiting on the input the memory controller on the cycle where it can accept a new packet.

The differentiating factor here is simply a question of scalability. As GSMT assumes that all packets are transmitted in the same instant, each multiplexer need only examine both inputs and pick the one with the highest priority. If one of these packets is a cycle late, the multiplexer will relay both in subsequent cycles, as they both are the "highest-priority" packet in their respective cycles, which may then lead to a packet being lost if the memory controller cannot accept it. As the tree grows larger, the power utilisation and skew inherent with large clock networks may necessitate a move towards a globally asynchronous, locally synchronous architecture [92] across the memory tree, which GSMT cannot support. Instead, Bluetree makes its scheduling decisions at each multiplexer and relies upon backpressure to decide when the memory controller can accept a new packet. Because of this, it is possible to place entire sub-trees within their own clock domain safely. Because of the attractive avenues for future scaling, Bluetree was chosen over GSMT within this work.

Bluetree is made up of a set of 2-into-1 multiplexers, as can be seen within Figure 4.1. An internal block diagram of these multiplexers can be found in Figure 4.2: they comprise of a single input buffer on each side, and an output link to the next stage with flow control. On every cycle in which a packet can be relayed up to the next level (i.e. the next buffer is empty), the multiplexer will pick one of the requests from the input buffers (using the arbiter), then relay this request up to the next stage.

The "down" path from memory back to the requesters is implemented using a single buffer. On each cycle, *without* flow control, the packet in the "down" buffer

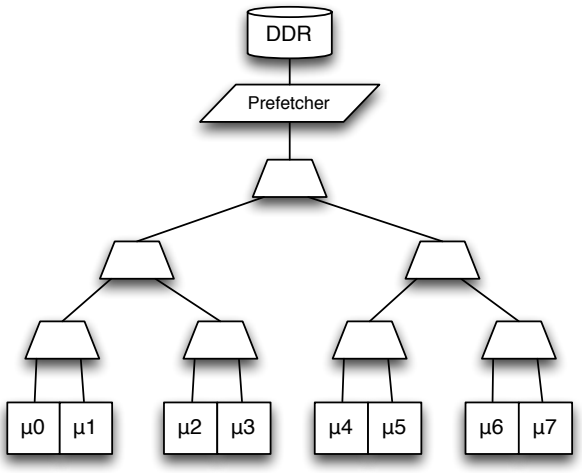


Figure 4.1: Example Bluetree structure for an 8-core system.

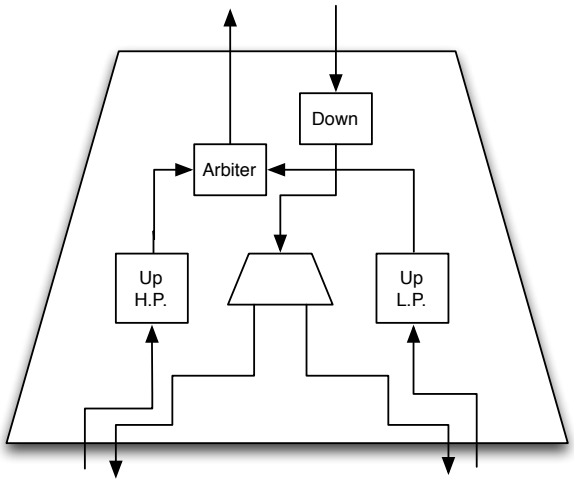


Figure 4.2: Internal block diagram of a Bluetree multiplexer.

is relayed back to the requester which initiated it. In order to prevent stalling the memory controller, the downwards path is defined to be non-blocking; the buffers are only utilised in order to split the critical path and to create a pipeline in order to allow Bluetree to be scalable.

The packet format which is used internally by Bluetree can be found in Figure 4.3, which is used by the requesters in the tree when initiating a request. Returned data follows much the same format, but without the byte enable, priority or size. The usage of each field is listed below:

TYPE: The message type (i.e. read, write, write ack etc).

4 bit	128 bit	16 bit	28 bit	8 bit	8 bit	4 bit	4 bit
Type	Data	BEN	Address	Task ID	CPU ID	Priority	Size

Figure 4.3: Bluetree Client Packet Format.

DATA: Sixteen bytes of read or write data. The size of this field corresponds with the cache line size of the target cache, and the size of a DRAM request when Burst Chop[1] mode is used.

BEN: Byte enable for a write transaction. Each bit in this field corresponds to a byte in the *Data* field, and hence allows a requester to write only a subset of the sixteen bytes to memory.

ADDRESS: The address to read or write from, aligned to a 16-byte boundary. Reads and writes are assumed to operate in a 32-bit address space, of which the least significant four bits will always be zero and are hence omitted.

TASK ID: Field to encode additional data which is returned with the response. Currently unused and exists for future expansion.

CPU ID: The ID of the requesting CPU. This is filled in by each multiplexer on the upwards path (by shifting in a 0 or a 1 for the left and right directions, respectively), and used by the multiplexers on the downwards path to determine where to route the packet.

PRIORITY: Field to encode a priority which may be used by the multiplexers or memory controller. Currently unused and exists for future expansion.

SIZE: Number of additional lines to read from memory. Used to initiate a burst transaction. Since the system model requires that each access is for a maximum of sixteen bytes, this field should be set to zero.

As an example, a standard memory read transaction will begin by the processor creating one of these packets of type *read*, with the *Data* and *BEN* fields left as zero. The *Address* field is then filled in with the address of the data to fetch, and all other fields set to zero. As the packet crosses the multiplexers on the way to the memory controller, each multiplexer will shift the *CPU ID* field left once, then set the least significant bit appropriately. When the packet reaches the memory controller, the controller will issue a request for memory address *Address*, then create a response packet of type *read*, with the same address, *CPU ID* and *Task ID* as the request, then relay it back down the tree. Each multiplexer then inspects the least significant bit

1 DDR3 memory has a fixed burst size of 8 words since DDR3 transfers eight words per cycle. Burst chop mode discards half of this data for applications which do not require the full set of data or to maintain compatibility with DDR2 memories.

of the *CPU ID* field, shifts it right once, and routes the packet appropriately. The returned data can then be used by the target processor.

A write packet then works in an identical fashion, but with the type set to *write* and the *Data* and *BEN* fields set appropriately. The data blocks specified by the *BEN* are then written to the specified address, and a *write acknowledge* packet returned in the same manner as a read, but with no data.

These examples only concern themselves with a single request. If there is a pending request at both inputs to the multiplexer, the multiplexer must decide which to relay. This is done using a simple, composable arbitration scheme. Each multiplexer uses an implicit static priority scheme which, simply by convention, favours the request waiting at the left-hand side of the multiplexer. In order to provide fair and composable arbitration, each multiplexer also contains a "blocking counter" which encodes how many high-priority requests have been given service while there has been a low-priority request waiting. Whenever a high-priority packet is relayed and there is a low-priority packet waiting, this counter is incremented. When this counter reaches a pre-defined value m, the priority ordering is inverted for a single request, allowing the low-priority request to take precedence. Whenever a low-priority request is relayed, this counter is reset back to zero. As an example, if this counter is set as $m = 3$, then a low-priority packet can be blocked by three high-priority ones. In the worst case, a high-priority packet may be blocked by a single low-priority request, assuming that there were three requests ahead of it.

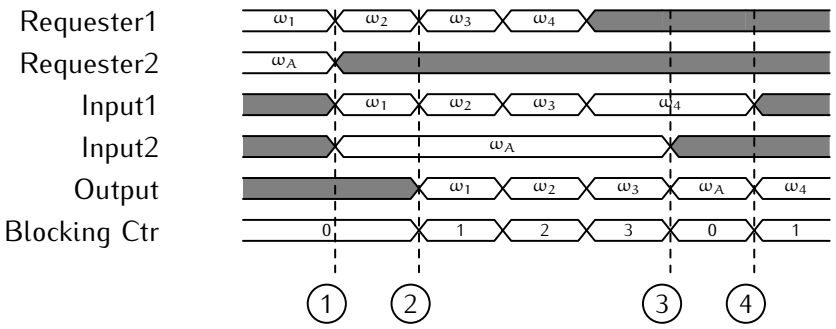


Figure 4.4: A timing diagram to show the blocking behaviour of a Bluetree multiplexer for two inputs. Numbered packets (i.e. ω_1) are from requester 1, lettered packets (i.e. ω_A) are from requester 2. This assumes a blocking factor $m = 3$.

A graphical view of this can be found in Figure 4.4. This shows the requests from two requesters transiting the multiplexer. Each stage can be broken down as follows:

1. Both requesters make a request, ω_1 and ω_A, both of which are accepted into the multiplexer and copied into the multiplexer's input buffers.

2. The request pending at input 1, ω_1, is relayed across the multiplexer and available at the output. Because this was the "high priority" side, the block-

ing counter is incremented. The multiplexer also accepts the next request from requester 1, ω_2. This process repeats for the next two requests (ω_2 and ω_3).

3. The blocking counter is equal to the blocking factor ($m = 3$). Because of this, the request at the low-priority input (ω_A) is given service, blocking the request at the high-priority input (ω_4). The blocking counter is then reset to zero.

4. The blocking counter is now no longer equal to the blocking factor, hence the high-priority side is given service again, as in step 2.

These multiplexers are then chained together into a tree in order to allow the requesters to communicate with the memory controller. In order to be predictable though, it must be possible to derive a set of values for the constraints $t_{wc}^{arb\uparrow}(\omega_n^r)$, $t_{wc}^{arb\downarrow}(\omega_n^r)$ and $\delta^{arb}(r)$, as required from the system model presented within Section 3.3. The derivation of these terms is presented throughout the remainder of this section.

This analysis takes the form of two parts; firstly, the timing behaviour for a single multiplexer and a given blocking factor m is presented in isolation. This derivation will then be extended further to analyse a set of multiplexers in a tree for a given depth.

Single Multiplexer

Recall that Bluetree uses a blocking factor, m to determine how many high-priority requests can be transmitted across the multiplexer before a low-priority request can instead be given service. For simplicity, however, this blocking factor has been re-defined to encode the multiplexer's "cycle length", that is, a low-priority request can be blocked by $m-1$ high-priority requests, then a low-priority request can take service. This does not negatively affect the analysis of Bluetree, it merely simplifies the following sets of equations.

For this reason, the maximum number of times a request ω can be blocked on the way up the tree, $B_{up}(\omega)$ is simple to define, and is presented in Equation (4.1).

$$B_{up}(\omega) = \begin{cases} 1 & \text{High Priority Input} \\ m-1 & \text{Low Priority Input} \end{cases} \tag{4.1}$$

That is, a request on the high-priority input can be blocked by a single low-priority request in the worst case, and a request on the low-priority input can be blocked by $m-1$ high-priority packets. It is now possible to ascertain the worst-case timing for a memory request transiting up the tree to transit the multiplexer

using Equation (4.2). For the system model described within Chapter 3, each component above the arbiter is scheduled based upon back-pressure from the memory controller. Because the memory controller can accept a request every δ^{mem} cycles, it hence follows that a request can emerge from the top of the arbiter every δ^{mem} cycles in the worst-case, hence:

$$t_{wc}^{arb\uparrow}(\omega) = (B_{up}(\omega) + 1) \times \delta^{mem} \tag{4.2}$$

This worst-case situation will occur when a requester has *just* had a request transit the tree, hence the $+1$, and will definitely have its next request blocked. For a high-priority requester, this situation occurs when it has just issued $m - 1$ requests in quick succession, and occurs whenever a request has been accepted for a low-priority requester.

After transiting the multiplexer, the request must then be serviced by all the components "above" the arbiter, like the prefetcher and memory controller, consuming $\hat{t}_{wc}^{arb}$ cycles, then transit back down the tree again (consuming $t_{wc}^{arb\downarrow}$ cycles). Because Bluetree is defined to be non-blocking on the downwards path though, this $t_{wc}^{arb\downarrow}$ is only a single cycle. The final timing behaviour for a single multiplexer is hence presented in Equation (4.3).

$$t_{wc}^{mux}(\omega) = t_{up}^{arb\uparrow}(\omega) + \hat{t}_{wc}^{arb} + t_{wc}^{arb\downarrow}(\omega) \tag{4.3}$$

Multiple Multiplexers

As a set of multiplexers is combined into a large tree, the timing analysis is made more complicated by the fact that an upstream multiplexer (i.e. one closer to the memory controller) can now block multiplexers underneath it. When blocked by the upstream multiplexers, the blocking counters of a downstream multiplexer are not updated at all, since no packets are transiting it. In effect, this causes entire sub-trees to be stalled.

The worst-case blocking conditions are hence when all multiplexers are blocked on the path to memory. This causes packets at the bottom of the tree to experience a huge amount of blocking before being able to transit even the first level of the tree. As an example, even for the highest priority requester, if all multiplexers above it are giving service to the low-priority side, then it will be blocked once due to the root of the tree, then due to the next level, and so on. Depending upon the configuration of the tree, the blocking counter for the root multiplexer may even give priority to the low-priority side again before the packet has even transited the first level of the tree.

This blocking pattern makes it difficult to derive a single equation to model the behaviour of the tree. Instead, three piecewise functions are defined to model the behaviour of each multiplexer. Moreover, to simplify these equations, each "block" is assumed to only be of a single cycle, and is then later multiplied by δ^{mem} in order to derive a final value. Finally, the concept of a "priority path", P is introduced, which encodes the path from the bottom of the tree to the root which a packet transits. As an example, if $P = \{H\}$, a requester is connected to a single multiplexer on the high-priority side. $P = \{H, L, L\}$ defines a system where a requester is connected to the low-priority side of one multiplexer, which is connected to the low-priority side of another, which is finally connected to the high-priority side of the root multiplexer. Finally, $P(l) : l >= 0$ defines the priority level at level l, where $l = 0$ is a packet which is above the root multiplexer, $l = 1$ is entering the root multiplexer and $l = n$ is the lowest level of the tree.

The three functions are as follows:

- $C_l^P(t)$: The current internal cycle for the multiplexer at level l is relation to the current global time t, given a priority path P.
- $B_l^P(t)$: Specifies whether the input to the multiplexer at level l given a priority path P will be blocked at time t.
- $\hat{B}_l^P(t)$: Specifies whether a multiplexer at level l will be blocked by those multiplexers above it at time t, given a priority path P.

And are defined as follows:

$$C_l^P(t) = \begin{cases} 0 & t = 0 \\ C_l^P(t-1) & \hat{B}_l^P(t-1) \\ C_l^P(t-1) + 1 & !\hat{B}_l^P(t-1) \end{cases} \tag{4.4}$$

$$B_l^P = \begin{cases} True & C_l^P \bmod m = 0 \wedge P(l) = L \\ False & C_l^P \bmod m = 0 \wedge P(l) = R \\ False & C_l^P \bmod m \neq 0 \wedge P(l) = L \\ True & C_l^P \bmod m \neq 0 \wedge P(l) = R \end{cases} \tag{4.5}$$

$$\hat{B}_l^P(t) = \begin{cases} False & l = 1 \\ True & B_{l-1}^P(t) \\ \hat{B}_{l-1}^P(t) & !B_{l-1}^P(t) \end{cases} \tag{4.6}$$

Equation (4.4) dictates the "internal cycle" of the multiplexer. This is a logical counter which is advanced whenever the multiplexer is *not* blocked by any multiplexer above it. Since each memory access is assumed to be a single cycle for now, this effectively records the blocking counter for that multiplexer, taking upstream blocking into account. Equation (4.5) then uses this definition to decide whether a packet being relayed from a client with a given priority path will be given service in any given cycle. Finally, Equation (4.6) dictates whether a multiplexer at level l will be blocked by an upstream multiplexer in a given cycle. Of course, the top level multiplexer will never be blocked by anything above it (hence the $l = 1$ case), and otherwise it will be blocked if the above multiplexer is currently blocked.

This worst-case blocking can then be ascertained by using the $\hat{B}^P(t)$ function. A function $L_l^P(t)$ can be defined which computes how far a packet, created at level l has travelled at time t for a given priority path P.

$$L_l^P(t) = \begin{cases} l & t = 0 \\ L_l^P(t-1) & B^P_{L_l(t-1)}(t) \\ L_l^P(t-1) - 1 & !B^P_{L_l(t-1)}(t) \end{cases} \tag{4.7}$$

That is, a packet begins at level l and can only progress up the tree if the current multiplexer is not blocked in that cycle. The worst-case blocking time, $t_{wc}^{arb\uparrow}$ is then:

$$t_{wc}^{arb\uparrow}(\omega) = \delta^{mem} \times \min_{t=0}^{\infty} t : L_l^P(t) = 0 \tag{4.8}$$

Proc. Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
$m = 2$	30	30	30	30	30	30	30	30
$m = 3$	15	18	24	32	29	33	47	60
$m = 4$	11	16	26	39	28	44	71	114
$m = 5$	10	17	27	50	33	58	102	195
Proc. Index	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
$m = 2$	30	30	30	30	30	30	30	30
$m = 3$	30	36	48	63	57	66	93	120
$m = 4$	32	48	76	116	84	132	212	340
$m = 5$	40	65	105	200	130	230	405	780

Table 4.1: Worst-Case blocking across a 16-core tree, measured in number of blocks.

Which finds the lowest value of t such that the memory request has just been emitted from the root multiplexer. Example blocking figures can be found in Table 4.1. Because it has been previously assumed that each memory request was a single cycle, it is hence multiplied by δ^{mem} to derive the actual number of cycles

for which it is blocked. Finally, since the downwards path on Bluetree is defined to be non-blocking, $t_{wc}^{arb\downarrow}$ is simply as follows:

$$t_{wc}^{arb\downarrow}(\omega) = l \tag{4.9}$$

These values can then be substituted into Equation 4.3, and hence the worst-case response time for a memory transaction is as follows:

$$t_{wc}(\omega) = (\delta^{mem} \times \min_{t=0}^{\infty} t : L_l^P(t) = 0) + t_{wc}^{mem} + l \tag{4.10}$$

4.2.2 Requesters

There are two different types of requesters which are used while evaluating the behaviour of the prefetcher in this system. Firstly, standard processors are used to ascertain the average-case behaviour of the prefetcher when used with the composable memory tree. Afterwards, a set of hardware traffic generators are used to emulate the worst-case conditions on the tree such that prefetcher evaluations under worst-case conditions can be investigated. The details of each of these are provided below.

Processors

To be evaluated within a real-time context, it must be possible to construct a model of a processor which defines the timing behaviour of the processor in the worst-case. Given such a model, a given task can then be evaluated against the worst-case processor model to ascertain the worst-case behaviour for the task when run on said processor. Because tasks running on a processor require access to shared memory, the processor model can then be combined with a worst-case model of the memory system, then the combined model can be used to bound the execution time of a section of code. In addition, it also should be possible to extend the processors themselves to perform any architectural changes which are required to implement the prefetcher.

Xilinx's Microblaze [88] processors were hence chosen for this purpose. This is a reasonably simple processor for which the latencies of each instruction are well defined and which does not contain any complex features such as out-of-order execution or superscalar pipelines. These characteristics allow a reasonably simple model of the processor to be constructed such that static analysis can be performed. In addition, while these processors *do* support features which are typically unpredictable (such as a branch predictor), the processor can be configured to disable them at design time. For future expansion, it uses a low-overhead, simple and

well defined bus to connect the processor to memory, hence custom caches can be easily connected without any significant performance overhead.

In order to fit the model defined in Section 3.3 a custom cache has been attached to the processor which allows data to be pushed in from an external source. If a prefetch response is delivered into the cache, the cache is updated to store the returned data into its internal storage such that it can be requested by the processor. In order for prefetch hit feedback to be supported (so that the cache can notify the prefetcher of a "useful" prefetch), each cache line also contains a flag to record whether the data stored in that line is the result of a prefetch, which is set when a prefetch is stored into the respective line and cleared when the data from a demand read is stored.

When the processor issues a read request to the cache, the cache first checks whether the data corresponding to the address is already resident. If so, it immediately responds with the relevant data. If the requested data line is tagged as "prefetched", the cache also initiates a "prefetch hit" request and issues it to the arbiter, clearing the "prefetched" line in the process to prevent multiple hit packets being issued for a single line of data. If the data is not already resident in cache, the cache will issue a read request for the accessed address to the arbiter and block until a response has been delivered. This must be responded to with a single read response; if spurious read responses are received, the cache may yield undefined behaviour.

Upon receipt of a write request, the cache will again check if the address to be written is already resident in cache. If so, it is updated with the data to be written, and the "prefetched" flag cleared (if set). Regardless of whether the data was resident in cache, the write request is forwarded to the arbiter (i.e. it is a write-thru cache) and the cache again blocks until the write acknowledge response is received.

Traffic Generators

Referring back to Section 2.1, in order to determine the worst-case execution time for a task analytically, the system must have been able to observe the worst-case conditions of the system. As many processors place limits on the number of simultaneous transactions which may take place at once (i.e. the processors in use can only issue a single read and single write with no reordering), processors are unsuitable at simulating the worst-case tree conditions.

Instead, hardware traffic generators are used in order to simulate these worst-case conditions. These are implemented as simple tree clients which simply issue a new read request on every single cycle for which their output queue is empty. In order to prevent the memory controller from being able to simply keep the same DRAM row open to satisfy most requests, the traffic generators read from sequential memory locations.

By fetching from sequential memory locations like this, the traffic generators will also exercise the worst-case conditions of the prefetcher. Because only a simple stream prefetcher is used for these experiments, the system can be easily configured to prefetch for the traffic generators by making the traffic generators fetch sequential memory locations, or configured to not prefetch for the traffic generators by using a longer stride (i.e. every other memory location).

The traffic generators consume all incoming data regardless of the message type, but simply throw it away. The traffic generators also do not wait for a response before sending the next memory request, they request as fast as possible. By doing this, this ensures that all buffers between each traffic generator and the memory controller remain as full as possible. Moreover, it will cause the "blocking counters" in the multiplexers to always operate as if the tree was at full utilisation, and hence will cause the worst-case blocking across the memory tree.

4.2.3 Prefetcher

In order to be utilised within a real-time system, the prefetcher itself should be predictable. As discussed within Section 3.3.4 however, the current system model does not provide a rigorous definition over *what* the prefetcher should fetch and *when*. No current prefetchers from Section 2.4 do define such a model of prefetch however, and as such these problems are explored further within Chapters 5 and 6 in order to create a "real-time" prefetcher.

To simply evaluate the effects which a prefetcher can have in the worst-case, a simple 1-lookahead stream prefetcher was used. The rationale for using such a simple design is that this can capture many of the patterns for tasks used within the evaluation; code is inherently streaming and many benchmarks operate over serial data. It is also trivially possible to construct the pathologically worst and best cases for evaluation; the best case comes when a task accesses perfectly sequential data, and the worst when it accesses enough "serial" data in a row for the prefetcher to begin issuing, then moves to another address and starts again.

Of course, other prefetching schemes may give better results, although may not be able to fit into the system model or are overly complex. Stride prefetching, for example, can detect the stride of accesses, but requires knowledge of the program counter when a load instruction is issued, hence is unsuitable for a memory-side prefetcher where this information is not available. Markov prefetching requires huge tables for its storage which will consume what fast storage is already available. Approaches which require knowledge of the program flow typically utilise annotations stored in memory, hence further compound the memory access penalty and may cause even more blocking to occur while these annotations are being fetched from memory.

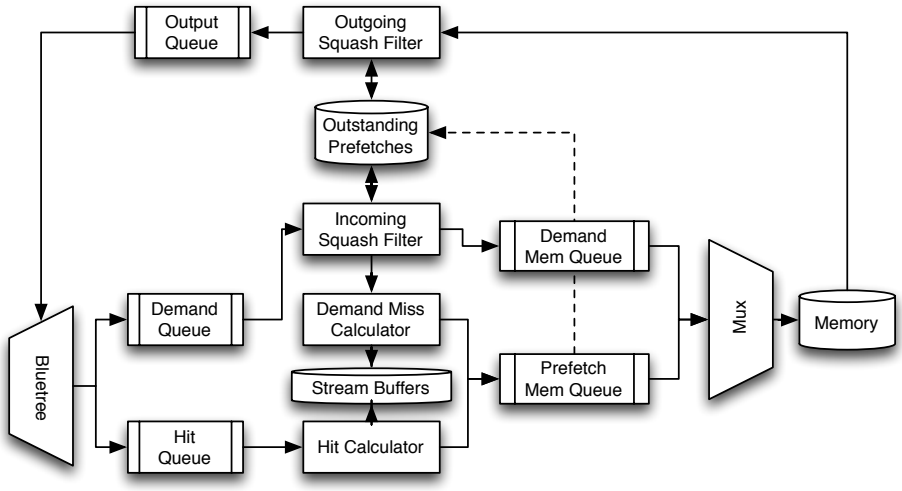


Figure 4.5: Block diagram of the internals of the prefetcher.

While the current system model does not make any assertions over what the prefetcher will attempt to fetch, Section 3.3.4 does place requirements on the timing behaviour of a normal memory request through the prefetcher and how prefetches and memory accesses can be coalesced. These must be considered when constructing the prefetcher, and worst-case timing figures provided for $t_{wc}^{pf\uparrow}$ and $t_{wc}^{pf\downarrow}$. To fit this model, a prefetcher design was created, a block diagram of which can be found in Figure 4.5. Each block is explained in detail below:

BLUETREE: The Bluetree adaptor first takes all requests coming in from Bluetree and splits them into two queues: the demand queue for standard memory accesses, and the hit queue for hit feedback.

DEMAND QUEUE: Stores standard memory accesses from the requesters on Bluetree.

HIT QUEUE: Stores hit feedback generated from the requesters from previous prefetches.

INCOMING SQUASH FILTER: Inspects incoming memory requests to ascertain whether they can be coaleasced with an outstanding prefetch. If so, the outstanding prefetch table is updated and the memory request is discarded.

DEMAND MISS CALCULATOR: Inspects the stream buffers to determine whether the current miss is part of a stream. If so, the stream entry is updated and a prefetch dispatched for the "next" address. If not, the demand miss is used to create a new entry in the stream buffers.

HIT CALCULATOR: Does the same as the demand miss calculator, but inspects the stream buffers based upon the hit notification. This block will never insert a new stream entry if the hit notification does not correspond to an existing stream.

DEMAND MEM QUEUE: Stores memory requests which are to be queued to go to memory after they have been checked against the outstanding prefetch table and forwarded to the demand miss calculator. This queue is then multiplexed with the prefetch queue with a static priority.

PREFETCH MEM QUEUE: Stores any pending prefetches generated by the demand miss or hit calculators. When an item is popped from this queue, it is also entered into the "outstanding prefetch table".

OUTSTANDING PREFETCH TABLE: Stores all outstanding prefetches, and is to be used to ascertain whether a demand access can be coalesced with a prefetch.

OUTGOING SQUASH FILTER: Only used for returned prefetches. The filter checks the outstanding prefetch table for the current prefetch. If the prefetch is marked as a "squash", the prefetch is instead returned as a read. The line in the outstanding prefetch table is then removed.

The prefetching scheme used is a simple stream prefetcher, hence it attempts to correlate a stream on the pattern of A, $A+1$, $A+2$ etc. In order to do this, it snoops all read requests before they are forwarded on to the memory controller in order to ascertain when a streaming access is taking place, and also receives "prefetch hit" requests from the requesters in order to carry on prefetching an already established stream.

To do this, incoming requests are first filtered into one of two streams: a demand queue for reads and writes, and a hit queue for any hit feedback from the processors. Read requests are then passed on to the incoming squash filter which checks the outstanding prefetch table to check whether there is an outstanding prefetch for the same address that is currently being requested. If so, the table is updated to mark the prefetch as "coalesced" and the read request discarded in order to fulfil the requirement that the prefetcher should coalesce demand accesses and their respective prefetches. If there is not an outstanding prefetch for the same address, the request is forwarded to the demand queue and eventually issued to the memory controller. Write requests are issued through the same functional units, but are ignored and simply forwarded on unmodified.

If the request cannot be coalesced, it also enters the demand miss calculator. This component inspects the stream buffers to ascertain whether the demand request is part of a stream (i.e. if address $A-1$ has also been observed). If so, a prefetch for address $A+1$ is initiated, and the table updated to record address $A+1$ as the last observed. This prefetch is then multiplexed onto the output queue with a configurable static priority. If not, a new table entry is added recording address A as the last observed address in round-robin order. Pseudocode describing the operation of the demand miss calculator can be found in Listing 4.1.

A graphical view of this can be found in Figure 4.6. A breakdown of the steps is as follows:

```
#define STREAM_BUFFERS 8
struct streamBufferEntry {
    addr_t last_address;
    bool valid;
};

streamBufferEntry streamBuffers[STREAM_BUFFERS] = {0};
int lastStreamBuffer = 0;

void pfMiss(req) {
    // Search for the last address
    int streamBufId = -1;
    for(i = 0; i < STREAM_BUFFERS; i++) {
        if(streamBuffers[i].valid &&
           streamBuffers[i].last_address == req.address - 1) {
            streamBufId = i;
            break;
        }
    }

    if(streamBufId != -1) { // Found a stream. Update and issue.
        // Update stream buffers. The newly prefetched line is
        // the ''last observed'' data
        streamBuffers[streamBufId].address = req.address + 1;
        issuePrefetch(req.address + 1);
    }
    else {
        // Replace an entry with a new stream buffer
        // Don't increment here, we didn't issue a prefetch
        streamBuffers[lastStreamBuffer].last_address = req.address;
        streamBuffers[lastStreamBuffer].valid = true;
        lastStreamBuffer = (lastStreamBuffer + 1) % STREAM_BUFFERS;
    }
}
```

Listing 4.1: Pseudocode description of how the prefetcher handles a demand miss.

1. The request, ω_1 enters the prefetcher and is stored in the demand access queue.
2. The request then enters the incoming squash filter when it is checked against the list of outstanding prefetches.
3. The request could not be coalesced, so it moves into both the demand miss calculator, which attempts to correlate it to an existing stream, and into the demand output queue.
4. The request is issued to the memory controller.
5. The demand calculator has finished processing. In this case, the access could be correlated to an outstanding stream, so a prefetch is initiated.
6. The prefetch is then issued to the memory controller.

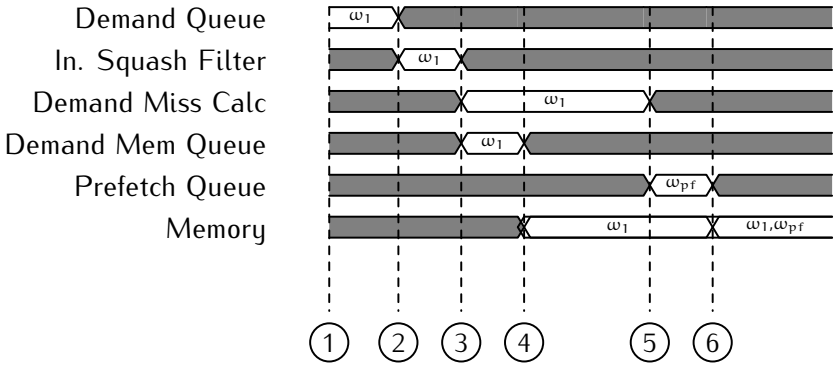


Figure 4.6: Timing diagram showing a single request, ω_{hit}, transiting the prefetcher. This also causes a prefetch, ω_{pf} to be initiated.

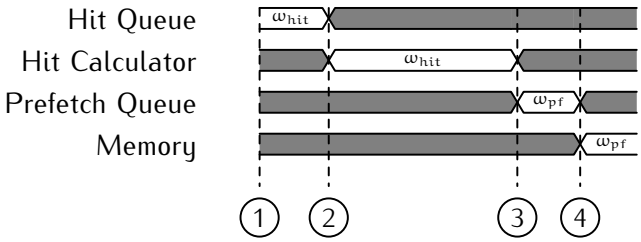


Figure 4.7: Timing diagram showing a single prefetch hit feedback request, ω_1, transiting the prefetcher. This also causes a prefetch, ω_{pf} to be initiated.

If the prefetcher observes a "prefetch hit" request, it is forwarded to the "hit calculator". This works in the same way as the demand miss calculator, but searches the table for a stream with address A as the last access. If found, a prefetch is again initiated for address $A + 1$ and the table updated to reflect this. The prefetch hit packet is then discarded and is not forwarded to the memory controller or acknowledged. Pseudocode for the operation of the hit calculator can be found in Listing 4.2, and a timing diagram of the request moving through the prefetcher can be found in Figure 4.7, and works in much the same way as that for a demand access:

1. The prefetch hit request, ω_{hit} enters the prefetcher and is placed into the prefetch hit queue.
2. The prefetch hit then moves into the hit calculator.
3. Again, the hit could be correlated to a prefetch stream, hence a prefetch ω_{pf} is initated. If a stream could not be correlated, no prefetch would be initiated, and the timing diagram would end here.
4. The prefetch ω_{pf} is then issued to the memory controller.

Finally, all data returned from the memory controller is snooped by the "outgoing squash filter". If the returned data is a prefetch, the "outstanding prefetch table" is checked to determine whether a demand access has arrived for the same

```
#define STREAM_BUFFERS 8
struct streamBufferEntry {
    addr_t last_address;
    bool valid;
};

streamBufferEntry streamBuffers[STREAM_BUFFERS] = {0};
int lastStreamBuffer = 0;

void pfHit(req) {
    // Search for the hit address
    int streamBufId = -1;
    for(i = 0; i < STREAM_BUFFERS; i++) {
        if(streamBuffers[i].valid &&
           streamBuffers[i].last_address == req.address) {
            streamBufId = i;
            break;
        }
    }

    if(streamBufId != -1) { // Found a stream. Update and issue.
        // Update stream buffers. The newly prefetched line is
        // the ''last observed'' data
        streamBuffers[streamBufId].address = req.address + 1;
        issuePrefetch(req.address + 1);
    }
}
```

Listing 4.2: Pseudocode description of how the prefetcher handles a prefetch hit.

address (i.e. a demand miss has been coalesced with the prefetch). If so, the prefetch is re-written as a standard read response. In all cases, the prefetch is then removed from the outstanding prefetch table. Standard read responses and write acknowledgements pass through this unit un-modified, as required in Section 3.3. Again, an example of this can be found in Figure 4.8, and is described below:

1. A prefetch, ω_{pf} has been initiated and placed into the prefetch queue. This updates the outstanding prefetch table. A demand access ω_1 also arrives at the incoming demand access queue.

2. The prefetch is issued to the memory controller. The demand access moves into the incoming squash filter.

3. The squash filter detected that the demand access could be coalesced with the prefetch, hence discarded the demand access and updated the outstanding prefetch table to reflect that the coaleasce occured.

4. The prefetch returned from memory and entered the outgoing squash filter.

5. The outgoing squash filter detected that the prefetch had been coaleasced, so re-wrote ω_{pf} to be able to satisfy ω_1 and placed it in the output queue.

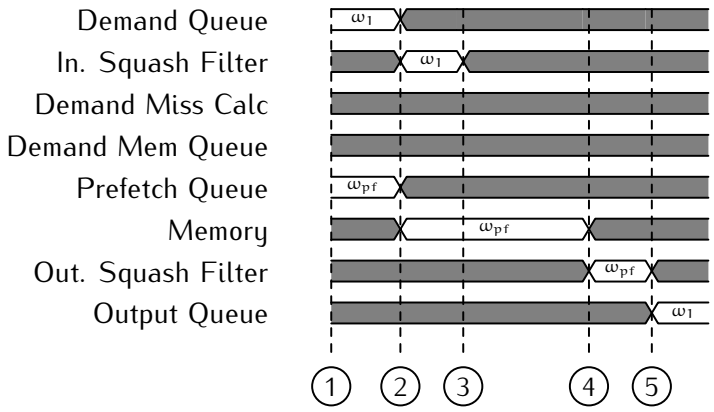


Figure 4.8: Timing diagram showing how a prefetch, ω_{pf} can be coaleasced to a demand access ω_1

For experimentation, many of the parameters of the prefetcher (i.e. the depth of the queues and the size of the prefetch buffers) can be tuned at design-time. For these experiments, the queues were implemented as two-entry queues with cut through (hence an entry pushed into an empty queue can be popped from the queue in the next cycle). For this prefetcher design though, the size of the queues is mostly irrelevant; they are simply used as buffers which can support simultaneous enqueue and dequeue of data items to prevent starvation on the prefetcher's input, while breaking up the critical path through the prefetcher.

For these experiments, the size of the stream buffers was set to be eight entries for each processor, for each interface (i.e. instruction and data) for each processor, requiring a total of 256 buffer entries, each storing an address and a valid bit. In order to reduce the number of hardware registers required, these stream buffers were implemented as FPGA block RAMs, making them reasonably cheap to store. The main overheads associated with the stream buffers arises from the lookup logic; to optimise performance, each of the entries in the stream buffer is looked up in parallel, effectively making the stream buffers for a given requester a content-addressable memory. For the purposes of these experiments, eight stream buffers is a good tradeoff between the complexity of the lookup logic and the number of streams that can be tracked at once.

In terms of timing behaviour, each squash filter takes a single cycle to cross, while each "calculator" takes three cycles to cross, but all blocks are pipelined. Each block can operate in parallel with every other block except for the "calculators"; while the three cycle crossing time of these is pipelined, but only one calculator can accept a new request in each cycle (hence no two requests can be in stage 1 at any time). This is implemented using a simple priority ordering to prioritise demand memory accesses. The calculators do not block on the output queue, if the output queue is full, any generated prefetches are discarded. Finally, each of the queues are two-entry FIFOs.

The prefetcher will only relay a request when the input queue of the memory controller has space, and hence each functional unit uses back-pressure to throttle itself. The worst-case crossing time is therefore expressed in terms of δ^{mem}, or the worst-case time between two requests being accepted by the memory controller. Because the critical path for a memory request uses two queues and a squash filter, there can be five blocks under worst-case conditions and hence $t_{wc}^{pf\uparrow} = 5 \times \delta^{mem}$. The "down" path is again defined to be non blocking (and hence cannot have any backpressure), therefore $t_{wc}^{pf\downarrow} = 3$.

As stated in both Section 3.3 and within this chapter, all of these figures assume that the prefetcher doesn't actually *do* anything. If prefetches are initiated by the prefetcher, any demand accesses may be blocked while the prefetch is being serviced by the memory controller. Because the behaviour regarding *when* the prefetcher operates is not currently well defined, this is impossible to bound. Again, this is a problem which is explored further within Chapter 5.

4.2.4 Memory Controller

The memory controller used in the platform is a standard Xilinx memory controller [93]. This memory controller is a DDR3 memory controller exposing a 2GB address range for both read and write transactions which are accessed with a 28-bit address, aligned to sixteen-byte boundaries. Upon receiving a read request, the memory controller will retrieve the data from main memory and respond with the relevant data. Any write requests are written to the DDR memory, and a write acknowledgement delivered back to the relevant requester.

Normally, the memory controller supports transaction reordering by storing a queue of requests, then servicing the requests from this queue which match the currently open row first. If no requests match the currently open row, then the row is closed and precharged, and another request chosen. In order to fulfill the requirement that the memory controller must return the results of requests in the same order in which they were issued though, the memory controller also supports "strict" mode, which forces all requests to be serviced in the same order in which they arrived which was used for these experiments.

Because there is no reordering of transactions, the timing of a memory transaction has a deterministic upper bound, which is simply the time taken to precharge the last command, followed by the time taken to open the required row and column required.

4.3 EVALUATION METHODOLOGY

In order to evaluate the potential effects of using a prefetcher within a real-time system, two things need to be ascertained. Firstly, the effect on the actual execution time must be evaluated to demonstrate the potential improvements by using a prefetcher with a "standard" task-set under average-case conditions. Of course though, to reason about the behaviour of the prefetcher within a real-time system, the *worst-case* execution times also need to be evaluated to determine if the prefetcher causes a performance detriment under worst-case conditions, and by how much.

In order to do so, a hardware design must be determined for the "average-case" systems, and also a way of simulating the "worst-case" conditions on the memory tree. After these two problems have been solved, a set of tasks must also be determined which can be used to evaluate the system, for both synthetic (i.e. "ideal" and "non-ideal") and "real-world" tasks.

4.3.1 Hardware Setup

The platform was evaluated using two different hardware configurations, each using a sixteen-input tree where $m = 4$ (i.e. a low-priority packet can be blocked by at most three high-priority packets). The actual choice of the number of requesters to use has little relevance to the work; the final system is intended to be composable and hence the results depend only on the number of cycles a packet can be blocked. Instead, this choice of sixteen requesters and blocking factor allows for the exploration of a good range of latencies, while not being too complex to reason about. Moreover, from an implementation standpoint this number of requesters can be synthesised on a modern FPGA easily while leaving room for extra hardware in the later stages of this thesis.

The platform was evaluated for average-case performance by utilising sixteen Microblaze [88] processors at the leaves of the tree. Identical software was run on each processor, and the execution time of the benchmark was taken over a number of iterations from each processor. The prefetcher is then toggled on and off to determine what impact the prefetcher makes to the execution time of the task.

In order to then asceratin the effects of the prefetcher on the worst-case performance of the systems, another set of systems are built which incorporate a single processor attached to one of the leaves of the memory tree, with fifteen traffic generators connected to the remainder of the leaves. These traffic generators issue a memory request on every cycle where they are given service by the arbitration tree without waiting for a response. Any responses delivered from the memory controller are simply consumed and thrown away.

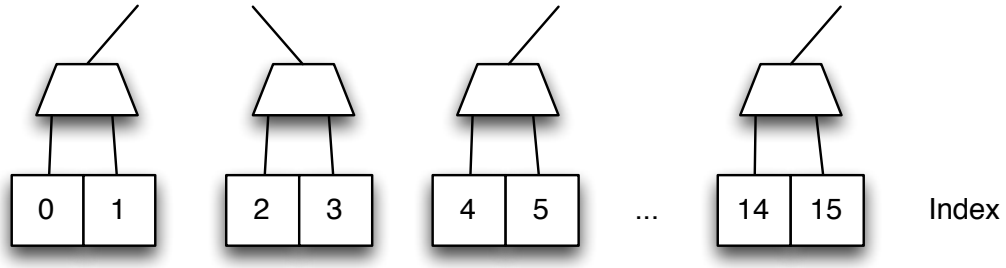


Figure 4.9: Graphical view of "processor index" on Bluetree.

The worst-case analysis of Bluetree presented within Section 4.2.1 assumes that all other buffers in the tree are always full with requests from other requesters, and that the memory controller always has work to do. This can be emulated using these traffic generators; because they issue on every possible cycle, it will ensure that all buffers in-between the requesters at the leaves of the tree and the memory controller at the root will remain full and cause maximal blocking to occur.

One issue with this analysis of Bluetree is that it assumes the absolute worst-case behaviour on each access. As an example, for a single memory request issued to a single multiplexer on the "high-priority" side, it assumes that it will be blocked by a "low-priority" request (i.e. the blocking counter is equal to m). For a set of memory requests issued to the "high-priority" side of the multiplexer though, this is not necessarially the case; in a set of m memory requests, at least $m-1$ requests will not be blocked by a "low-priority" packet. For simplicity, the current worst-case system analysis does not take this into account when analysing the time taken to issue a set of memory requests, as to do so requires accurate knowledge of the current state of the blocking counters and the requests issued by other requesters. Of course, it is possible to integrate this into the worst-case analysis of Bluetree if required, although this extension is currently left as further work.

For this reason, the worst-case performance of the benchmarks and traffic generators within this "worst-case" system is better than the performance of the tasks when analysed against the Bluetree model presented in Section 4.2.1. Ultimately though, flooding the tree with requests from all other requesters should maximise the blocking as much as possible and be very close to the actual worst-case behaviour of the memory tree.

In these systems, the behaviour of the task from the perspective of each tree input is realised by moving the processor to each possible slot on the tree, filling the other fifteen inputs with the traffic generators. Each input to the tree is given an index from left to right, where index 0 is the leftmost slot and index 15 is the rightmost. A graphical view of this can be found in Figure 4.9. Because of the arbitration scheme used, the response time of a memory transaction does not monotonically increase with the processor index, and hence the processor index

cannot be viewed as a "priority". The actual response times corresponding to each index can be found in Section 4.2.1.

4.3.2 Workload Generation

To go along with the hardware, two different classes of workload are evaluated. The first of these is a traffic generator intended to demonstrate the best and worst case behaviours of the prefetcher to show both the potential performance improvement and detriment which can be caused by the prefetcher. The second of these is a set of benchmarks intended to show the impact of the prefetcher on a set of representitive workloads for embedded or multimedia systems.

The software traffic generators simply issue a memory request, await the response, the waits for a number of processor cycles before dispatching the next request. Any data which is read is simply thrown away; the purpose of the set of reads is simply to measure the latency of a set of memory requests. These reads are issued for sequential memory lines in order to provide the best conditions possible for the prefetcher, with a configurable "hold-off" period between each access to simulate differing bandwidth requirements.

This traffic generator is then evaluated using a delay between 5 and 500 cycles, in increments of 5 (because the delay loop takes five cycles to execute). Each combination of delay and processor index is then run 20 times and recorded. It is expected that by increasing the hold-off period, the prefetcher will have more time to be able to operate and hence more prefetches will be able to be dispatched and complete fully, hence fewer prefetches will be coalesced with demand misses.

The rationale for using such a traffic generator is that it can be used to demonstrate the absolute best and worst case performance differences for an application with a given average request rate. As an example, a software traffic generator with a fully sequential access pattern will yield the greatest potential from the inclusion of the prefetcher in the systems, and demonstrate where the limits for the prefetcher are in the best case. A software traffic generator can also be programmed with a completely non-sequential traffic pattern to demonstrate the "worst-case" effect, when the prefetcher is not able to perform any prefetches on behalf of the running task, but may be causing extra interference by fetching data for other tasks.

The results from these traffic generators can then be carried forward into subsequent chapters to propose modifications to the prefetching system in order to not damage the worst-case execution time through prefetching, and to improve the worst-case execution time through prefetching. Of course, this only presents the best and worst cases with no data in-between (i.e. fetching x sequential locations, then a non-sequential fetch etc). While these data points would be interesting, this work mainly concerns itself with worst-case performance of the system, and

so the "non-prefetchable" traffic generator is of most use for demonstrating the prefetcher's behaviour in the worst-case, with the "fully prefetchable" traffic generator being used to demonstrate if prefetching can actually yield any performance benefit in the best case.

While the full spread of traffic patterns has not been investigated, certain patterns representive of typical embedded and multimedia workloads have been investigated to ascertain the prefetcher's impact on "real" applications. In order to facilitate this, a selection of benchmarks have been used from the TACLeBench suite [94]. These incorporate a number of benchmarks from many different sources (e.g. Malardalen [95], MiBench [96] and Mediabench [97]), but any required data has been inlined into the program in order to run the benchmarks without a filesystem (or filesystem emulation layer). Moreover, static loop bounds have already been ascertained for each benchmark in order to ease static analysis of the tasks in later chapters. Many of the benchmarks contained within this suite are generally regarded within the embedded and multimedia research communities as benchmarks representitive of real tasks running on these systems.

By inlining the files into the benchmarks, the benchmarks have been reduced to single-path applications, easing both the static analysis and asserting that full coverage can be attained for the measurement based approaches easily. These benchmarks are then executed a number of times on both the sixteen processor system and the "worst-case" systems, and the results taken.

4.4 RESULTS – LOW PRIORITY PREFETCHING

The naïve intuition to ensure that the prefetcher does not have an impact on the execution time of a task is to simply prefetch data at a low priority level. This approach has two main problems; firstly, while it will work in a system such as Bluetree which does not have a concept of an "active period", it will not work in any arbitrated system which uses active periods without modification. This is because the prefetcher may observe some time at which there are no incoming memory requests, but it is not able to distinguish whether there is actually no request in-flight, or if the arbiter is simply waiting for the next active period before initiating another memory request without global clock synchronisation.

4.4.1 Worst-Case Conditions

When the hardware traffic generators are used on a system where the prefetcher is operating at low-priority and worst-case conditions are being simulated, the problem arises that prefetches can be blocked indefinitely as demand traffic enters the prefetcher and hence prefetches may actually never be dispatched. Figure 4.10

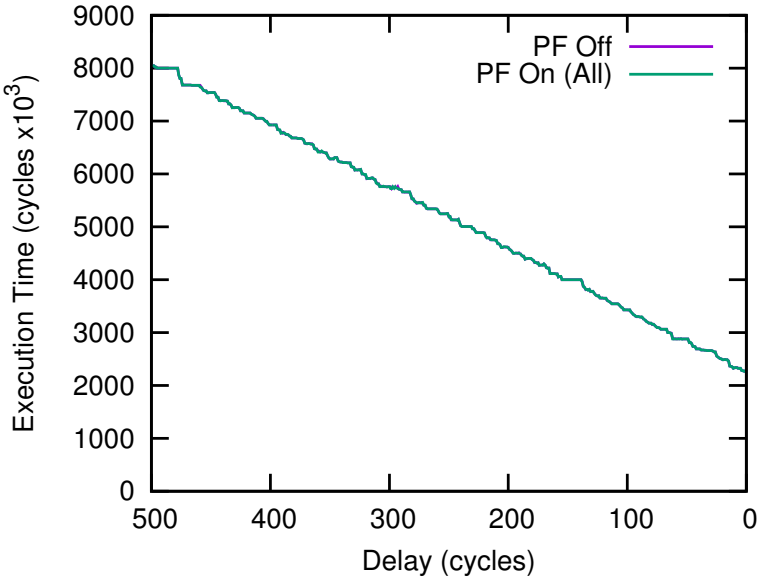


Figure 4.10: Graph of execution time of a software traffic generator with and without the prefetcher in a "worst-case" system.

shows the execution time for a software traffic generator running on such a system, fetching 4096 memory blocks with a delay between initiating transactions between 0 and 500 cycles.

Because Bluetree is work-conserving, the prefetcher can never operate; it is always blocked because the tree is always backlogged in the worst-case conditions and hence another memory request from a traffic generator is always available. In this case, the prefetcher can never initiate a request. While this is the results for a processor in index 1, all other results show the same form due to the issues listed above, and equally for all benchmarks in the TACLeBench suite and hence have been omitted from this thesis.

4.4.2 Average-Case Conditions

Traffic Generators

Firstly, a the software traffic generators were executed on sixteen processors within the system. This measured the amount of time taken for sixteen of these traffic generators to each fetch 4096 memory blocks from memory, each waiting between 0 and 500 cycles between each access. The results of this experiment can be found in Figure 4.11. This shows improvements throughout for most of the delay values up until a delay of around 80 cycles. Throughout this period (i.e. 80 - 500 cycles), the apparent speedup is of the region of 4-20%. With higher delays, the apparent speedup is of course lower, as the measured execution times also include the delay factor which dominates the execution time for higher delays.

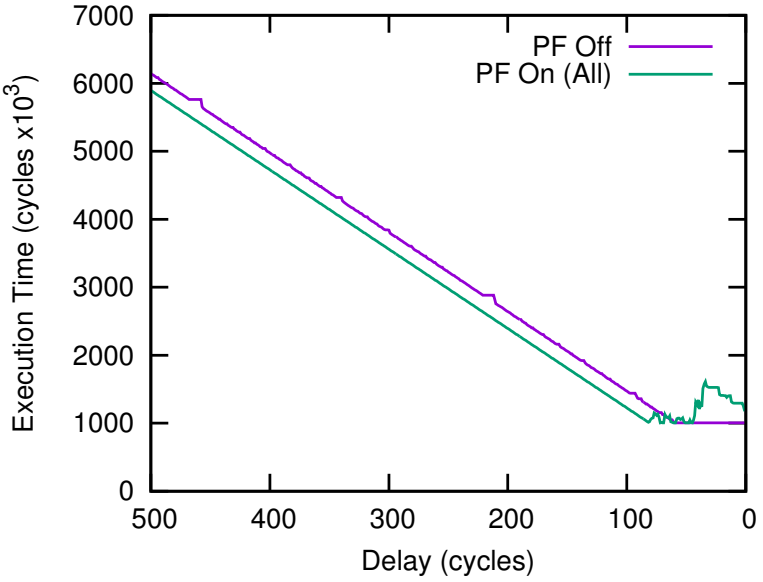


Figure 4.11: Graph of execution time of sixteen traffic generators running on sixteen processors with the prefetcher both enabled and disabled.

Below a delay of 80 cycles, the execution time with the prefetcher disabled shows no change with respect to the delay factor. This is the side effect of the "blocking counters" used on Bluetree; because of the amount of blocking which some processor indices may experience is reasonably high (and higher than the inter-request delay) and that the blocking counters are only updated when a packet is relayed, it may lead to a situation where two requests issued in a given time interval will actually be relayed at the same instant, regardless of when in the interval the request was issued. This effect also appears in a much clearer form in the worst-case systems presented in Section 4.5.1.

In addition, below a delay of around 80, utilising prefetching causes a significant detriment to the execution time of the task, translating to a slowdown of up to 50% in some cases. There are a few causes of this. Firstly, the measured execution time of the traffic generator must wait for all processors to request 4096 words of data, although because of the non-uniformity of the tree some processors may finish before others. At low delays though (i.e. below 50), the pressure caused by all requesters causes the prefetcher to not be able to operate at all, because there are always demand accesses available which take precedence over prefetches. After some of these requesters complete however, the prefetcher is able to initiate some prefetches, but they are stale ones generated previously and are hence useless. After these stale ones have been dispatched, the prefetcher may be able to operate some of the time, but because it is also blocked by the requesters much of the time, still initiates prefetches too late and hence causes a further execution time detriment.

As the delay is lowered further, the prefetcher does not have as much of a negative impact on the execution time, but still causes an execution time detriment. This is due to the increased request rate which will prevent the prefetcher from being able to operate for longer, until additional processors have completed the traffic generator task, and also lowers the probability that the prefetcher will be able to initiate a prefetch due to the increased load on the memory system. Because the prefetcher cannot operate until later in the task's life cycle, it stands to reason that it cannot cause as much of a detriment simply because it is operating for less time.

Benchmarks

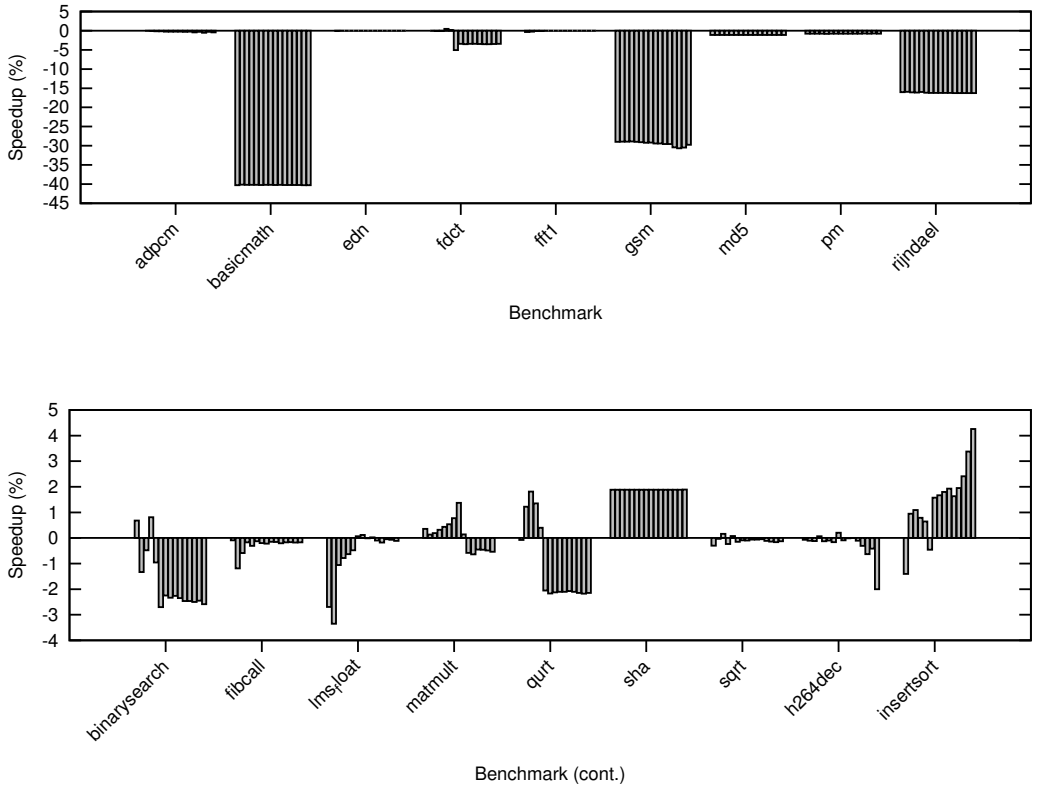


Figure 4.12: Change to the average-case execution time of a set of benchmarks when the prefetcher was enabled or disabled.

The sixteen processor system was then used to evaluate the prefetcher's average-case behaviour on the set of TACLeBench benchmarks. The result from this can be seen in Figure 4.12. Each benchmark was executed on all sixteen cores at once for fifteen minutes, with each processor reporting the execution time of each run of the benchmark with the prefetcher disabled. The benchmark was then re-run with the prefetcher enabled for another fifteen minutes. Within Figure 4.12, each bar in a group shows the change to the execution time which the prefetcher caused for a

processor in each processor index slot (where the left hand bar represents index 0, and the right hand bar represents index 15).

Here, many of the benchmarks showed a slowdown to their execution times, although the results of this correlate with those found for the traffic generators previously. Benchmarks such as *basicmath*, *gsm* and *rijndael* are mainly code-bound benchmarks which typically do not fit in the processors cache. Because the caches in use are 16 bytes wide, and Microblaze has four byte opcodes, this means that the "next" cache line will be required after four instructions have executed. This in turn means that the apparent inter-load delay is reasonably small, and that the prefetcher will not be able to initiate prefetches very often because they will normally be blocked by demand accesses.

When the prefetcher is able to operate, however (i.e. if the task is executing a loop out of cache and hence does not initiate many accesses), then the blocking means that the prefetcher will again initiate stale and useless prefetches, causing the memory system to be tied up with useless accesses. The effect of this is not as bad on the *gsm* benchmark, as some of its execution time is tied up with data loads which, because the prefetcher is mainly blocked, typically cannot be prefetched successfully. *rijndael* also fetches large amounts of data from memory, hence is not affected as much as *basicmath*.

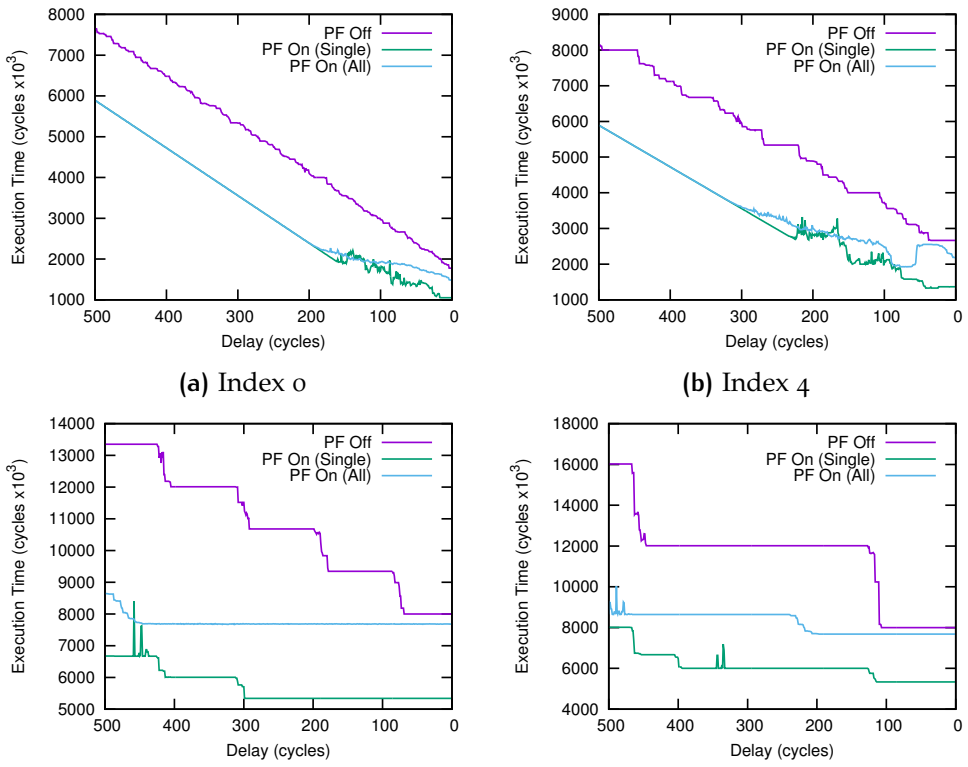
Other benchmarks are either small (e.g. *matmult* and *sqrt*) and perform most of their execution from cache, hence are not affected significantly, or are mostly data bound. *md5* is a benchmark without much code but which performs many data fetches. Because this benchmark can operate mostly from cache, the prefetcher may initiate some data accesses, but is mostly blocked or fetches useless data. *sha* is another data bound benchmark and hence operates in much the same vein as *md5*. This benchmark performs more computation in-between data accesses however (hence has a slightly higher inter-access delay), which allows more prefetches to be initiated which are actually useful and hence has a slight improvement to its execution time.

4.5 RESULTS – HIGH PRIORITY PREFETCHING

When the prefetcher is able to issue prefetches with high priority mode, a prefetch for a given item of data will effectively come straight after the demand request for it. In these cases, prefetches will actually be dispatched and may actually be useful to the target processor, but may also cause significant interference to other requesters in the system. This section will discuss the results for a prefetcher operating at the highest system priority under the average-case and worst-case conditions.

4.5.1 Worst-Case Conditions

Traffic Generator



(a) Index 0 **(b)** Index 4

(c) Index 7 **(d)** Index 14

Figure 4.13: Execution times with the prefetcher enabled and disabled in the worst-case conditions when prefetching for a single processor, and for all processors.

The green line in Figure 4.13 shows the traces for another set of software traffic generators when used under "worst-case" system conditions, again fetching 4096 blocks from memory with a programmable delay. In these systems, the prefetcher is only configured to try and prefetch memory items for the processor; it does not attempt to prefetch anything for the hardware traffic generators.

In these systems, good improvements can be seen; the prefetcher is able to pick up the stream quickly and correctly dispatch prefetches for the useful data. Because of the latency of a single memory transaction, and because the prefetch is issued almost immediately after the demand access is issued to memory, the prefetches can complete in good time and thus all systems see a good improvement by utilising the prefetcher, and none show a performance detriment. With that said, some system begin to see some "noise" as the delay is reduced (for example, below around 175 cycles in Figure 4.13a). This is due to the "squash" mechanism where a prefetch cannot be fully completed before the demand miss for it is required. In many cases, the prefetch and the demand miss can be successfully coalesced, but in some cases, it can be missed. This can happen, for example,

if a prefetch traverses down the tree in the exact cycle that the demand miss for the same line transits up the tree. In these cases, neither of the multiplexers see both of the packets in the same cycle and hence the "squash" is missed. Because this may happen sometimes for a set of requests, depending upon the exact response time of the memory controller, the coalescing of memory requests is semi-random and hence creates the observed "noise".

This noise also exhibits itself in Figure 4.13c around delay 450 and Figure 4.13d around delay 350. This is again caused by the failure to coalesce prefetches, but at higher levels in the tree. A prefetch hit will be issued by the processor, then the demand request shortly after. Around these points, the demand request may be crossing an upstream multiplexer at the same instant that the prefetch is travelling back to the processor and be missed. As the delay decreases, it is more likely that the demand access has been accepted into the multiplexer where the problem occurred (or even make it to the prefetcher), and hence the coalesce can occur successfully.

Processor indices which experience more blocking also see a form of a "stepping" effect, for example, in Figures 4.13c and 4.13d. This is caused by the delay caused at the leaves of the tree due to the arbitration across the whole tree. In these cases, the demand request is dispatched in-between intervals where the processor would be given service, hence all delays between two intervals will have the same execution time. The "stepping" effect is still evident even when the prefetcher is enabled, simply because the hit feedback is relayed through the same mechanism and must still wait for arbitration. There may again also be noise near these steps due to prefetches being coalesced with their demand misses, which may be missed by the multiplexers.

Of course, these results only concern themselves when performing prefetch for a single processor, and not pictured here is the effect that the prefetcher has on the response time of the traffic generators. By prefetching at high-priority without any arbitration, the prefetcher may well cause a performance detriment to the system as a whole, while it still appears as though it causes an increase in performance for a single requester.

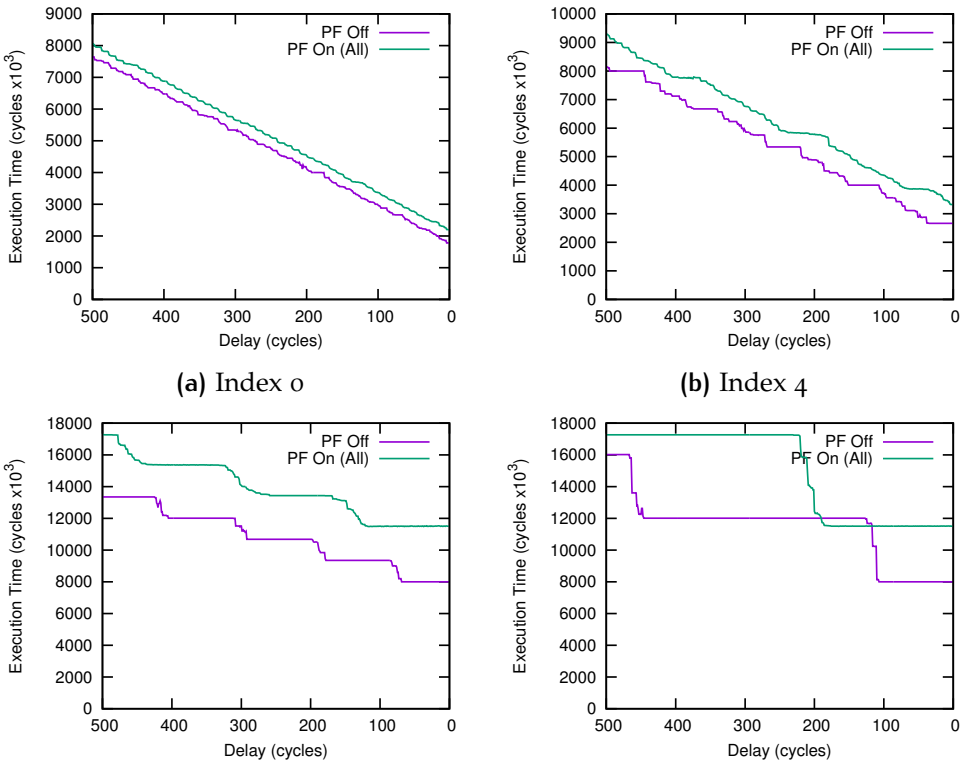
The blue lines in Figure 4.13 show the effect of the prefetcher when it is operating for all cores within the system. In these cases, the prefetcher will have a higher latency before being able to prefetch, and the extra prefetches may cause interference to the rest of the system. Firstly, take the system with the processor at index 0 (Figure 4.13a). At the start of the observed stream, the prefetcher performed basically identically. This is because requests from index 0 experience the least amount of blocking in the system, as such has a reasonably low response time for a memory transaction and hence even with the extra latency brought about by extra prefetches, the "delay" period is typically still sufficient for a hit notification to reach the prefetcher and to be fully serviced.

This extra load on the memory subsystem does take its toll, however. With the processor at index 0, squashes being to occur slightly earlier (at delay 200 rather than 175), and much more of an impact can be seen, with the gains brought about by the prefetcher to be almost negligible at delay 0. This is generally caused by the additional latency in the system from the prefetcher causing both demand requests and hit notifications to reach the prefetcher later. On top of this, there are additional prefetches being dispatched, with the net effect that prefetches for core 0 are dispatched later and hence, the execution time increases. Similar effects can also be seen when the processor is at index 4 (Figure 4.13b), but of course, the latencies are higher and the divergence begins at a higher delay value due to the increased blocking which memory accesses from this index experience.

Processor indices 7 and 14 (Figures 4.13c and 4.13d) also show interesting behaviour with the prefetcher fetching for all cores. Not only is the prefetcher not as effective (for the reasons listed above), but the latency "steps" shift slightly. This is again caused by the additional interference due to the prefetcher. Generally, this is caused by the distance up the tree that prefetch hit feedback managed to reach before the delay was elapsed. The most clear example of this is the far right-hand side of index 14 (Figure 4.13d). In this case, the hit packet would be generated by the processor, then due to the blocking incurred at the lowest levels of the tree, the hit feedback would be immediately followed by the demand miss for the next line. In this case, the tiny improvement to the execution time is caused by the hit feedback reaching the prefetcher *just* before the demand miss for the next line, hence the prefetcher is still doing some work to improve system performance, but not much.

In all of these cases, the prefetcher has still been able to improve system performance, despite the additional latency generated by prefetching for all cores. This is because the hit feedback is still generated just before the demand miss for the next line, and hence has some possibility to travel up the tree and reach the prefetcher just before the demand miss for the next line. Of course, additional prefetches are generated, but the extra blocking causes by these fetches is still not too great. Moreover, since prefetches are dispatched at high priority, the prefetch hit notification effectively causes a high-priority memory request for memory address $A + 1$ to be generated (along with the rest of the prefetches), and hence using the prefetcher effectively shifts most memory requests into the prefetch queue rather than the demand queue for tasks with a "perfect" access pattern.

In order to ascertain the actual performance hit caused by these extra prefetches, Figure 4.14 shows the performance for the aforementioned traffic generators, but when fetching an unprefetchable stream (i.e. fetching 4096 non-consecutive memory words), while prefetching for all hardware traffic generators. In effect, this shows the impact of the extra traffic caused when prefetching for the hardware load generators on the execution time of other tasks, and demonstrates the actual



(a) Index 0 **(b)** Index 4

(c) Index 7 **(d)** Index 14

Figure 4.14: Execution times with the prefetcher enabled and disabled in the worst-case conditions, for a non-prefetchable task.

"worst-case" behaviour of the system (fully backlogged tree, and the prefetcher causing as much interference as possible). It can be seen that all tasks take a performance hit when the prefetcher is running for other tasks due to the created interference. Effectively, each memory request for the hardware traffic generators causes two memory accesses to take place instead of the single access and hence for the most part, the latency of a memory transaction can now be doubled.

For all tasks, this causes the shape of the timing curve to be effectively the same with the prefetcher enabled and disabled, just with more latency. Since the execution times shown in Figure 4.14 also incorporate the delay time, it is difficult to demonstrate the actual impact caused by the prefetcher on the execution time, but for the most part, the prefetching typically causes the "memory latency" for each task to double.

The important result from these graphs, however, is the fact that the prefetcher causes a significant increase to the execution time of the task for each task when the prefetcher is not able to correlate the access pattern of the stream, where all tasks experienced a slowdown of 20-40% at delay 0 when the prefetcher was enabled. Importantly, this detriment is occurring under worst-case conditions and hence shows that the prefetcher can cause an increase to the worst-case execution time of a task of up to 40%, which is clearly unacceptable for a real-time system.

Benchmarks

Of course, the traffic generators shown only capture two models of traffic; a perfectly prefetchable stream, and a completely non-prefetchable stream. Actual tasks do not follow such uniform patterns; they may perform burst, sequential accesses, or they may access purely random data relatively infrequently. The best example of this is code. If a task contains large, single path routines (e.g. unrolled loops), then there is a high probability that the stream will be able to be prefetched. Code with many conditional statements, however, is difficult to prefetch as the processor will only read a couple of lines of memory, then jump elsewhere in the program and repeat.

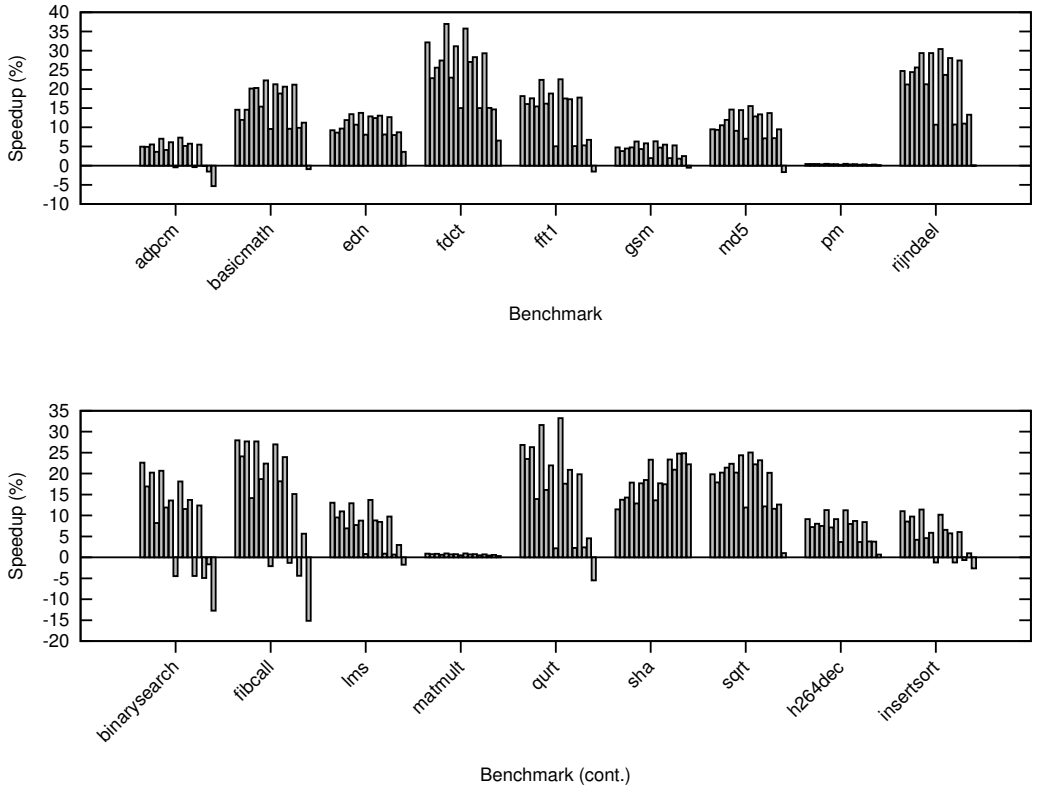


Figure 4.15: Improvements seen by using a prefetcher on a number of TACLeBench benchmarks, prefetching for only the core running the benchmark.

Figure 4.15 shows the potential gains that can be realised by using the prefetcher in a system under worst-case conditions, where the prefetcher is only operating for a single core. Here, each set of bars corresponds to a benchmark, where each bar denotes the processor index where the benchmark was executed. As with the traffic generators, most of the benchmarks shows good gains when the prefetcher was enabled, typically in the range of 10-35%.

The actual speedup depends on the benchmark being executed. For example, *rijndael*, *md5* and *sha* all give good speed-ups because at their core they are processing a serial data array. Moreover, the *rijndael* benchmark performs a large amount

of straight-line computation on said large array, hence sees good speedups on both the code and data interfaces of the processor. For the most part, the *basicmath* benchmark contains many large mathematical routines, which can also be easily prefetched and hence sees a good execution time improvement due to the instruction side alone.

Other examples are the *adpcm*, *h264* and *gsm* benchmarks. These benchmarks still decode data from a buffer in a serial fashion, but their decoding routines are reasonably complex. Due to this, it is difficult for the prefetcher to be able to accurately prefetch the code side, and hence the performance improvement is mainly just on the data side, with a little improvement due to code prefetching. To go further, it is difficult to realise good improvements on the *matmult* and *pm* benchmarks purely due to their size. They are small example processing kernels, using a loop with a high iteration count which can easily reside in cache, hence the memory is only hit once on the first invocation of the benchmark and the remainder of the benchmark is then executed solely from cache without issuing any more memory transactions.

Interestingly, some benchmarks even showed a slowdown, even when the prefetcher is only fetching for a single core. The *binarysearch* benchmark has a fairly random access pattern. For this reason, it is almost impossible to create a correlation for the data being accessed and as such, the prefetcher initiates many useless requests. Some processor indices showed a good improvement because the prefetcher was still *sometimes* accurate, but the major problem for lower-priority requesters is that the hit feedback would be created, which would then block the next demand access for a large amount of time while also causing the prefetcher to initiate another useless prefetch, causing a huge performance detriment. For higher-priority cores, this hit feedback would not be blocked at the leaves of the tree, and not cause significant blocking to the next demand access.

Other benchmarks also showed similar behaviour for processor indices with a high amount of blocking, again, due to the prefetcher fetching effectively useless data. *insertsort* contains a fairly random data access pattern and as such the prefetcher has great difficulty attempting to correlate a prefetchable pattern. The *fibcall* is a small benchmark with very few iterations, hence any mis-predicted code prefetches (i.e. when the prefetcher has fetched an entire routine, and starts fetching past the current routine) can cause a great impact to the execution time simply due to the overall number of fetches which need to be performed anyway.

Figure 4.16 shows the same benchmarks in the same system when the prefetcher is fetching data for all requesters, within which almost all benchmarks show a great detriment to the execution time. Many code-bound benchmarks show similar results (e.g. *adpcm* and *basicmath*). This is again caused by the hit feedback message being blocked at the leaves of the tree, blocking the next demand access since if the amount of blocking is sufficiently high, the prefetch hit message will be

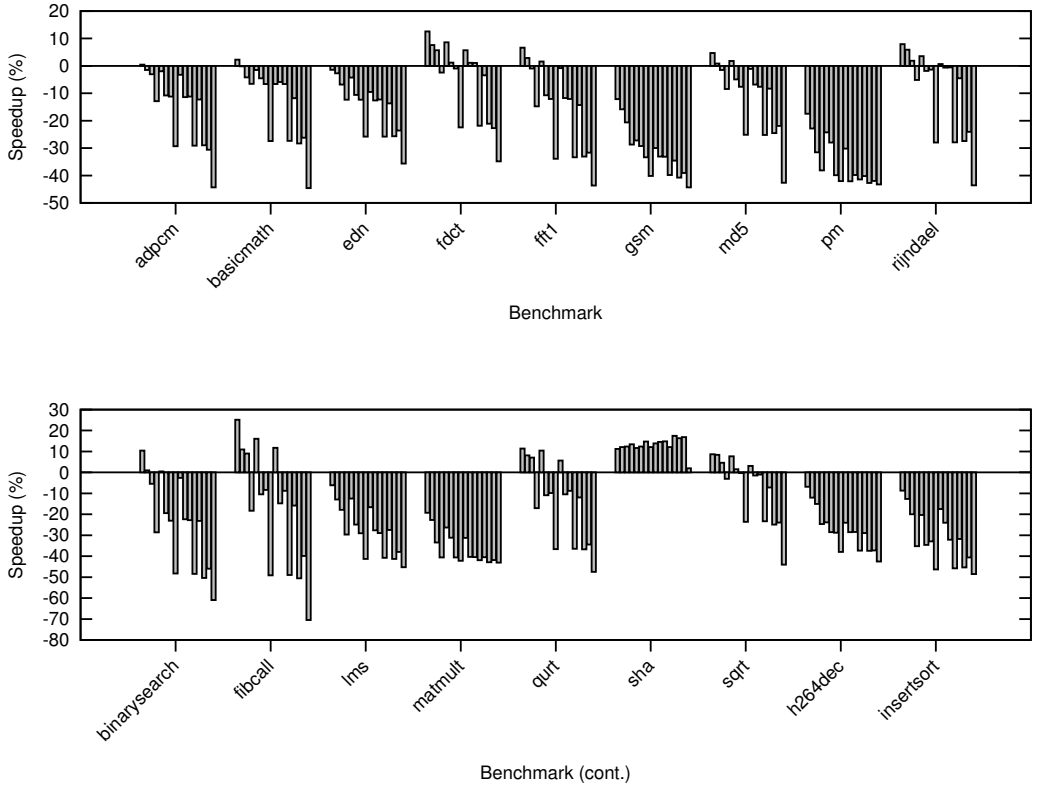


Figure 4.16: Improvements seen by using a prefetcher on a number of TACLeBench benchmarks, prefetching for all cores.

scheduled on to the tree at the same point that the next demand access would have been without the prefetcher. This poses virtually no savings if the next prefetch is accurate and coalesces with the next demand access, and potentially causes a huge detriment if the prefetcher was not accurate, as is evident for many benchmarks.

Many data-bound benchmarks (e.g. *md5*, *sha* and *rijndael*) can still see some performance improvement in these systems, partially due to the blocking nature of data accesses. With some requesters at a sufficiently high priority level, the hit feedback from the last instruction access may begin transiting the tree just before a demand data access occurs. This effectively allows the instruction prefetch to occur while the demand data access is taking place and can improve performance. A similar effect may also be evident with the data and instruction sides switched. The *sha* benchmark also fetches a large amount of data from main memory in a serial fashion. Moreover, the processing performed is such that the execution time is not trivial, but small enough that the loop kernel can fit into instruction cache easily. The prefetcher can hence easily correlate the data stream and somewhat improve the performance of the benchmark.

Finally, for the *fibcall* benchmark, many processor indices do not experience much blocking, hence the hit feedback tends not to block demand misses, and the control flow is not very complicated, hence the prefetcher can fairly accurately fetch most of the code and the hit feedback ends up fetching useful data, improv-

ing performance. Of course, there are still some "useless" prefetches (i.e. when fetching the data past the end of the benchmark), which cause a performance detriment with processor indices with a greater amount of blocking due to the hit feedback both blocking demand misses and fetching useless data.

4.5.2 Average-Case Conditions

Traffic Generators

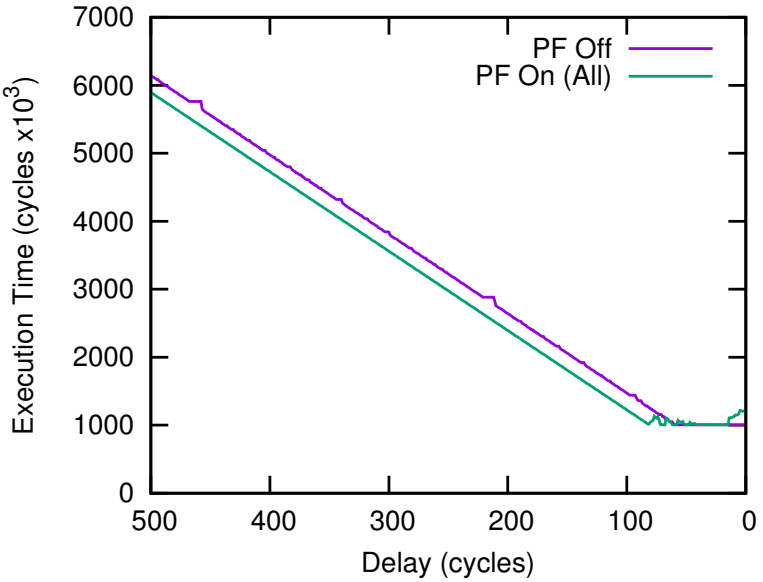


Figure 4.17: Graph of execution time of sixteen traffic generators running on sixteen processors with the prefetcher both enabled and disabled.

The results from executing the traffic generator on sixteen processors when the prefetcher could initiate prefetches at high priority can be found in Figure 4.17. For the most part, the results here are much the same as those found within Section 4.4.2, except after a delay factor of around 80.

Of course, with the prefetcher disabled, the results are identical to that with low priority prefetching, although the merits of high priority prefetching begin to show. Because the prefetcher can now operate at a high priority, stale prefetches do not back up in the prefetcher's output queue. This means that any prefetches which are actually initiated have a higher probability of actually being useful to the requesting task. Despite this though, the prefetcher was still not able to improve the execution time by much in these cases, which for the most part is because the prefetcher *did* manage to initiate a useful prefetch, just that it was initiated just before the demand access from that processor also arrived and hence the savings were not significant.

Towards a delay factor of zero, the prefetcher did also manage to cause a detriment to the task's execution time too. This can be caused if the prefetch and

the corresponding demand access fail to be coalesced with each other and hence, many memory accesses may effectively be issued to the memory controller twice. In addition, the "prefetch hit" notification must also traverse the tree, which may potentially block the next memory access. Because the prefetch calculators will simply throw away a prefetch if the prefetch output queue is full, this can effectively cause the prefetch hit notification to block the next memory access and be completely useless.

Benchmarks

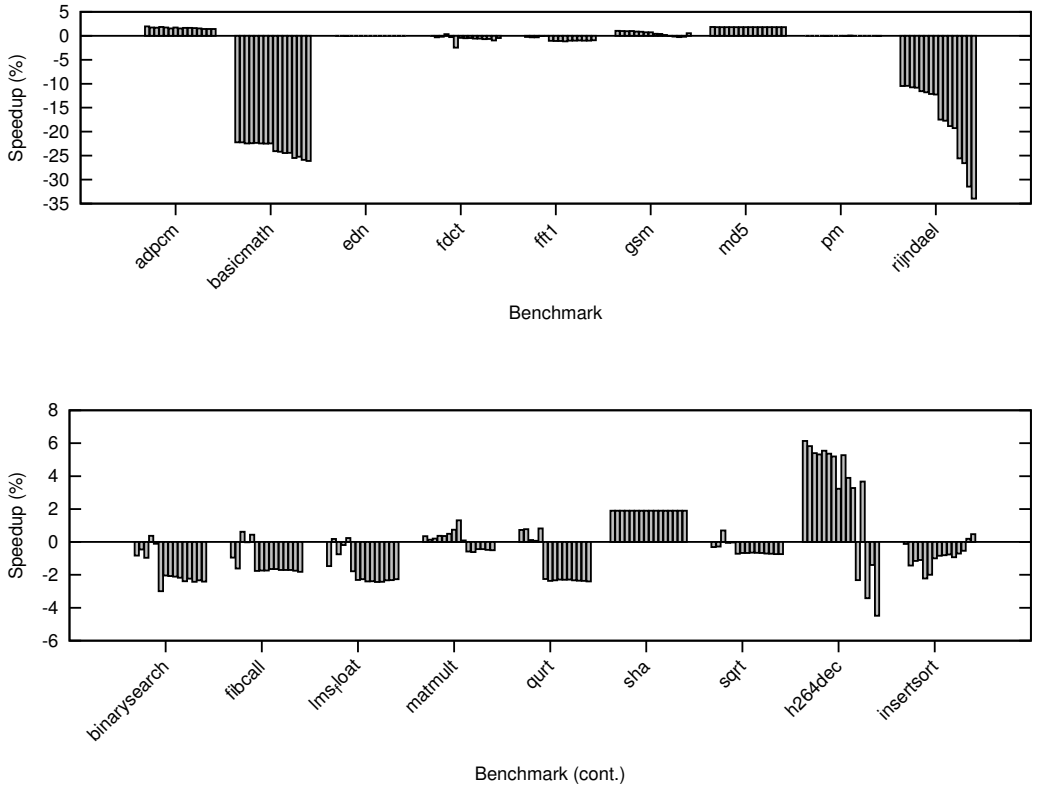


Figure 4.18: Change to the average-case execution time of a set of benchmarks when the prefetcher was enabled or disabled.

The results from the suite of TACLeBench benchmarks when high-priority prefetching is used in the average-case can be seen in Figure 4.18. As with the traffic generators, the results are broadly in the same vein as when low priority prefetching was used. Because prefetches being initiated are not blocked for as long now though, prefetches tend not to be stale and hence the impact is not as bad. Moreover, many data bound benchmarks (i.e. *sha*, *md5* and *h264dec*) now show a speed improvement, although it is still marginal compared with the potential detriment.

Many of the issues that caused problems with the traffic generators are still present with these benchmarks though. As mentioned previously, both *basicmath* and *rijndael* are mostly code-bound benchmarks and hence typically have a rather

low inter-request delay and therefore, as with the traffic generators, show a performance detriment. This is again caused because prefetches can prevent other prefetches from ever being initiated, and hence the "prefetch hit" notification which caused the prefetch to occur in the first place only causes additional blocking for the "next" memory access. This is an effect which is especially evident on the *rijndael* benchmark, where the processor indices which experienced the most blocking show a great amount of slowdown.

4.6 SUMMARY

Clearly, prefetching is potentially useful for managing the increasing latencies brought about by the need to share a large external memory between an increasing number of requesters. Even under absolute worst-case conditions, the prefetcher was still able to improve performance by around 15% when fetching an easily predicted stream. As also shown in the benchmarks in this chapter however, this figure is only really observed when fetching a perfectly prefetchable memory stream; as soon as the traffic pattern deviates from what is expected, the performance soon drops since the prefetcher is now causing a huge amount of additional blocking and is also fetching useless prefetches for the processor in question.

Moreover, the naïve way of prefetching without harming the worst-case execution time of a task, by issuing prefetches at low priority, simply does not work when the prefetcher is not controlled in any way. In these cases, prefetches are created, but sit indefinitely in the output queue as they are blocked by all other requests. These prefetches then prevent any other prefetches from entering the output queue, and because they are not dispatched in a timely fashion, will not ever fetch any useful data when they are finally issued to memory. Without control, it is also impossible to integrate the prefetcher into a system which is scheduled based upon an active period, since the prefetcher does not know when the active period starts.

While the prefetcher does, at times, give a performance gain for a perfectly prefetchable stream (i.e. a serial traffic generator or the *sha* benchmark), many tasks do not fit into this model and hence are not suitable for prefetching. That said, the problem here is not that the access pattern of the task is not perfectly suitable for the prefetcher, but more that the prefetcher is not controlled by the arbitration scheme in any way. Because of this, the prefetcher issues memory accesses whenever it can which of course may cause more a memory request to experience more blocking that it expected, and hence will cause the task to miss its worst-case deadlines.

Instead, to be able to realise these benefits in a system where the tasks have real-time deadlines, the prefetcher must be controlled in a way such that it can

only perform memory accesses when it will not cause any tasks to miss their deadlines. Of course, constraining it in this way will not allow it to fully realise the performance gains found in Figure 4.13, but will hopefully allow it to increase performance somewhat while not causing any detriment to the worst-case execution times. This is a problem which will be explored further in Chapter 5.

Another problem which has been outlined in this chapter is ultimately that one-lookahead prefetching with an output queue is not the best prefetching scheme. While it did manage to yield good results in some of the worst-case systems and for many of the traffic generators, it is very limited in other ways. One of the major reasons behind this problem is that a queue is not the best data structure to use for prefetching; the low-priority prefetch experiments highlighted that the data in the queue can fast become stale, and the high-priority experiments demonstrated that new prefetch data could be discarded if the queue was full and again cause an execution time detriment. Solutions to this architectural problem will be explored further in Chapter 5.

The design of the prefetcher, however, is only the first of the problems. While one-lookahead prefetching can work for data accesses when there may be a significant period of time between accesses, it begins to fall down on code accesses due to the extremely small latency between memory accesses. Because of this, the prefetch hit notification may not have even been accepted by the arbiter before the processor initiates an access for the "next" instruction line, and hence the prefetch hit notification only blocks the processor for longer. If this hit notification causes a useful prefetch to be initiated then it may save a couple of cycles, but otherwise it will only cause a performance detriment. Alternative prefetching schemes for code accesses will be explored further in Chapter 6 when attempting to build a predictable prefetching scheme.

5 WORST-CASE AWARE PREFETCHING

As identified in Chatper 3, the behaviour of the prefetcher is not currently well defined regarding when it can actually operate. This leads to it being impossible to actually reason about the prefetcher itself, and to assert that the prefetcher will not affect the worst-case execution time of a task, or to bound the effect which the prefetcher has on the worst-case. Chapter 4 then demonstrated this effect; while the prefetcher is good on an "ideal" stream of data, it can cause a dramatic detriment to the worst-case execution time of a task as soon as it begins fetching data incorrectly.

To be able to make any guarantees about how the prefetcher will operate, its behaviour must be properly defined and integrated into the system model presented within Chapter 3. This chapter will explore a model of prefetching which can be integrated into said system analysis such that the behaviour of the prefetcher can be bounded, and potentially improve the average-case execution time of a set of tasks without harming the worst-case. Moreover, Chapter 4 also identified some problems with the fundamental design of the prefetcher which may cause it to fetch useless data, or to simply prefetch late. This chapter will also attempt to propose a new prefetcher design to alleviate some of these issues. The remainder of this chapter will therefore propose the theory behind how prefetching can be performed safely on a real-time system, before integrating this into the system model from Section 3.3, creating a hardware realisation of the new model and finally evaluating the performance of the system.

5.1 PREFETCHING SAFELY

5.1.1 Task Model

For a task to be able to be analysed within a real-time context, it must be analysed against a model of the system which describes the worst-case behaviour of each of the units in a system. This model describes the worst-case execution time of each instruction, incorporating the worst-case time to access memory, the worst-case behaviour of cache and the worst-case behaviour of all other architectural features. An example of the breakdown of the execution time of a task is shown graphically in Figure 5.1.

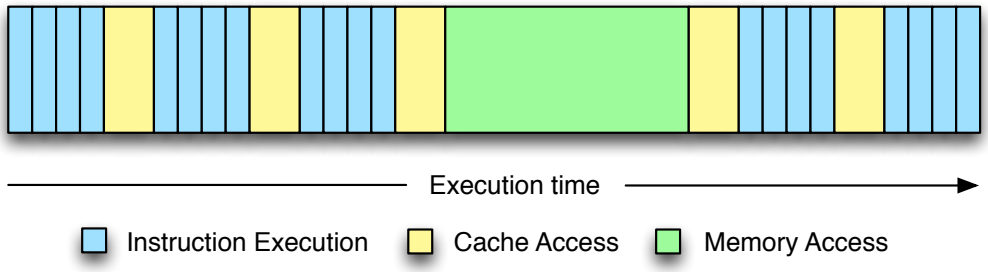


Figure 5.1: Example execution time breakdown of a hypothetical task and architecture.

This model is used when examining the time taken for each instruction in a task, and is used to analyse the *whole* system. The memory subsystem, on the other hand, does not care about the intricate details of the processor cache, or how long each opcode takes, and only concerns itself with how long each memory transaction takes, and when they are initiated. Ultimately, the memory subsystem only cares about the "memory access" box within Figure 5.1.

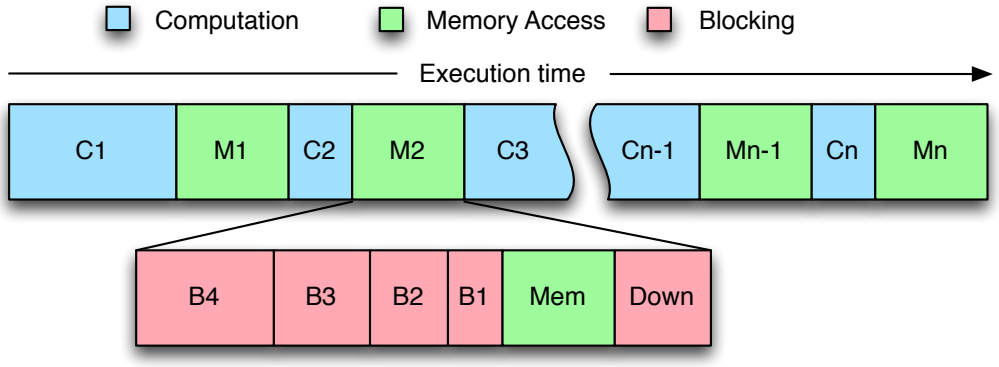


Figure 5.2: Example execution time for a task from the memory subsystem.

Given this, we can create a model for a task from the perspective of the memory subsystem. This model simply encodes the time taken to execute a memory transaction, with the amount of computation time which separates them. Given that the worst-case memory latency for a given core is constant, this model is simply a set of computation times $(C1, C2, C3, ..., Cn-1, Cn)$ which occurs between memory transactions $(M1, M2, M3, ..., Mn-1, Mn)$, hence $task = ((C1, M1), (C2, M2), (C3, M3), ..., (Cn-1, Mn-1), (Cn, Mn)$. This model is shown graphically in Figure 5.2.

This memory latency can then be broken down further into the respective components. Each memory access is made up of blocking at each level in the memory tree, a memory latency as it crosses the prefetcher and DDR controller and is issued to memory, then finally a latency to travel back down the tree again. Of

these, the blocking factors depend on the requester's position in the system, but the memory latency and down-path time are constant across all requesters.

Since it is difficult to make any assertions on the state of the memory system when a memory request is generated, each of the memory access times given in Figures 5.1 and 5.2 are a constant time. This time must encode the worst-case time for which the processor will be blocked while waiting for a packet to be accepted onto the memory tree, which also involves the worst-case blocking which a memory packet could experience when transiting the tree on the way to memory. This latency cannot be reduced from the computed worst-case latency for a memory access based upon the behaviour of other tasks, since this would break the composability property of the memory arbiter.

5.1.2 Prefetching Methods

Given this task model, the prefetching strategy must be modified to fit around the model such that it does not cause any of the blocking terms to increase, and does not cause the memory latency to increase. There are a few methods by which this could be realised, which are explored throughout the remainder of this section.

Exploiting Prefetch Hits

The first method by which the prefetcher can be controlled already effectively exists within the system. Recall from Section 5.1.1 that a task is simply a set of computation times separated by memory accesses. Now assume that a prefetch has previously been delivered to the processor, and is currently residing in the processor's cache. In this case, the memory access corresponding to the prefetch Mx no longer needs to be issued, since the prefetch has already been delivered to the target cache.

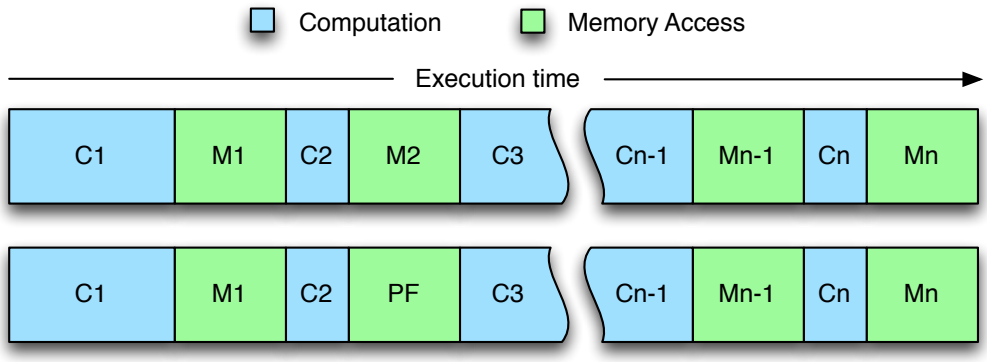


Figure 5.3: Example of replacing the "missing" memory access caused by successfully prefetching data with another memory access. In this example, access M2 was successfully prefetched ahead of time and can be replaced with another prefetch, PF.

In this case, because the prefetcher has speculatively fetched data ahead of time that the processor requires, the processor no longer has to issue a memory access for that data and hence will issue fewer memory accesses than the system analysis assumed would take place. This "missing" memory access can then be replaced by a prefetch for a different data item which may be required in the future; in the best case, the prefetch will be useful and can cause a performance improvement, and in the worst will simply replace this "missing" memory access with a useless fetch which will cause no more performance detriment than if the original memory access had never been prefetched. A graphical view of this can be seen in Figure 5.3, where the memory access M2 has been successfully prefetched, and hence can be replaced by another prefetch at run-time.

Because the system model defines that the worst-case response time of a memory access, $t_{wc}(r)$ only depends upon the requester index r and not upon the actual contents of a memory access, it is therefore possible to issue any other memory access in the place of the "missing" memory access. As defined within Chapter 3, the time to cross the arbiter, $t_{wc}^{arb\uparrow}(r)$ and $t_{wc}^{arb\downarrow}(r)$ also depends only on the requester ID, so the hit message will experience the exact same blocking as that of a standard memory access. The only constraint that needs to be added is that the worst-case time to cross the prefetcher, $t_{wc}^{pf\uparrow}$ remains constant whether it regards a standard memory access, or the time between a prefetch hit message entering the prefetcher and it causing a prefetch to be initiated. Finally, the worst-case memory access time, t_{wc}^{mem} must be constant regardless of the details of the request, so a prefetch must have the same worst-case memory response time than the memory access it is replacing.

This mechanism requires no modification to the system arbitration scheme, but of course, it assumes that an initial prefetch has been somehow delivered to the processor. Moreover, this approach ignores the pollution aspect; if a prefetch displaces any useful data which resides in cache, then it will cause another memory access to be initiated which was potentially not expected, and hence cause an increase in the execution time. That said, the same is true for any other prefetching scheme, unless the prefetcher is also given information on the caching policy and which memory addresses are potentially considered to be "useful" so that the prefetcher can skip these addresses. Instead, a small intermediate cache can be used to buffer the last few prefetches before they are required.

Explicit Bandwidth Reservation

Another simple way to utilise the prefetcher in a predictable manner is to explicitly allocate it a slice of the available system bandwidth. Prefetches can then be arbitrated onto memory using the same arbitration method that normal tasks utilise, and hence are rate limited by the global system arbitration. Given that the amount of bandwidth allocated to the prefetcher is known, the response time of any mem-

ory transaction can still be bounded and hence, "runaway" situations where the prefetcher causes a severe detriment to the observed task execution times can be avoided.

While this method is safe, it requires the prefetcher to be considered ahead of time when calculating the response time of a memory transaction and hence causes the execution times of all other tasks to increase when it is used. This method could also be combined with the hit feedback mechanism listed above and hence may not require much bandwidth, there is a chance that all prefetches dispatched are completely useless. In this case, the execution times of all other tasks will be higher to facilitate the prefetcher, and the prefetcher does not bring any average-case improvement to the system.

For this reason, explicit bandwidth reservation is not really applicable for this system. Moreover, since we do not yet have a framework for the prefetcher to improve the *worst-case* execution time of a task, this method will simply cause an apparent performance detriment and hence violates the requirement that the prefetcher will not harm the worst-case execution time of a task.

Slack Stealing

Another approach is based upon Lehoczky and Ramos-Thuel's "Slack Stealing" approach [98]. In effect, this approach attempts to statically ascertain how much slack is available in a system at a given point of time (i.e. how much free time there is in the schedule after all periodic tasks have run), and hence how much service can be given to aperiodic tasks without causing other periodic tasks to miss their deadlines.

A similar approach can be extended to prefetching within a real-time system, where the periodic tasks are the requests of each task (which are semi-periodic within the arbitration scheme) and prefetches are treated as aperiodic tasks. If prefetches are treated as not having a deadline, then they can be speculatively scheduled, and a similar system can be run on-line to reclaim any system slack.

The blocking term as shown in Figure 5.2 incorporates the time for which a memory request will be blocked at each level in the tree by other memory requests in the worst-case, when using Bluetree. Because the system model mandates that all memory requests have identical worst-case timing, the worst-case response time for each request is equal and hence this blocking factor can be re-written as the number of requests which will block the current request at that level. This factor can also be used with other arbitration methods; for distributed TDM and GSMT, it will be the number of active periods, in the worst-case, before the memory request is given service.

This then makes it possible to calculate the number of times which a memory request can be blocked by other memory requests. If one of these requests is missing (i.e. the processor did not dispatch a memory request in that instant),

then work conservation will normally allow another request to take its place. This allows a memory request to be serviced faster, causing it to complete faster.

Instead of being used for work-conservation, this generated slack can instead be used by the prefetcher to issue a request speculatively. Because the prefetcher is then just using bandwidth which would have been consumed by another memory request, and the worst-case response time for the prefetch is the same as that of a standard memory access, then the prefetcher can be effectively rate limited without any modification to the system model and without causing any change to the worst-case execution time of any other task.

This idea can be implemented by using the concept of a "prefetch slot". These slots will be created within the arbiter whenever a memory request was expected, but absent. As an example, in a simple TDM model, if a requester is given service but has no outstanding requests, the arbiter can instead synthesize a prefetch slot for that requester and send that in the place of a standard memory request. Upon receiving this message, the prefetcher can then fill in the slot with a prefetch request and send that to memory instead. Since the prefetch slot is generated at the same place as a standard memory request, traverses the same arbiter and must also traverse the prefetcher with the same latency as a standard memory request, its timing behaviour is identical to that of a standard memory request.

In systems which use back-pressure to rate limit requesters (i.e. Bluetree), this technique is also work-conserving. The prefetcher can simply throw away the prefetch slot if there is no work to do, incurring only a single cycle of latency. This then allows the next request to be serviced.

This model of prefetching can then be used within all current composable arbitration schemes, and an example implementation is shown for each of the major distributed arbitration schemes below:

BLUETREE: This can be implemented by always running the "blocking counter", regardless of whether the multiplexer has pending work at both inputs. When there is space available in the output buffer, and the input buffer has no pending work, then a prefetch slot is created instead and the blocking counter incremented.

GSMT/DISTRIBUTED TDM: All input queues are considered as having pending work, regardless of the number of pending requests contained within them. At the start of each active period, if the input queue for the requester which is receiving service is empty, a prefetch slot is created instead.

5.2 SYSTEM DESIGN

Before a new system design can be created, the ideas from Section 5.1 must be integrated into the original system model from Section 3.3. This section will first detail these modifications to the system model, then propose a new hardware realisation of this model which incorporates the ideas from Section 5.1 in order to support "safe" prefetching on a real-time system.

5.2.1 Updated System Model

In order to support slot-based prefetching, the system model must be modified in two places. Firstly, the model of the arbiter must be modified in order to define when a prefetch slot will be generated and relayed on. Secondly, the model of the prefetcher must be modified such that it will only issue a prefetch when it receives one of the prefetch slots, or it receives a prefetch hit notification. Since the prefetcher treats prefetch hit requests as prefetch slots, the model of the requesters need not be changed.

Arbiter

Firstly, the model of the arbiter must be extended to define how prefetch slots are initiated. This model again assumes a set of requesters $\mathbb{C}$, each of which issues a set of requests $W(r), r \in \mathbb{C}$. The arbiter then defines a set of requests issued from the arbiter, $W*$ such that:

$$W* = \bigcup_{r \in \mathbb{C}} W(r) \tag{5.1}$$

In addition, the arbiter itself currently defines the inter-request time between two requests from a single requester r, $\delta^{arb}(r)$. It must also define a worst-case time between two requests being *emitted* from the top of the arbiter in the worst-case, Δ^{arb}. In a system that schedules based upon backpressure, this is equal to the time between two requests being accepted by the prefetcher, δ^{pf}, or is equal to the active period in systems which are scheduled based on an active period. It is hence assumed that the arbiter makes a scheduling decision every Δ^{arb} cycles in the worst-case.

A function $W(r, t)$ is also assumed, which yields the current outstanding request for requester r at time t. If there are no outstanding requests, a prefetch slot ω^{pf} should instead be created. $W(r, t)$ is therefore defined as follows:

$$W(r,t) = \begin{cases} \omega_n^r & \omega_n^r \text{ is outstanding at time } t \\ \omega^{pf} & \text{otherwise} \end{cases} \tag{5.2}$$

It is then assumed that $W*$ can be expressed in terms of $W(r, t)$ and Δ^{arb} to define an order by which requests will be issued in the worst-case. As an example, for a two input TDM arbiter with two inputs, L and R a possible representation of $W*$ is as follows:

$$W* = \{W(L,0), W(R,\Delta^{arb}), W(L,2\Delta^{arb}), W(R,3\Delta^{arb})\} \tag{5.3}$$

Finally, in the worst-case, it is assumed that each request utilises the worst-case timing, and hence the time between two requests is always Δ^{arb}. It therefore holds that

$$\forall \omega \in W* : t_d^{arb\uparrow}(\omega) \bmod \Delta^{arb} = 0 \tag{5.4}$$

This implies that in the worst-case, all requests are always initiated on a multiple of Δ^{arb}. Given the definition of $W(r, t)$ above, this implies that a prefetch slot will be dispatched in lieu of an actual memory request from requester r, and moreover that it will be initiated at the exact same time that the "missing" memory request would be. Since Section 3.3.5 defines that the time taken by components above the arbiter to service a request, $\hat{t}^{arb}$ must be constant, the inclusion of prefetch slots does not affect the timing behaviour of the system.

Prefetcher

Given that it is now defined when a prefetch slot can be dispatched from the arbiter, it is now possible to define when the prefetcher will operate. The timing behaviour of the prefetcher is as before, there must be a worst-case time to cross the prefetcher in both directions, $t_{wc}^{pf\uparrow}$ and $t_{wc}^{pf\downarrow}$. The prefetcher can then be treated as a mapping $PF : \mathbb{W} \mapsto \mathbb{W}$ as follows:

$$PF(\omega) = \begin{cases} \omega & \omega \neq \omega^{slot} \\ \omega_x^{pf} & \omega = \omega^{slot} \end{cases} \tag{5.5}$$

In this case, ω_x^{pf} is an arbitrary prefetch; the actual address being prefetched does not matter at this point in time. If there are no available prefetches, this access ω_x^{pf} may be null. This mapping simply states that any non-slot messages must pass through the prefetcher verbatim, and that prefetch slots can be replaced by another access. This scheme must still be able to define a worst-case crossing time, $t_{wc}^{pf\uparrow}$ such that:

$$\forall \omega \in W* : t_d^{pf\uparrow}(PF(\omega)) \leqslant t_a^{pf\uparrow}(\omega) + t_{wc}^{pf\uparrow} \tag{5.6}$$

This constraint ensures that the prefetcher has a worst-case crossing time which covers the time taken for a standard access to cross the prefetcher, and that the worst-case time between a prefetch slot being accepted by the prefetcher and the dispatch time of its replacement access must be the same as for an un-transformed memory access. As before, there is the exception to this rule that a standard memory access and a corresponding prefetch can be coalesced. This exception still exists, and is functionally the same as detailed within Section 3.3.5.

5.2.2 Updated Bluetree Multiplexers

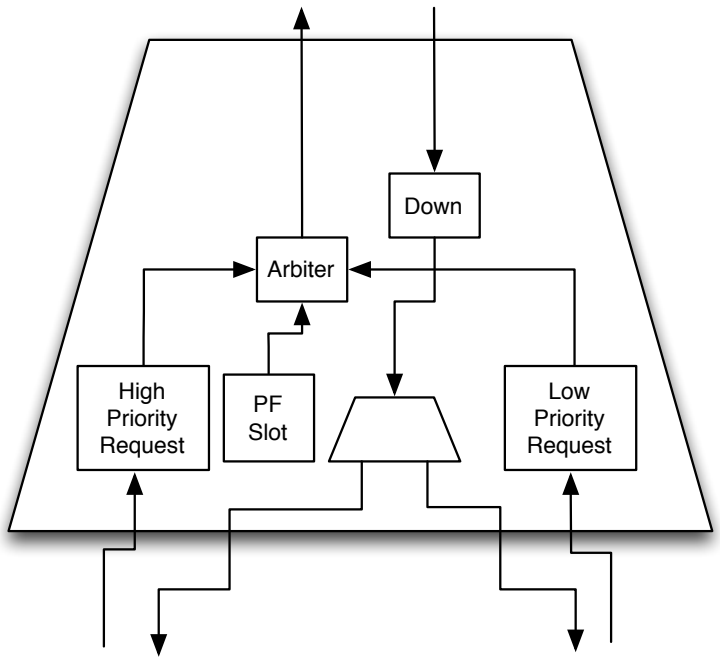


Figure 5.4: Internals of the modified Bluetree multiplexer.

Extending from Sections 5.1.2 and 5.2.1, the slot-based prefetching scheme can be implemented on Bluetree by dispatching a prefetch slot every time the scheduled requester has no outstanding work. In order to support this, each router simply

stores an empty "prefetch slot" in a register, waiting to be relayed. When there is space in the upstream buffer at the exit point of the router, if the arbiter detects that the input buffer that would be given service has no work to do, then it will instead relay a copy of this prefetch slot message. A new architecture for this router design can be seen in Figure 5.4.

Secondly, the arbiter has been modified such that every time a message is relayed across it, the blocking counter is incremented, whether the message was a prefetch slot or an actual request. In effect, each router falls back to using a mini TDM schedule. Because the prefetcher can choose to throw away a prefetch slot when there is no work to do, another request can be scheduled in the next cycle and hence the memory system can be kept busy (minus a single cycle delay), hence the system as a whole can still be considered to be work-conserving.

The multiplexer itself therefore operates in the following manner. Whenever there is space in the output buffer of the multiplexer, it will inspect the blocking counter. If this counter is equal to m, it will then inspect the "low-priority" input queue, otherwise it will inspect the "high-priority" queue. If there is a request waiting at the relevant queue, the multiplexer will update the CPU ID as described in Section 4.2, then move the request to the multiplexer's output. The multiplexer will then increment the blocking counter, or set the blocking counter to zero if it is equal to m.

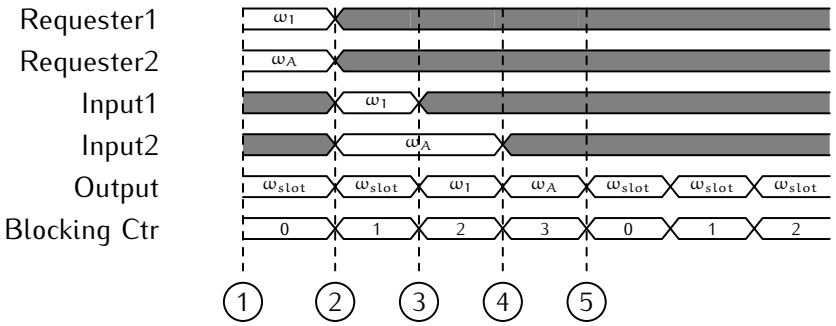


Figure 5.5: Example of a Bluetree multiplexer relaying prefetch slots (ω_{slot}) in lieu of a request. Numbered requests (i.e. ω_1) are initiated by requester 1, and lettered requests (i.e. ω_A) are initiated by requester 2. This assumes a blocking counter $m = 3$.

If, however, the relevant input queue does not contain a memory request to be relayed, the multiplexer will forward the "prefetch hit" placeholder message to the output queue, and again update the blocking counter as before. While this placeholder message does not contain sufficient routing information for it to be delivered back to a processor (i.e. it does not contain a valid CPU ID), it is not required as all fields within the slot will be filled in by the prefetcher. This situation is shown in Figure 5.5, which shows how prefetch slots are created when two requesters initiate only a single request each. Each stage in this diagram is as follows:

1. Both requests each initiate a request, ω_1 and ω_A. In addition, the blocking counter is not equal to m, which implies that input 1 should be given service. Because there is no request here, a prefetch slot is relayed instead. The blocking counter is then incremented as if requester 1 did have a request.

2. Both requests are accepted from the requesters. Because no data was available in the input buffer at the start of this cycle, a prefetch slot is again issued. The blocking counter is again incremented

3. The blocking counter is not equal to m, so requester 1 is given service. There is an accepted and outstanding request from requester 1, ω_1, hence the request is serviced. The blocking counter is then incremented.

4. The blocking counter is now equal to m. The request from requester 2, ω_A is now given service, and the blocking counter reset to zero.

5. Requester 1 would be given service, but it has no requests outstanding. A prefetch slot is hence instead issued. This behaviour then repeats until there is a request available in the input buffers.

Finally, the downwards path for the multiplexer works as previously; the least significant bit of the CPU ID field is inspected, then the CPU ID is shifted right by one position and the response forwarded to the relevant output queue.

Relating to the system model presented in Section 5.2.1, these multiplexers can be constrained in a similar way. For simplicity, this description will only consider itself with a single multiplexer, but this analysis can be chained together to form a full system by assuming a different value of Δ^{arb}. Bluetree is entirely throttled by using backpressure from the next stage, and hence the router will make a decision every time the "next" stage (i.e. the prefetcher) has space in its input buffer. Δ^{arb} is therefore equal to δ^{pf}. Assuming two inputs L and R, a blocking factor of m, and the fact that the blocking counter now runs regardless of whether each input has any outstanding work, $W*$ is defined as follows:

$$W* = \{W(L,0), W(L,\Delta^{arb}), W(L,2\Delta^{arb}), ..., W(L,m\Delta^{arb}), W(R,(m+1)\Delta^{arb})\} \tag{5.7}$$

The definition of $W(r,n)$ is then as described within Section 5.2.1; it either relays the request waiting in the buffer for requester r, or relays the prefetch slot. Because the prefetch slot is logically dispatched whenever a requester without any outstanding work would have been scheduled, its timing behaviour is the same as that of a standard memory request, and the slot would be dispatched whenever a standard memory request would be dispatched. For this reason, it will not cause any detriment to the worst-case execution time of the system.

5.2.3 Updated Prefetcher

The prefetcher has also been modified in order to allow it to utilise prefetch slots as specified in Section 5.2.1. Firstly, the design has been modified such that there is only a single pipeline through the prefetcher, thus both standard memory accesses and prefetch slots being converted into prefetches will incur the same latency. A prefetch merger then takes the requests from the input queue and, if the request is a prefetch slot, merges it with one of the pending prefetches before issuing it to the memory controller. If the request is a "standard" memory request (i.e. a read or write request), it is forwarded on un-modified, where the prefetch merger only exists to ensure both prefetches and standard accesses have the same latency.

In order to ensure that prefetches and demand accesses are handled in the same manner, the prefetch merger is blocked if the output queue is full, allowing prefetch slots to be blocked if there is no space in the output queue, as a standard memory access would be. If there are no candidate prefetches at the point that the prefetch slot would be passed to the output queue, the prefetch merger will discard the slot to ensure that no spurious blocking will take place.

The *prefetch buffers* have also been modified due to behaviour observed in Section 4.3. These experiments showed that prefetches were, at times, being dispatched late due to the blocking from demand memory accesses and from other prefetches. This lead to them effectively being useless, as they could not be coalesced to a demand miss when they were in the output FIFO, and would simply be dispatched later on and cause a performance detriment.

Clearly, a FIFO is not the ideal data structure to use for these output buffers. Instead, the buffers have been changed to a set of registers; one for each processor. These buffers are then accessed by the *prefetch merger* in a round-robin fashion. A restriction has been placed on the system that each processor can only have a single outstanding prefetch at once to simplify the prefetcher design, and so prefetch buffers which correspond to a requester with an outstanding prefetch are skipped from the schedule. Finally, these buffers are only a single register holding the most recent prefetch. If the prefetch is not serviced by the time the "next" prefetch has arrived, the currently pending prefetch is discarded, with the rationale that it was unlikely to have been required by the processor.

While this scheme potentially throws away useful prefetches, it does still encompass many traffic patterns. Instruction fetching is inherently serial and hence if the prefetcher encounters a new prefetch, it either means that the old prefetch was accurate, but dispatched too late (and hence the processor has already fetched those instructions), or that the processor has jumped elsewhere, and hence the instructions being prefetched would not be required by the processor (and are analogous to a branch mis-prediction). On the data side, the same ideas hold for a processor fetching sequential items of data, although there is an obvious counter-example

when the processor is fetching two (or more) different streams of data at once in an interleaved fashion. In this case, some prefetches may be discarded if the prefetch of stream A is not initiated before the processor fetches another item from stream B, overwriting the current prefetch buffer in the process, althoug the prefetcher will still be able to fetch some items accurately and importantly, will not prefetch stale data.

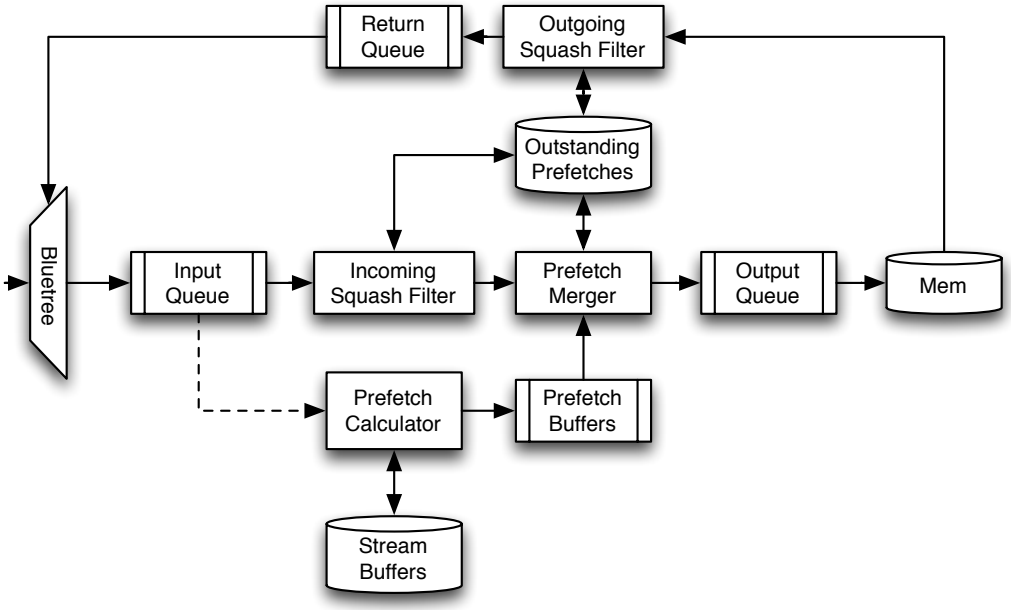


Figure 5.6: Internal block diagram of the modified prefetcher.

A new block diagram of the prefetcher can be found in Figure 5.6. Each block works in much the same way as before, with the modifications listed above. As before, each queue is of size 2, and each other block takes a single cycle to cross with the exception of the calculator block, which takes three cycles. The prefetch merger only takes a single cycle to cross, but both prefetch slots and standard accesses must cross it with the same cycle of penalty and hence, the crossing time $t_{wc}^{pf\uparrow}$ is the same for both prefetch slots and memory accesses. There are two major scenarios for the operation of the prefetcher, which are explored below:

Demand Memory Access

The memory access will first be delivered into the input queue, then pass into both the incoming squash filter and the prefetch calculator. The squash filter, as before, checks whether the request can be coalesced with an outstanding prefetch and if so, updates the outstanding prefetch table to mark the prefetch as "coalesced". If not, the memory access will pass into the prefetch merger, which will simply pass the memory request on to the output queue where it will be issued to the memory controller.

The prefetch calculator works in an identical fashion to before; it will check the set of stream buffers to ascertain whether it can be correlated to an existing stream. If it does (i.e. there is a stream for which the last access was for address $A-1$), then the stream is updated to record $A+1$ as the last address accessed, and the prefetch buffer for the relevant processor will be updated with a prefetch for address $A+1$. If an existing stream does not exist, an existing stream will be replaced in round-robin order with a stream on address A, and no prefetch issued.

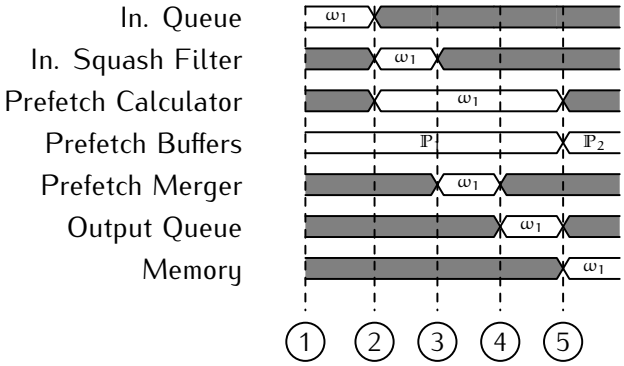


Figure 5.7: Timing diagram showing a standard demand read access, ω_1 transiting through the prefetcher.

A graphical view of a packet moving through the prefetcher can be found in Figure 5.7, where each step performs the following actions:

1. A demand access, ω_1, enters the prefetcher. It is assumed that the current state of the prefetch buffers is described with $\mathbb{P}_1$.
2. The demand access moves into the squash filter and the prefetch calculator.
3. The demand access could not be coalesced with an outstanding prefetch, hence it transits to the next stage, the prefetch merger.
4. Since this is not a prefetch slot, the merger does nothing and simply emits the memory request into the output queue.
5. The memory request is issued to memory. The prefetch calculator has finished processing the memory request and updates the set of prefetch buffers accordingly. The new state of the prefetch buffers is denoted with $\mathbb{P}_2$.

Prefetch Slot

The prefetch slot again is delivered into the input queue. It will then pass through the incoming squash filter, which will ignore it since it is not a read request. It then passes into the prefetch merger, which will inspect the candidate prefetch buffers to find an available prefetch for a requester which does not already have an outstanding prefetch. If found, it will fill the slot in with the details of the

prefetch and issue it to memory, and if not, simply discard the slot and await the next request.

The prefetch merger will also update the outstanding prefetch table to record the new prefetch. As before, when the prefetch is returned from memory, the outstanding prefetch table is checked again. If the respective record for the prefetch is marked as "coaleasced", the prefetch will be returned as a demand read, otherwise it will pass through un-modified. The respective record in the outstanding prefetch table will then be cleared.

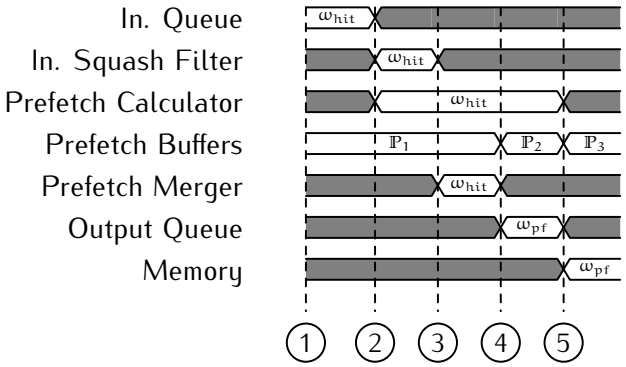


Figure 5.8: Timing diagram showing a prefetch hit request, ω_{hit} transiting the prefetcher. Because there is an available candidate prefetch $\omega_{pf} \in \mathbb{P}_1$, a prefetch can be initiated.

A graphical view of this process can be found in Figure 5.8, and is described in detail below:

1. The prefetch hit enters the prefetcher. The current state of the prefetch buffers is assumed to be $\mathbb{P}_1$.

2. The prefetch hit moves into the incoming squash filter. Because this is a prefetch hit from the processor and not a prefetch slot initiated from the arbiter, it also moves into the prefetch calculator. If it was a plain prefetch slot (i.e. just the arbiter notifying the prefetcher of spare time) then it would only pass into the incoming squash filter.

3. Because prefetch slots do not cause any data to be fetched, the incoming squash filter ignores the requests and passes it on un-modified.

4. The prefetcher merger searches for a candidate prefetch in $\mathbb{P}_1$ and re-writes the prefetch slot to be a prefetch, ω_{pf}. It then updates the prefetch buffers to remove ω_{pf} from them, leaving $\mathbb{P}_2$ as the new state of the prefetch buffers.

5. The prefetch ω_{pf} is issued to memory. In this cycle, the prefetch calculator also completes its processing and adds a new prefetch to the prefetch buffers. The new state of these buffers is then $\mathbb{P}_3$.

As noted within these steps, practically the same flow occurs for both prefetch hits (which are prefetch slots with some added feedback) and prefetch slots originating from the arbiter. The only difference is that a plain prefetch slot does not enter the prefetch calculator in step 2, and hence the state of the prefetch buffers will not be updated in step 5.

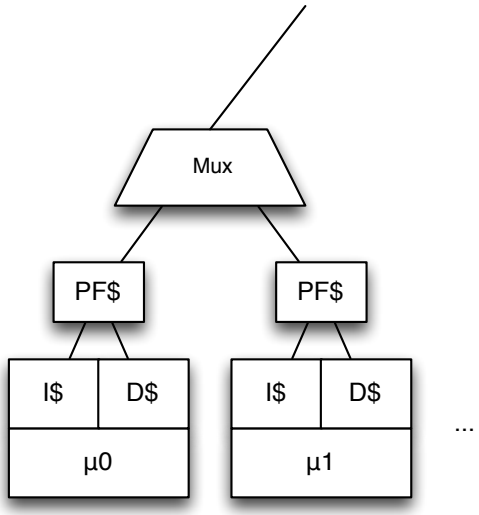


Figure 5.9: Location of the small prefetch cache relative to the processor's own caches.

The final modification that is required to prevent the prefetcher from negatively affecting the worst-case execution time of the tasks running within the system is the inclusion of a small "prefetch cache", as mentioned within Section 5.1.2. This is simply a small direct-mapped cache located between the processor's main caches and the memory interconnect, as seen in Figure 5.9. This cache is read-only, and intercepts prefetches when they are delivered from the prefetcher, storing them for later retrieval by the processor. Whenever the processor performs a read operation, this cache is checked for the required line. If it is found, it is returned to the processor directly from the prefetch cache, and a "prefetch hit" notification delivered to the prefetcher. If two prefetches attempt to write to the same cache location, the previously stored data is simply discarded from the cache.

The actual size of the prefetch cache is a design space parameter which can be explored. If the system is space constrained, it can be reduced to be a single entry buffer storing the most recent prefetch, throwing away any delivered prefetches which were not required. Of course, this may be very wasteful, especially if the prefetcher is prefetching two or more different streams of data at once (as the elements of one data stream may be thrown away if a prefetch from another data stream arrives before the initial prefetch was required). A good middle-ground is to set the number of entries in the prefetch cache to be the same as the number of stream buffers within the processor, with the intention being that each entry in the prefetch cache will map to a prefetch from each stream buffer. Of course though,

being a direct mapped cache, it is possible that two prefetches map to the same entry in the cache, so larger is always better.

For these experiments, the size of the prefetch cache was set to be 32 entries. This is because the prefetch cache operates on behalf of both the instruction and data sides of the processor, each of which has eight stream buffers within the processor (i.e. there are sixteen stream buffers assigned to the processor as a whole). In order to ensure that prefetches were not easily lost, the number of entries was then increased further to 32. Since the main focus of this chapter is to demonstrate that this system setup does not cause any detriment to the worst-case behaviour of a task though, the size of this cache is mostly irrelevant; the important detail is that it is included.

5.3 SYSTEM EVALUATION

Given the theory from Section 5.1.2, which has then been implemented into a hardware system in Section 5.2, it must now be shown that the theory and practise actually stands up to real-world conditions. This section will provide an updated evaluation methodology, and finally an evaluation of the behaviour of the prefetcher within a real-time system using a measurement-based approach.

5.3.1 Evaluation Methodology

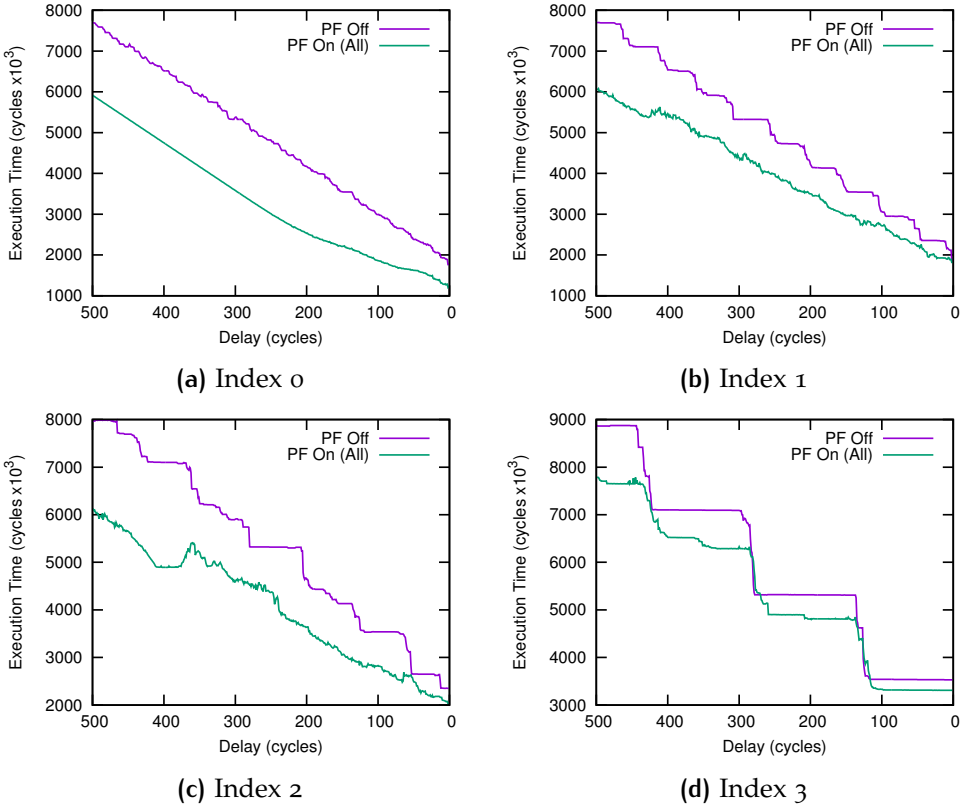
Because this work is focusing on being able to prefetch without harming the overall worst-case execution time, the evaluation methodology is much the same as that used within Section 4.3.

The system will again be evaluated on a sixteen-core system, where fifteen of the available clients are hardware traffic generators, each of which initiate a memory request on every cycle for which their output queues have available space within them. This is used to simulate absolute worst-case conditions across the memory tree. The final remaining client slot will then be used to connect a processor which is then running the task under evaluation.

Again, two sets of applications will be used. Firstly, a set of *software* traffic generators will be used which issue an "ideal" access pattern. These just fetch sequential cache lines from memory, then wait for a holdoff period before initiating the next access. This is used to evaluate the behaviour of the prefetcher under worst-case conditions, but when seeing an ideal traffic pattern. This task is then run again but with a non-prefetchable pattern to demonstrate the impact that the prefetcher has under absolute worst-case conditions. These two tasks should demonstrate both extremes for the prefetcher on a single input.

The prefetcher is then evaluated using a set of benchmarks from the TACLe-Bench suite [94]. These benchmarks are single-path, and already have loop annotations for the purposes of worst-case behaviour analysis. Despite this, they are still executable, and provide a simple way to evaluate the behaviour of a real-world task with multiple running streams alongside the prefetcher.

5.3.2 Software Traffic Generators



(a) Index 0 **(b)** Index 1

(c) Index 2 **(d)** Index 3

Figure 5.10: Execution times with the prefetcher enabled and disabled in the worst-case conditions.

The execution times for a software traffic generator running on processors at indices which do not experience much blocking can be seen in Figure 5.10. There are a few interesting behaviours which are exhibited in these graphs compared with the graphs presented within Section 4.3.

The first of these is that the "prefetch off" execution times are actually lower for many of the benchmarks using the slot-based prefetching scheme. The reason for this mainly lies in the slot generation mechanism; by dispatching prefetch slots (which are still dispatched even when the prefetcher is not running), the "blocking counters" are always free running. This means that it is now unlikely for a memory request to be blocked at every level throughout the memory tree; a memory request on index 15 may be blocked by three other requests at the lowest level in the tree,

but it may then only have to wait for a single other request at the next level because of the effect of the prefetch slots on the blocking counters.

This means that many requesters which were connected to the low-priority input of a multiplexer have observed an overall improvement to the execution time of their tasks. Conversely though, requesters connected to high-priority inputs tend to observe additional blocking because m prefetch slots have been dispatched and hence the multiplexer now blocks the request where it did not before. Of course, the state of the tree cannot be ascertained at any one point in time, and hence the absolute worst-case response time for a memory transaction is as it was before, as a memory request *may* encounter the maximal amount of blocking at each level within the tree.

When the task is executing from index 0 (Figure 5.10a), memory requests from the running task experience the least amount of blocking along their path to memory. The behaviour of the traffic generator is almost identical to that observed without any control over the prefetcher, where the task still has a good speedup without any performance detriment. The reasoning for this is mostly due to the method by which prefetch slots are generated; because all other requesters are issuing memory requests as fast as possible, prefetch slots will only be generated along the processor's path to memory while it is not initiating any requests of its own. Given the holdoff period is of the order of hundreds of cycles, this allows many prefetch slots to be generated in-between the processor creating demand misses.

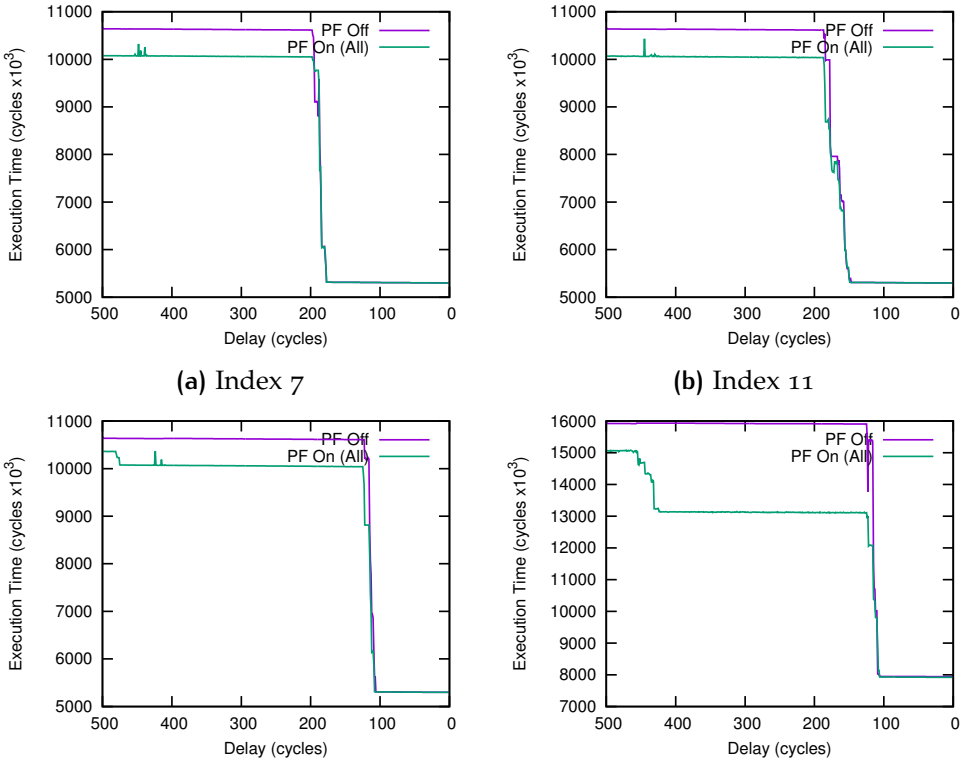
Other processor indices without much blocking also show good results from this approach. Again, this is because of the number of prefetch slots which can be created while the processor is performing computation, hence allowing the prefetcher to run and create prefetches for both the hardware traffic generators and the running processor. Of course, these gains decrease as the amount of blocking increases, but all tasks running on processor indices 0-2 show improvements of 15-40% when the prefetcher is enabled. Processor index 2 (Figure 5.10c) also shows an interesting effect at around delay 375 where the execution time with the prefetcher enabled increases again. This occurs around the same area that there is a "step" in the prefetch off line and can be explained similarly to some of the noise spikes found within previous experiments. At around this delay point, the multiplexers fail to coalesce the prefetch (travelling from the prefetcher to the processors) and the demand miss for the same line, hence the execution time increases again. This also explains the second spike at around delay 75.

The graph of index 3 (Figure 5.10d) shows what happens when the amount of blocking starts to increase. All memory requests from this index must cross two multiplexers on the low-priority side first, then two on the high-priority side. The result of this is that the "prefetch off" trace has the stepping effect seen in the

previous chapter, and also that it is difficult to initiate prefetches under certain circumstances.

As before, the steps are caused because there are effectively now time intervals between possible memory accesses. Any memory fetch which happens between these intervals will have to wait until the next available interval. On the boundary between these "steps" there will be fewer prefetch slots since what was once a prefetch slot will now be used to transmit a demand miss instead, although this is noisy due to the non-deterministic nature of DDR. Moreover, at these boundaries, some prefetches fail to be coalesced with their demand misses since the coalescing of prefetches on Bluetree is not perfect (it is possible that a prefetch transits out of the multiplexer in the same cycle that the respective request enters, making the coalesce impossible) and hence the improvement due to the prefetcher is lower.

Despite this, the system still observes improvements of 0-15% depending on the delay. Importantly, however, the prefetcher does not cause a detriment to the observed worst-case execution time of the task. The execution time with the prefetcher enabled is slightly higher at some points (i.e. around delay 130), although this is only by around 1%, and can be explained because of the apparent execution time improvement due to the slot generation mechanism as discussed before; in these cases, it is likely that the system never caused the worst-case conditions to occur on the tree.



(a) Index 7 **(b)** Index 11

(c) Index 13 **(d)** Index 14

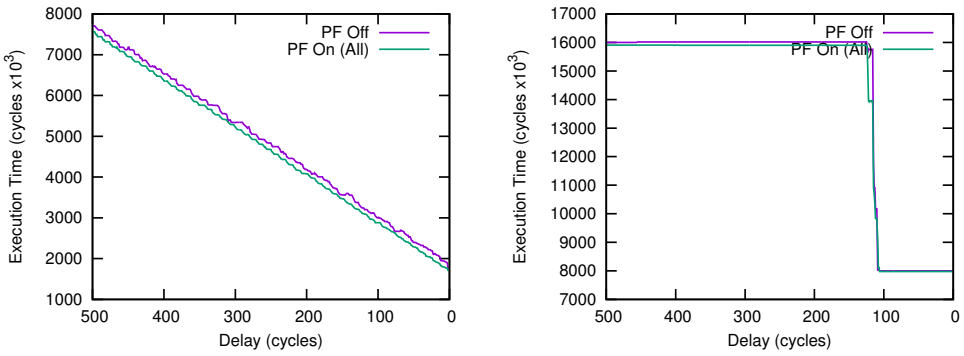
Figure 5.11: Execution times with the prefetcher enabled and disabled in the worst-case conditions.

As the processor index increases, of course, as does the amount of blocking. For the reasons explored earlier in this section, this means that fewer prefetch slots are created and hence fewer prefetches can be initiated. As an example, the processors in index 7, 11, 13 and 14 experience some of the highest levels of blocking when attempting to transit the memory tree, and the results from these processors can be found in Figure 5.11. For all of these systems, an improvement can be seen while prefetch slots are still actually being generated. Of course, when the delay between prefetches is reduced, eventually no slots can be generated and hence the prefetcher is unable to operate. It should be noted, however, that even in this case, the prefetcher does not cause any detriment to the worst-case execution time. Finally, when the processor is running from index 15 (not pictured), the blocking is so large that prefetch slots are *never* generated, and the execution time is no different when the prefetcher is enabled or disabled.

All other processor indices exhibit similar behaviour; the lower-priority indices begin to exhibit the previously seen "stepping" effect, and eventually as the number of prefetch slots dispatched becomes zero, there is no performance improvement or detriment by using the prefetcher in the system. There is one interesting point in Figures 5.11a and 5.11d though; the line for "prefetch off" has a spike around the point of the "step". The "stepping" effect can be explained as previously, but the spike occurs because there is some jitter in the time at which the request is issued, relative to the multiplexer's view of time. This is because the time at which a request is issued is relative to the time at which the last request completed, for which there may be some jitter depending on the state of the other multiplexers, or if the memory controller scheduled a refresh etc. This means that the "next" request from the processor may be slightly early, in which case it would hit the window at which the processor would be scheduled anyway and hence gain an improvement to the execution time, or be slightly late in which case it would just miss its scheduling window and have to wait for the next one. The fact that it sometimes hits and sometimes misses its window then leads to the noise shown in the graphs.

The results from the same systems when an "unprefetchable" stream is used for two of the processor indices can be seen in Figure 5.12. While only two systems are shown here, all systems present almost identical behaviour; the system performs almost the same with the prefetcher enabled or disabled.

In these systems, while prefetch slots may still be created within the arbitration scheme, they can never be used for anything which is useful for the running task. Nevertheless, these systems show no slowdown (except a marginal slowdown in some cases as explained earlier in this section) as a result of the prefetcher being operational in the system. Some of these systems actually show a marginal improvement to the execution time; this is typically caused by the rare cases where the prefetcher accurately fetching data for the hardware traffic generators which



(a) Index 0 **(b)** Index 14

Figure 5.12: Execution times with the prefetcher enabled and disabled when an un-prefetchable traffic generator is used.

are simulating the worst-case system conditions, hence the response time of *other* requests (which are blocking the stream of memory requests being inspected) is lower and hence the amount of apparent blocking is lower.

5.3.3 Real-World Benchmarks

As in Chapter 4, the same systems were then used to run a selection of the benchmarks from the TACLeBench [94] suite of benchmarks. Again, the benchmark was run from each processor index on the tree, and the change in execution time observed.

The results of this can be found in Figure 5.13. As with the traffic generators, these benchmarks show improvements typically in the region of 5-30%, depending on the access pattern of the benchmark being run, however, the apparent improvements are lower than those observed when using the un-constrained prefetcher (from Figure 4.16), again because prefetches begin to be throttled, and a side-effect of creating prefetch slots does actually slightly improve observed execution times in some situations.

The reasoning for the performance increase of tasks is also the same as that demonstrated within Chapter 4; benchmarks which have a regular and predictable access pattern can be greatly sped up by the use of a prefetcher. *basicmath* and *rijndael*, again, utilise large subroutines which appear serially in memory, hence is an ideal pattern to prefetch. *md5* and *sha* operate on large serial data streams. *gsm* and *h264dec* again begin to observe diminishing returns, since the amount of computation begins to domainate over their memory blocking times. Finally, some other benchmarks (e.g. *fibcall*) are very small, hence the prefetches which do manage to be dispatched in time have a large effect on the overall execution time.

There are some interesting effects which are apparent within these graphs. The first is that there is almost always a huge difference in the impact of the prefetcher

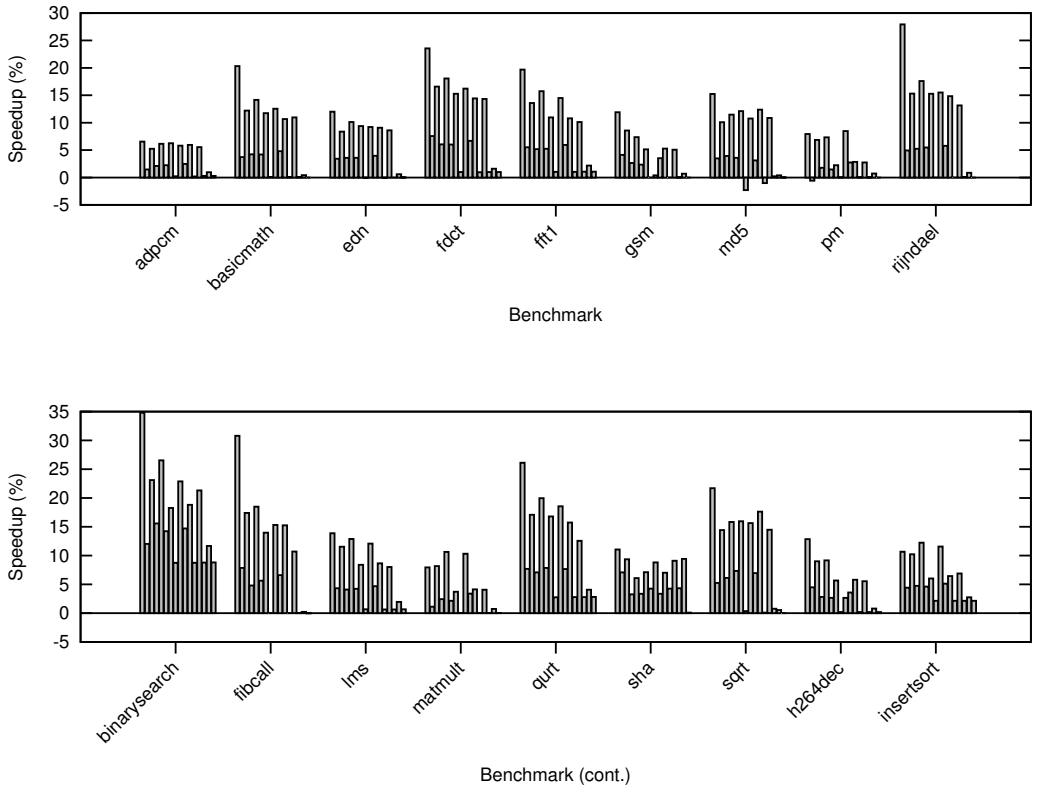


Figure 5.13: Observed execution time improvement by utilising a prefetcher with the TACLeBench benchmarks.

between processor indices 0 and 1, and between 2 and 3 and so on. This is caused by the difference in the number of slots which may be generated by the arbiter for these indices. Because they tend to be blocked towards the bottom of the tree, the probability that a prefetch slot will be generated between two memory accesses is lower, and hence the potential improvement is lower.

In addition, many of the tasks show absolutely no detriment to the worst-case execution time of a task, even for the processor indices with the highest amount of blocking (i.e. 7, 13, 14 and 15). This is simply because prefetch slots are only created if a a prefetch hit occurs, in which case the prefetch slot is only taking up the time within which the memory access would anyway, or created when the processor did not issue a request by the point it would have received service, in which case "dead" time would have been created. Two benchmarks (*md5* and *pm*) do show a slight performance detriment for some cores. Again, this is more due to the fact that the actual worst-case was never observed; the apparent detriment is caused by worst-case conditions actually occurring on the memory tree.

5.4 SUMMARY

The work presented within this chapter provides a framework for performing feedback control of a prefetcher by a distributed, composable arbitration scheme in order to utilise a prefetcher within a real-time system, without causing any detriment to the worst-case execution time of the tasks running within said system. Furthermore, this framework was evaluated on a real hardware platform to demonstrate that it does not cause any detriment to the worst-case execution time, and that it typically improves the average-case execution time of the running tasks.

While this framework cannot currently improve the *worst-case* execution time of a running task, all is not lost. Such a system may be used within systems with both hard real-time tasks running, and either soft or non real-time tasks. By prefetching for both hard and soft real-time tasks, it may be able to improve the execution time of some tasks and hence may be able to help some soft real-time tasks to meet their deadlines. It may also allow for soft real-time tasks to perform any optional processing if there is sufficient resources remaining for it.

While prefetch slots are currently generated in a predictable way with respect to the running task, it is not possible to accurately ascertain when prefetch slots will be generated ahead of time. To do so relies upon determining how many prefetch slots may be generated from the "slack" time of a set of tasks, while ultimately depends upon the memory access pattern of each task and how much blocking each of these memory accesses will experience. To do so would require analysing each task together and hence breaks composability. Further work therefore can build upon this framework, and attempt to develop a method by which prefetch slots can be generated ahead of time, and where a lower bound on the number of prefetch slots can be ascertained. From here, the predictability of the prefetching method can used in order to attempt to improve the worst-case execution time analytically ahead of time.

Finally, as also seen in Chapter 4, stream prefetching with "confirmation feedback" from the prefetch hits begins to limit the effectiveness of the prefetcher. Simply enough, by the time that the prefetch hit has managed to reach the prefetcher, the next demand memory request has typically almost also reached the prefetcher and also requires servicing. Ideally, a better feedback mechanism or a more aggressive prefetching scheme should be used in order to further reduce the time between two prefetches for a running task.

6 WCET IMPROVING PREFETCH

Currently, Chapter 5 has provided a methodology by which a prefetcher can be utilised in a real-time system without harming the worst-case execution time of the running task. By the inclusion of this prefetcher, a task's average-case execution time can be shown to improve, even while the system is under worst-case conditions (i.e. maximal blocking on every memory access). While this approach can be useful in mixed-criticality systems in order to allow soft real-time tasks to perform more computation, it does not yet address the growing problem that rising memory latencies are causing *worst-case* execution times to dramatically increase.

This chapter will take the work presented within Chapter 5, and attempt to ascertain the conditions under while the inclusion of a prefetcher can be used to *improve* the worst-case execution time of a given task. Moreover, the conclusions of both Chapter 4 and 5 identify the fact that "confirmation-based" prefetch approaches (i.e. ones where every prefetch must be acknowledged to be useful) do not scale well, and hence this chapter presents a new prefetch scheme which does not suffer from such issues.

6.1 IMPROVING THE WORST-CASE

This section will outline the prerequisites under which a prefetcher can be used to improve the worst-case execution time of a running task. It will then outline a new prefetching scheme which is more suited to these prerequisites and the arbitration scheme, before finally detailing how this prefetcher is integrated into a worst-case execution time calculation.

6.1.1 Worst-Case Execution Time Theory

As briefly mentioned within Section 2.1, static worst-case execution time analysis tools operate using a model of the platform being used. The tool then splits the program into a set of blocks, then analyses the program instructions within each block against the system model in order to ascertain, under worst-case conditions, how long the block of code will take to execute. Finally, according to the call graph of the task, the execution times of each block is combined in order to derive the worst-case execution time of the whole task. An example of this can be found in

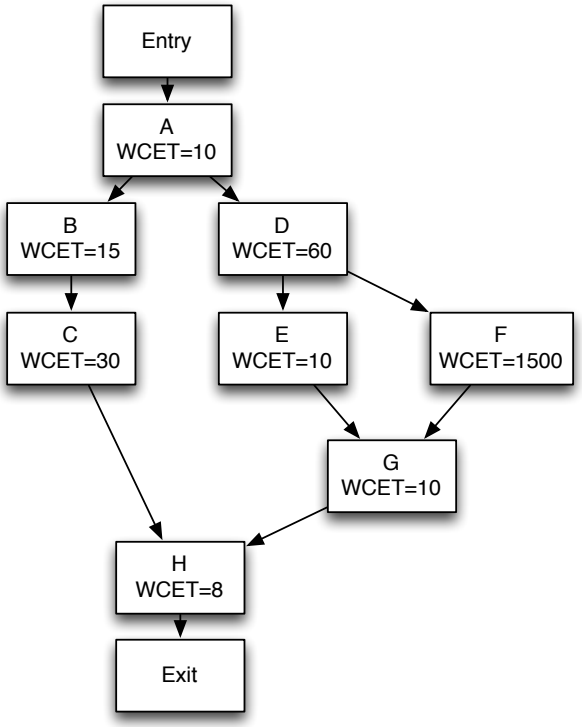


Figure 6.1: Example call graph for a hypothetical task.

Figure 6.1, where the worst case path is $\{A, D, F, G, H\}$ with a worst case of 1588 cycles.

For feasibility and safety, these tools typically assume that each memory access will take a fixed number of cycles, which is the worst-case latency for a memory access. This worst-case figure includes the time taken to transit the arbitration, the blocking from other tasks due to the arbitration scheme, the time taken to fetch the request from memory, and finally the time taken to deliver the request back to the processors again.

In order to integrate the prefetcher itself into the worst-case execution time calculation, the prefetcher must be able to be modelled in a predictable manner, and the behaviour of the prefetcher given an input task must be able to be modelled. The prefetcher presented within Chapter 5 shows such a predictable design; it will only issue a prefetch when it receives a prefetch slot, and it will service prefetches for each processor in a round-robin fashion. For this reason, a prefetch will be initiated on behalf of a processor, in the worst case, after the prefetcher has initiated $n - 1$ other prefetches, where n is the number of processors in the system.

Of course, this only details for which processor a prefetch will be initiated when a prefetch slot has been received. In order to be fully predictable, it must also be possible to bound both *what* will be prefetched and *when* these prefetch slots will be created. This section will concern itself with being able to predictably create these

prefetch slots, while Section 6.1.2 will concern itself with what will be prefetched in these slots.

There are two main sources of prefetch slots. The first of these is from a processor not completely utilising its allocated bandwidth. In every case where there is no pending memory access when the arbiter "expects" there to be an outstanding request, a prefetch slot will instead be generated. Assuming that a worst-case service period (i.e. the longest period between two requests being relayed for a given arbiter input) $\delta_{wc}^{arb}(r)$ can be ascertained for a given arbiter input $r \in \mathbb{C}$, it can therefore be asserted that if two memory accesses of a task in the *best* case are initiated at more than $\delta_{wc}^{arb}(r)$ cycles apart, then a prefetch slot will certainly be generated in the meantime. The best-case model is used in this case to assert that a prefetch slot will *definitely* be issued in the given time period, since if the worst-case is used, some of the operations in-between the given memory accesses may execute slightly faster, meaning that the two memory accesses are now less than or equal to $\delta_{wc}^{arb}(r)$ cycles apart, in which case a slot will not be generated.

There are problems with this approach, however. The first is that a *best-case* model of the system must also be constructed. While this is not much extra work when also designing the worst-case system model, it does mean that the number of potential prefetch slots may be extremely pessimistic. Furthermore, the generated slots will only be able to apply to the currently running task; for these slots to be used for other cores would again require that each task be evaluated against all other tasks, breaking composability. This means that prefetches cannot be shared at all between cores, and hence any prefetch slots generated by an un-prefetchable task are effectively useless.

Another approach is to modify the system such that it is possible to statically assert how often a prefetch slot will be generated in the worst-case. Composable system analysis operates by assigning a partition of system resources to each task, then analysing each task within said partition. If the combination of the partitons used by each scheduled task does not fully utilise the available system resources though, there will be spare capacity in the system. If the memory system is under-utilised, this then implies that then prefetch slots may be generated from the "spare" capacity. For Bluetree, this would be equivalent to one of the tree inputs being un-connected as there is no task issuing requests from said input. In this case, it can be asserted that a prefetch slot will be generated every $\delta_{wc}^{arb}(r)$ cycles due to a lack of a requester connected to this input. Given this knowledge, it is therefore possible to predict the worst-case time in-between prefetch slots being created and, given a predictable prefetcher, the worst-case number of slots which need to be generated before a prefetch can be initiated for a given processor (in this case, $n - 1$).

While this approach does break composability, the cost of analysis is much lower than if each task's access pattern was evaluated in order to ascertain when prefetch

slots could be generated from a task not fully utilising its bandwidth quota. In addition, if all available system bandwidth has been allocated to other tasks, then the prefetcher cannot be used to improve the worst-case execution time (since analytically, prefetch slots will not ever be generated). If it can be asserted that a tree input will be left un-connected, however, this technique can be used. Moreover, this technique can be used to derive a set of execution times, where each corresponds to a set of un-connected tree inputs.

6.1.2 A Better Prefetching Approach

Now that Chapter 5 gives a prefetcher design which is predictable in so far as which processor will receive a prefetch, and Section 6.1.1 can be used to ascertain when a prefetch will happen, the final item which must be predictable is *what* will be prefetched. Previous work presents techniques where there are multiple prefetch buffers from which a single prefetch may be outstanding. These present problems for system analysis as upon entering a code block, it is not easily possible to ascertain whether a prefetch is already outstanding for a processor. A tool may hence determine that a prefetch can be dispatched, when it may actually be blocked by another outstanding prefetch.

Moreover, this technique uses a "confirmation-based" approach, where the processor must acknowledge that a prefetch was useful before another can be created. Of course, this acknowledgement must also transit the tree, being blocked by other memory accesses and hence this approach is reasonably slow; previous work within this thesis has shown that in many cases the prefetch hit notification reaches the prefetcher only just before the next demand miss. For code prefetching, confirmation-based prefetching is not ideal; such a prefetcher requires a number of sequential fetches before it will begin creating prefetches, which may be longer than the size of many basic blocks in the program and hence the analysis tool may determine that prefetches can never be created. Moreover, if the basic block is the same size as the number of fetches required to train the prefetcher, then the prefetcher will begin fetching useless data just after the current basic block, damaging performance.

Instead, the prefetcher should be able to issue prefetches as fast as possible. The rationale here is that every memory request is assumed to have maximal blocking, where non-existent requests are instead converted into prefetch slots. For that reason, even if created prefetches are entirely useless, there is no impact on the worst-case execution time. A better prefetching approach can therefore afford to be as aggressive as possible, on as little information as possible.

While data fetching is typically viewed as "random", instruction fetching does typically have some form of inherent pattern. Compilers typically generate a set of "basic blocks", each of which is a single-entry multi-exit block of code which

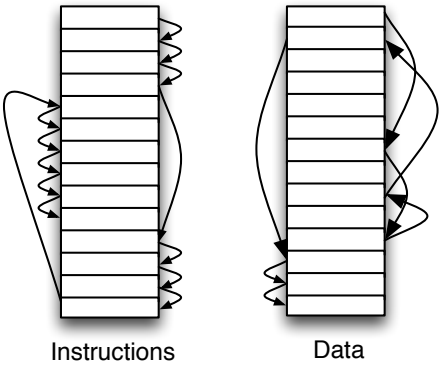


Figure 6.2: Comparison of an example data access and instruction access.

is executed serially. Some compilers may also attempt to ensure that the most likely path to be followed appears serially in memory too. A comparison between such an instruction fetch pattern and an example data pattern can be found in Figure 6.2.

The obvious technique to perform code prefetching is therefore to simply always prefetch the next cache line. Because of the serial nature of programs, it will *probably* be useful, and for reasons discussed previously, even if it is not, it will not affect the task's worst-case execution time. The prefetcher can then simply fetch sequential lines from memory until it observes a demand miss for a different address A, in which case it can change its fetch location to $A + 1$ and continue as before.

Of course, this approach only works well for code, and is not useful for the random nature of data fetching. In this case, the prefetcher can fetch instructions in this manner, but may also support the prefetching of data using the previously evaluated method in order to attempt to improve the average-case execution time. Since this work is currently focusing on improving the worst-case, however, it will only concern itself with this instruction fetching scheme.

This technique should also give better results than those presented within Section 5.3 due to the increase in aggression. Because the latency between a prefetch being completed and the next prefetch being created is now zero, more prefetches should be created in a more timely fashion.

Because of this increased aggression, however, cache pollution is now a very real problem. Take the situation of a small program which is quickly loaded into cache, then executes a large number of iterations exclusively from cache (e.g. *matmult*). In this case the prefetcher will simply continue forever fetching data which is never required, and eventually overwrite the useful contents of cache. The first obvious fix is to cap the number of prefetches which can be fetched in a stream. This prevents the prefetcher from fetching too much data, and if set correctly will not

have an effect over the static analysis of the task (since extremely large basic blocks are very rare).

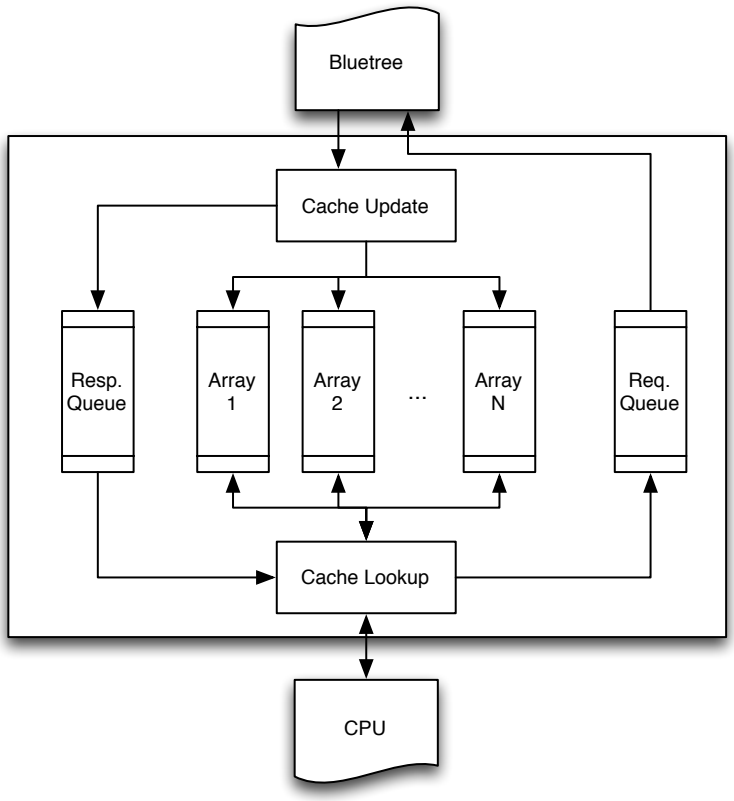


Figure 6.3: Diagram of the internals of the intermediate stream cache.

The second fix is to again provide an intermediate cache to prefetch into. A sufficiently small direct-mapped cache does still have the issue where a long prefetch stream (or a large subroutine which calls a small function and returns again) can overwrite the current prefetches in the cache though, causing a performance detriment and even potentially invalidate the WCET calculation. Instead, a new design for the intermediate cache is presented which better fits into the model of stream prefetching used here. This cache uses a set of "arrays", each of which stores a fixed number of blocks of sequential data, as in the block diagram shown in Figure 6.3. When data is delivered to the cache, if the data is a prefetch (or a prefetch squash), then each of these arrays is checked to determine if the new data is one part of one of the already stored sequential streams. If so, it is added to the end, and if not, a replacement candidate is picked in round-robin order. Example pseudocode for this process can be found in Listing 6.1. On a request by the processor, the requested address is checked to determine whether it already exists in one of these arrays, and is returned from cache if so. Example pseudocode for this process can be found in Listing 6.2.

```
#define CACHE_WAYS 8
#define CACHE_DEPTH 1024

addr_t addresses[CACHE_WAYS] = {0};
addr_t sizes[CACHE_WAYS] = {0};
data_t data[CACHE_WAYS][CACHE_DEPTH] = {0};
int buffer_replacement = 0;

void handle_incoming_data(data) {
    // Only attempt to add prefetches
    if(!is_prefetch(data))
        return;

    int foundBuffer = -1;
    for(i = 0; i < CACHE_WAYS; i++) {
        if(data.address == addresses[i] + sizes[i]) {
            foundBuffer = 1;
            break;
        }
    }

    if(foundBuffer == -1) {
        // Replace an existing buffer
        addresses[buffer_replacement] = data.address;
        sizes[buffer_replacement] = 1;
        data[buffer_replacement][0] = data.data;
        buffer_replacement++;
    }
    else {
        data[foundBuffer][sizes[foundBuffer]] = data.data;
        sizes[foundBuffer]++;
    }
}
```

Listing 6.1: Pseudocode description of how the stream cache inserts incoming data.

In order to reduce the implementation overhead of these arrays, each array is a small piece of FPGA block-RAM. Two registers store the *base address* and *size* of an array, and hence the address to be accessed for any requested address is simply *address - base address*. In terms of FPGA resources, each array is reasonably cheap to implement and hence each array can be the same size as the maximum stream size to be prefetched at once. Multiplexing multiple arrays onto the input or output of the cache is more expensive, however, and so the *number* of arrays should be minimised.

In terms of timing, the prefetch cache is reasonably cheap to cross. From the point that a request enters the cache, it will take three cycles to either reply with the cached data, or to issue the request to memory. If a request is issued to memory, this cache causes another cycle of delay when the response is being delivered, in order for the response to cross the cache. For the purposes of this work, the cache should be treated as an extension to the requester, and the model of the processor

```
#define CACHE_WAYS 8
#define CACHE_DEPTH 1024

addr_t addresses[CACHE_WAYS] = {0};
addr_t sizes[CACHE_WAYS] = {0};
data_t data[CACHE_WAYS][CACHE_DEPTH] = {0};
int buffer_replacement = 0;

void handle_request(req) {
    int foundBuffer = -1;
    for(i = 0; i < CACHE_WAYS; i++) {
        if(req.address >= addresses[i] &&
           req.address < addresses[i] + sizes[i]) {
            // Found it. Reply with the correct data.
            reply_with_data(data[i][req.address - addresses[i]]);
            return;
        }
    }

    // Not found, issue to memory
    relay_request(req);
}
```

Listing 6.2: Pseudocode description of how the stream cache performs a lookup.

used should be modified to incorporate this extra latency into that of each memory access. From the perspective of the system model from Chapter 3, this simply causes the dispatch time of each memory access (i.e. $t_d(\omega_n^r)$) to increase by three cycles, and requires no modification to the remainder of the system.

Finally, the cache supports simultaneously inserting prefetches from memory and lookups from an attached processor. The base addresses and sizes of each array are stored in registers, but with bypass capability such that the "lookup" stage for a read can be notified of data which is currently being inserted. The next cycle for the lookup stage then issues a request to the array's block-RAM storage, where the final cycle then reads the requested data and returns it to the processor. Of course, any data which is being inserted as a result of a prefetch may write to an address which the lookup logic is attempting to read from. In this case, the block-RAMs are configured in read-first mode, which causes the cache read to complete before the replacing data is written to that location.

As before, this prefetch cache is located close to the processor, in-between the processor's standard caches and the memory interconnect, as detailed in Section 5.2.3.

6.1.3 Improving the Worst Case

Section 5.2.3 gives a prefetcher design where it can be predicted *which* processor a prefetch will be created for. Combined with an analysis of *when* the prefetcher will be able to fetch from Section 6.1.1 and the analysis of *what* will be prefetched from Section 6.1.2, it should now be possible to reduce the worst-case execution time using the prefetcher.

If an input to the tree is left unconnected, it will generate prefetch slots whenever requests from it would be granted service under normal conditions. For this reason, the time between prefetch slots being created from the multiplexer corresponds to the worst-case inter-request time for a given arbiter input, $\delta^{arb}(r)$, which for Bluetree corresponds to the worst-case time for a request to transit the first level of the tree. This time can be ascertained using a modified form of the analysis presented within Section 4.2.1, specifically from Equation (4.8), which instead finds the time until a packet has reached level $l-1$ in a tree with l levels:

$$\delta^{arb}(r) = \delta^{mem} \times \min_{t=0}^{\infty} t : L_l^P(t) = l-1 \tag{6.1}$$

Because the worst-case blocking from level $l-1$ of the tree to the top of the tree is then constant, it can be stated that the prefetcher will receive a prefetch slot originating from this tree input every $\delta^{arb}(r)$ cycles, in the worst-case. This is then equal to the worst-case time between two prefetches being initiated. Given that this figure is known, and that the prefetcher may be operating for up to n requesters, the worst-case time between two prefetches being initiated for any given processor is then derived as:

$$\Delta_{pf} = n \times \delta^{arb}(r) \tag{6.2}$$

This then forms a model of prefetching which can then be integrated into a worst-case analysis tool. The prefetching scheme dictates that the prefetcher, when possible, will always attempt to fetch the "next" set of instructions for any observed instruction fetch. Given a set of contiguous instructions, it is therefore possible to assert that a prefetch will be initiated, in the worst-case, Δ_{pf} cycles after the first instruction fetch of the set of contiguous instructions. This prefetch will then definitely fetch the "next" set of instructions, after the last observed instruction fetch, and will, in the worst-case, consume t_{wc}^{mem} cycles.

For this reason, if the prefetch completes during the processor would have issued a request for the prefetched instruction line, then the delay associated with the fetch of the cache line can be eliminated entirely. If the processor would have issued a request *during* the period in which the prefetcher is fetching the target

line, then the latency of the memory fetch is reduced by the difference. After this prefetch has been accounted for, the process can then repeat, using the start time of the previous prefetch as the "initial" access in the block.

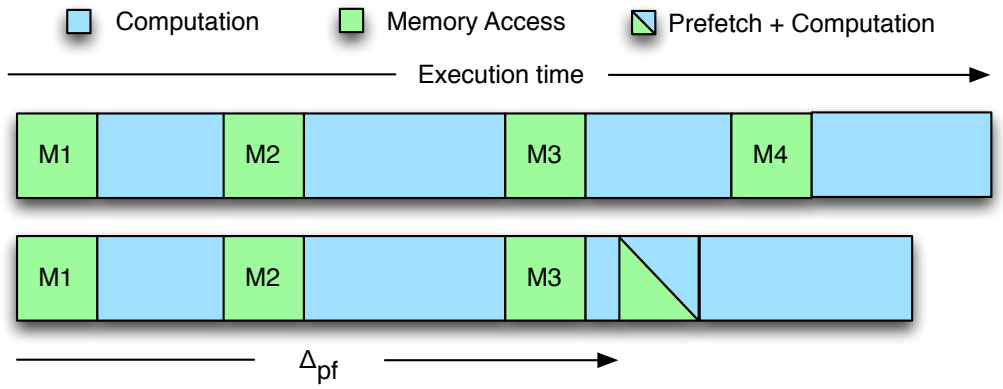


Figure 6.4: Example memory access scheduling with the prefetcher enabled.

Figure 6.4 shows an example of a prefetch being scheduled for a stream of instructions. This diagram shows four memory accesses (M1..M4), each separated by a period of computation. After Δ_{pf} cycles, the prefetcher has observed three memory accesses, and has an access for M4 available in its output buffers. After Δ_{pf} cycles, this prefetch is scheduled, and can complete in its entirety before M4 would normally be required. In this case, the access for M4 can be removed entirely, and the worst-case execution time updated to reflect this.

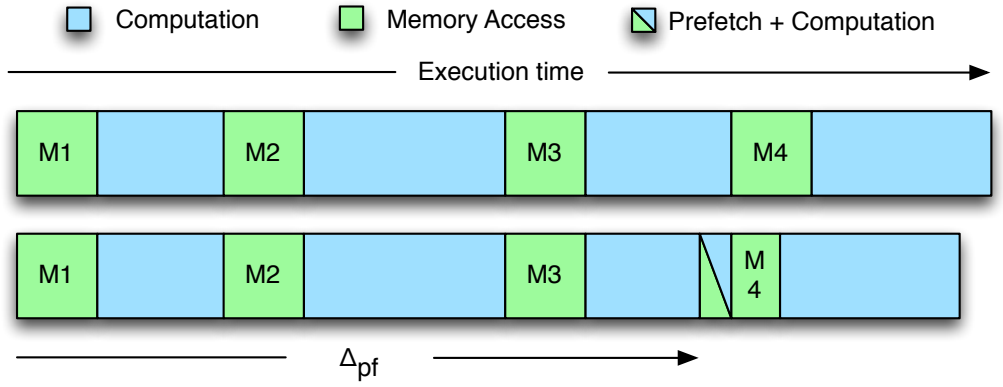


Figure 6.5: Example memory access scheduling with the prefetcher enabled, where the prefetch overlaps the demand access.

Figure 6.5 shows another possible case where Δ_{pf} is slightly larger. In this case, the processor requests M4 while a prefetch is already outstanding for it. Because the prefetcher can coalesce prefetches, the time in which the processor is blocked waiting for M4 to be delivered is slightly lower, and hence if the request for M4

is dispatched at t_{M4}, then the worst-case execution time is reduced by $t_{M4} - \Delta_{pf}$ cycles.

6.2 EVALUATION

Now that a technique to improve the execution time of a task through prefetching has been determined, it needs to be evaluated to determine the impact it may have on the execution time of a given task. To do this, the technique has been integrated into the OTAWA [99] worst-case analysis tool. This is a tool which analyses the machine code for a task, then by extracting the timing behaviour from a model of the task to be run, ascertains a worst-case bound for each basic block (or pairs of contiguous basic blocks), then combines these times together.

The impact of the prefetcher on the static analysis of a set of TACLeBench benchmarks will then be evaluated. In order to aid analysis, these benchmarks have already been annotated with the required loop bounds so that the analysis is feasible. In order to evaluated the performance of the new prefetching technique, a set of "worst-case" systems will again be built, and a set of the TACLeBench benchmarks executed on the platform. Since this work is only evaluating the effect of code prefetching, it does not make sense to execute the software traffic generators for this platform.

6.2.1 Static Analysis

Because the worst-case analysis of the system takes place on single basic-block at a time, there are two parameters which control the potential gains which can be observed given any block of code at a time. The first of these is the length of the basic block; if the basic block is longer, the WCET tool can analyse more code and hence, given a sufficiently long Δ_{pf}, a longer basic block may yield better results as more prefetches will definitely be able to be initiated in this period.

The other of these parameters is how long each fetched instruction takes to execute. Since many of the instructions in the Microblaze processor only take a single cycle to execute, the differentiating factor is how many memory accesses each block of code fetches from memory. If a block of code initiates a data memory access, then the execution time for that block will be longer, and again due to the longer length will allow more prefetches to be initiated.

Table 6.1 shows the potential saving which the prefetcher may bring to a hypothetical block of code under analysis, where prefetch slots are being generated from index 0, demand misses are being issued from core 1 and each memory access takes 50 cycles. This leads to a memory access response time of 850 cycles and a Δ_{pf} time of 4000 cycles.

	Zero loads		*One load*		*Two loads*	
BB Length	*Standard*	*Saving*	*Standard*	*Saving*	*Standard*	*Saving*
1	850	0	1700	0	2550	0
2	1700	0	3400	0	5100	0
...	...		...		...	
5	4250	0	8500	850	12750	1700
6	5100	250	10200	850	15300	1700
...	...		...		...	
10	8500	250	17000	2550	25500	3400
11	9350	500	18700	2550	28050	4250
...	...		...		...	
20	17000	750	34000	4400	51000	7700
21	17850	1000	35700	5250	53550	7700

	Three loads		*Four loads*	
BB Length	*Standard*	*Saving*	*Standard*	*Saving*
1	3400	0	4250	0
2	6800	0	8500	250
...	...		...	
5	17000	1700	21250	1000
6	20400	2550	25500	1250
...	...		...	
10	34000	4650	42500	2250
11	37400	5500	46750	2500
...	...		...	
20	68000	10150	85000	4750
21	71400	11000	89250	5000

Table 6.1: Potential WCET savings within the analysis by using the prefetcher, for varying numbers of cache lines in each basic block each with a given number of loads. Each cache line is assumed to be four words long.

In these results, *BB Length* refers to the size of the basic block under analysis in caches lines, where each cache line can hold four instructions. This then evaluates the potential saved cycles in the cases where each of these cache lines contains between zero and four memory access instructions. *Standard* then shows the execution time of this block with the prefetcher disabled, and *Saving* shows the number of cycles by which the worst-case can be improved by.

Of course, as the length of the basic block increases, the potential to prefetch increases as there is now sufficient time for the prefetch to actually be dispatched. Moreover, as the proportion of memory accesses increases, as does the potential gains. Again, this is because the memory accesses "create" time in which a prefetch can be successfully dispatched. Clearly, the prefetcher is most effective for tasks with long, straight-line code blocks (e.g. large math routines), or for data-bound applications with many reads from shared memory.

Figure 6.6 shows the potential improvement that is available when the tree input at index 0 (i.e. the one with the least blocking) is left unconnected. Many bench-

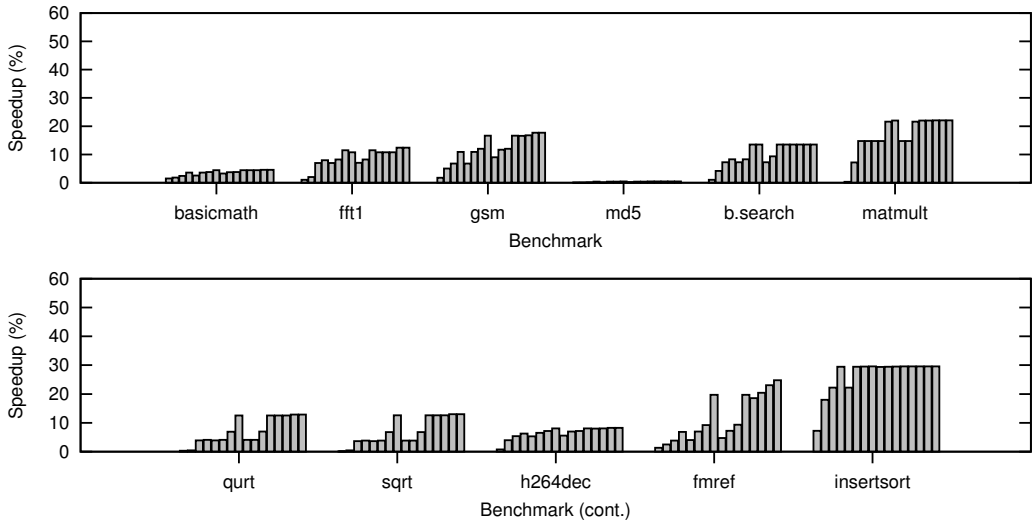


Figure 6.6: Improvement in the WCET for a number of benchmarks when the processor at index 0 was left un-connected.

marks yielded a good improvement (between zero and 30%) depending upon the access pattern of the task. Data-bound benchmarks with reasonably large routines, such as *gsm* and *h264dec* yielded good results, because the size of the basic blocks were large enough for something to be prefetched, and the data access could also allow a prefetch to be initiated.

Despite having large math routines, the *basicmath* benchmark did not yield as good results as other "large" benchmarks. This is because many of the math routines operate solely on registers, without accessing main memory much. This access pattern means that even with large basic blocks, there is not actually much potential for prefetching. Nevertheless, it could still observe improvements to the worst-case execution time.

Other benchmarks, for example *matmult* and *sqrt* are reasonably small, but still fetch some data to memory. Because of the size of these benchmarks, even one or two instruction lines being successfully prefetched translates to a very good improvement in their overall execution time.

Figure 6.7 then shows what happens if the input at index 1 (with a slightly higher amount of blocking) is instead left un-connected. Of course, this translates to a higher value for Δ_{pf}, and hence fewer prefetches will be dispatched in a given window. For sufficiently small basic blocks, or basic blocks which do not operate on memory, this may lead to no prefetches being dispatched at all where before they could be.

Many tasks therefore have a performance improvement, but not as much as when tree index 0 was left unconnected. Because of the latency for a prefetch to be dispatched, the cores with a low blocking term now typically do not experience a good improvement due to the fact that the basic blocks under inspection have typically been fully executed in less than Δ_{pf} cycles.

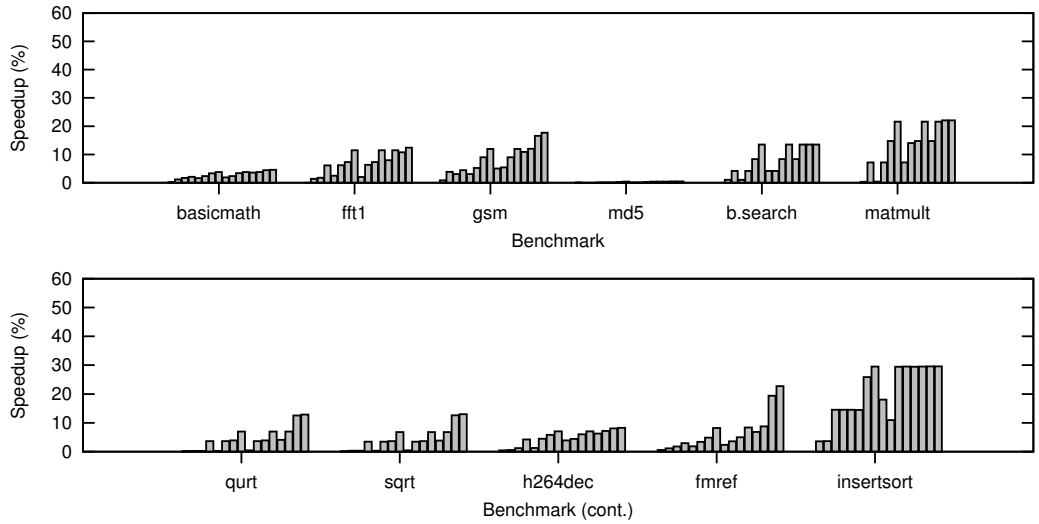


Figure 6.7: Improvement in the WCET for a number of benchmarks when the processor at index 1 was left un-connected.

Figure 6.8 finally shows the case where tree input 2 is left un-connected. The trend is largely the same as the difference between leaving index 0 and index 1 un-connected; a prefetch slot will experience more blocking before it is able to transit the lowest level of the tree, hence Δ_{pf} is larger and hence fewer prefetches will be able to be initiated in any given time interval. Many of the tree indices with less blocking are now unable to observe an increase to the worst-case execution time, again, because the basic blocks under inspection complete before Δ_{pf} cycles have elapsed, in the worst case.

6.2.2 Worst-Case Hardware System

In order to evaluate the performance benefits of the new prefetching system on a task under "worst-case" conditions, the prefetcher was also synthesized into a hardware system and placed next to the memory controller, as before. The system was then evaluated with the same set of benchmarks as in Section 6.2.1 and the performance improvement recorded when the prefetcher was enabled. Again, as the prefetcher is only used for code prefetching, the software traffic generators were not used, as they are data-side benchmarks. The experimental evaluation is the same as in Section 5.3.1; fifteen hardware traffic generators were used, each of which requests on every possible cycle to simulate the worst case, while a Microblaze processor was connected to the final input to run the benchmark.

Figure 6.9 shows the realised performance increase when slots were being generated from index 0 on the tree. Here, many tasks realise huge improvements from utilising these generated slots for prefetching. Because index 0 is left unconnected, a great amount of prefetch slots are generated reasonably quickly. Moreover, ex-

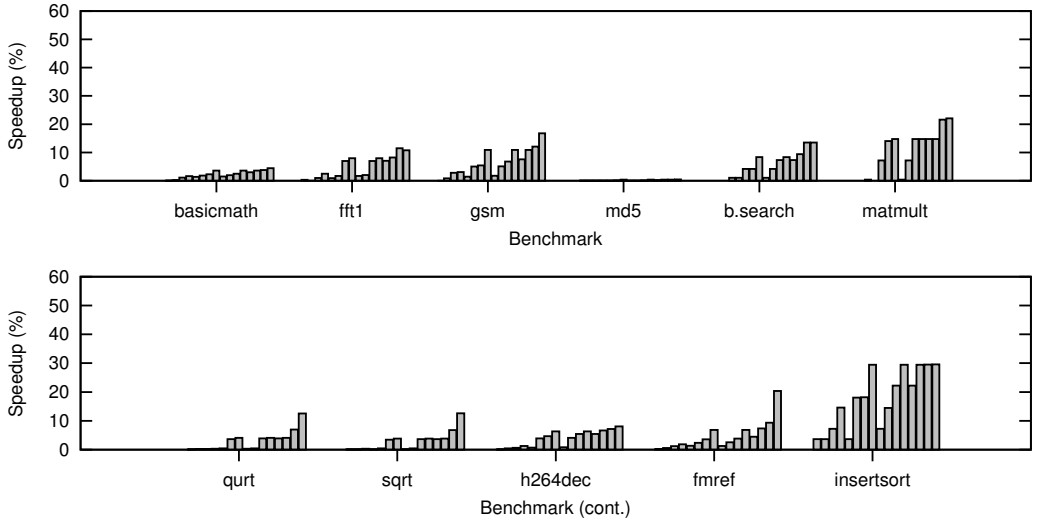


Figure 6.8: Improvement in the WCET for a number of benchmarks when the processor at index 2 was left un-connected.

ecuting the task in this manner allows the prefetcher to also yield slots from the task itself not fully utilising its allocated bandwidth budget.

Because the prefetcher is operating on the code side only, tasks with large basic blocks yield very good results. The first example is *basicmath*, which contains large math routines. Because of the size of these routines, the prefetcher can accurately predict a large number of the potential memory accesses, and hence the benchmark yields very good results. *md5* also uses reasonably large routines, and hence yields a good performance increase.

Data bound applications, for example *gsm* and *h264dec* still yield good benefits due to still having reasonably large routines for audio and video decoding, but due to the large number of data accesses (which are not prefetched), only yield a 20-40% increase to their execution times.

Many smaller benchmarks (for example, *qurt* and *sqrt*) may still realise good benefits due to being reasonably straight-line, hence the prefetcher can operate well, but due to their size even a few accurate prefetches will dramatically change the execution time for the better. *matmult* is another small benchmark which can benefit by the program being accurately prefetched, but does not yield huge performance increases due to the number of iterations of this small kernel, which is resident in cache.

Many benchmarks show a similar "shape" for the performance increase. This is simply due to the amount of blocking which tasks experience while waiting for memory accesses. Tasks which are blocked for longer will typically not coalesce their prefetches, and of course have much higher gains to be realised by the inclusion of the prefetcher and see a better performance increase. Data bound applications, however, typically can only be improved to a point and show a "flatter"

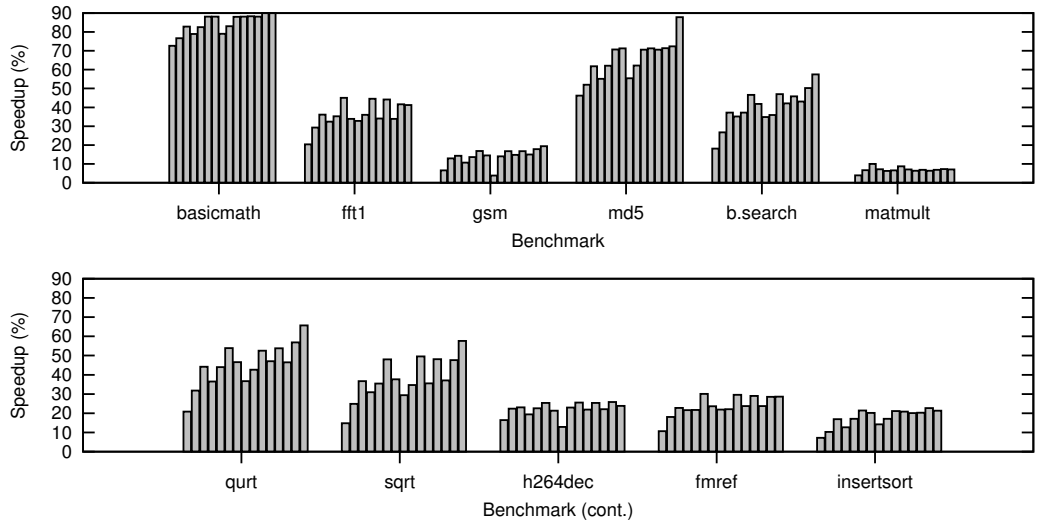


Figure 6.9: The performance improvement from the prefetcher being enabled under "worst-case" conditions.

shape. This is simply because the waiting time for data access to complete consumes most of the time, hence forming a limit on the effectiveness of the prefetcher.

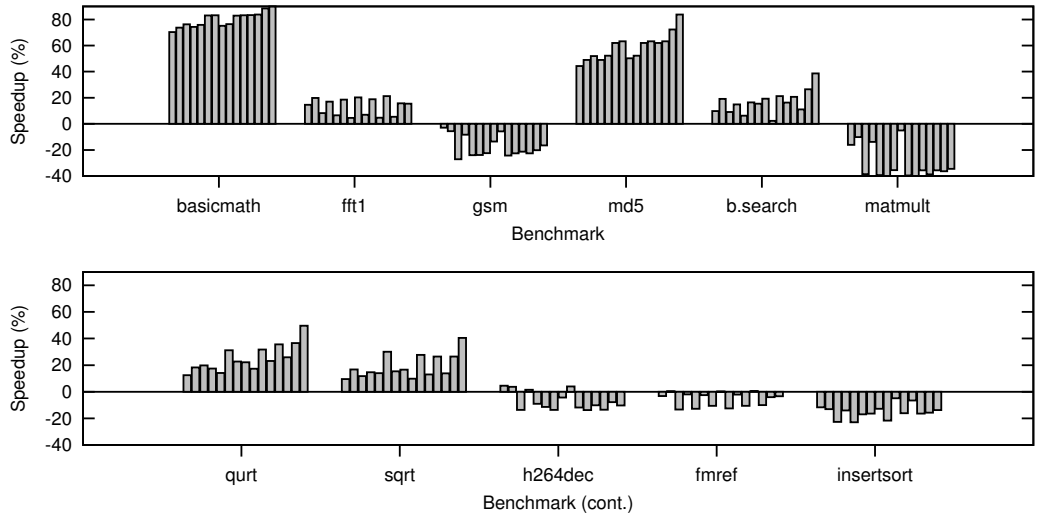


Figure 6.10: The performance improvement from the prefetcher being enabled under "worst-case" conditions, compared with work conservation

Of course, within a real system, it would be possible to simply use these prefetch slots as work-conserving accesses, and effectively share the "spare" bandwidth amongst all other requesters. Figure 6.10 shows the performance increase that the prefetcher can yield when compared with work-conservation.

The potential performance improvements again largely depend upon the access pattern of the task being executed. Because the prefetcher can speculatively issue memory accesses, code-bound applications (e.g. *basicmath* and *md5*) yield very

good performance improvements when compared to work-conservation, as do some of the smaller benchmarks (e.g. *sqrt* and *qurt*).

On the other hand, any data bound applications, for example, *h264dec* and *gsm* yield better performance improvements by the utilisation of work-conservation. This is simply because of the number of (unprefetchable) data loads to main memory which take place. In this case, work-conservation will perform better, since the "spare" bandwidth can be allocated to both the data loads and the instruction loads, and due to the large number of data loads, the net result is better system performance under work-conservation. Due to the very small size of the *matmult* benchmark, the prefetcher will successfully fetch the whole benchmark, but will also continue past the benchmark fetching useless data. This slows down any memory accesses which it may perform, and hence work-conservation again performs better as it will only fetch data which is definitely required.

6.3 SUMMARY

This chapter presents a system design, building on top of the work within Chapter 5, which facilitates predictable prefetch. This is achieved by exploiting the inherent access pattern of program code to simplify the internal state of the prefetcher such that it can be accurately predicted what will be prefetched for any given stream of code. Given the slot-based system also presented within Chapter 5, it then provides a method to ascertain the worst-case time between two prefetch slots arriving at the memory controller and hence provides a system to ascertain what will be prefetched and when, for each processor in the system.

This methodology gives a good improvement to the worst-case execution time of a task when integrated into a static analysis tool, realising a performance of up to 30%, depending on the access pattern of the benchmark and the period between slots being received by the prefetcher. Moreover, the prefetch yields very good performance increases of up to 90% when evaluated in a hardware system, and even improvements of up to 80% when compared with standard work-conservation because of the speculative nature of the prefetcher.

This approach does require an amount of "spare" bandwidth to be known about ahead of time. This may be known about ahead of time; tasks within a composable system do not necessarily consume all available system bandwidth. Moreover, tasks may be added or removed from the system at run time, and hence a system may provide service levels for a task, or different WCET guarantees, depending on the currently allocated portion of the bandwidth. While this does break composability, it is reasonably simple to analyse and does not require an expensive analysis of the interactions of a set of tasks.

Of course, this "spare" bandwidth can instead be used to reduce the worst-case of a task through reducing the worst-case blocking which the task can experience because of work-conservation. While this is technically possible, the work-conserving behaviour of an arbiter is rarely known, and typically depends upon the access pattern of the other tasks in the system, hence again breaks composability and requires an expensive system analysis. This approach provides a good alternative which, by issuing a memory access ahead of time at a known time from a separate functional unit, can improve the analytical worst-case execution time of a task and in some cases, perform better than work-conservation due to its speculative nature.

7 CONCLUSIONS AND FUTURE WORK

To recall from Section 1.2, this thesis attempted to prove the following hypothesis:

> A prefetcher can be constructed in such a way that it can be used within a real-time system in a predictable way such that it does not cause any detriment to the worst-case execution time of the tasks running within that system. Furthermore, this methodology to predict when the prefetcher will operate, it is possible to utilise "spare" or unallocated bandwidth within the system to actively improve the worst-case execution time estimate of a task running in a real-time system by using a predictable prefetcher.

This thesis has provided a framework by which a prefetcher can be integrated into a system utilising composable arbitration without modification to the system analysis in order to improve the average-case execution time of a task without any harm to the worst-case execution time of a task. It then moves on to define a predictable bandwidth source which can then be combined with the system model in order to provide a safe and analysable *improvement* to the worst-case execution time of a task.

This work provides a valuable method by which a predictable prefetcher can therefore be used to slow the ever-growing memory latencies found in modern multi-core real-time systems given a static amount of "spare" system bandwidth known ahead of time. Moreover, this work can also be used to define service levels based upon the current difference between the total available system bandwidth and the bandwidth assigned to requesters by the arbiter.

7.1 CONTRIBUTIONS

The contributions of this thesis can be summed up as follows, organised by chapter:

CHAPTER 4 provides an insight into the behaviour of a standard, off the shelf prefetcher on a system currently under worst-case conditions. This chapter shows that while under "perfect" conditions (i.e. a stream of accesses which can be perfectly predicted) the prefetch provides a good improvement to the execution time of a task, it quickly provides a detriment to the execution time

of a task as soon as it starts mis-predicting memory accesses. This chapter then goes on to show that the prefetcher can cause a slowdown of up to 70% to the worst-case execution time of a task, an effect which is clearly unacceptable within a real-time system.

CHAPTER 5 builds upon these results to provide a model by which the prefetcher can be notified of slack time in the system and by which the arbiter and requesters in the system can generate this feedback. Since this feedback is generated when the worst-case analysis would expect a request to be present, but it is missing, the prefetcher can operate in such a way that it does not exceed the interference which is predicted by the worst-case analysis. From this, the prefetcher can thus operate and improve the average-case timing of a real-time system without any detriment to the worst-case execution times of real-time tasks.

CHAPTER 6 finally identifies locations in the system where "spare" bandwidth can be predictably sourced from, which can then be used in the above system model. Because a predictable amount of "spare" bandwidth can be identified, it is then possible to identify the worst-case time between the prefetcher being able to issue two prefetches. This chapter then provides a more predictable model of prefetch for a task's instructions and hence provides a system by which it can be predicted what will be prefetched for a given task. The combination of being able to identify what will be prefetched and when then allows a task's worst-case execution time to be improved by the inclusion of the prefetcher.

These contributions together provide a system which, under certain conditions, allow a prefetcher to provide an improvement to the analysed worst-case execution time of a task, and hence prove the hypothesis that a prefetcher can be constructed to operate in a sufficiently predictable manner such that the worst-case execution time of a real-time task can be improved.

7.2 CONCLUSIONS

The first major conclusion of this work relates to the first part of the thesis hypothesis. Through modification to the system arbiter and prefetcher, it is possible to extend the system in such a way that the prefetcher can operate in a safe manner without any detriment to the worst-case execution time of the system. This works well; Chapter 5 demonstrates that the prefetcher can yield an average-case execution time improvement of up to 30%, while guaranteeing that the prefetcher will not cause a detriment to the worst-case execution time analytically and also demonstrating the fact in a real system.

This method is not without its limitations, however. The first of these is that the feedback required for the prefetcher to operate must still travel over the same arbitration system as "standard" memory accesses. While it can be asserted that these feedback messages do not harm the worst-case execution time, it does yield problems for low-priority requesters as the feedback messages will be blocked for a long period of time. Because it is these messages which stimulate the prefetcher to issue the "next" prefetch, it means that low-priority requesters do not see a great performance improvement, while only making the higher priority requesters even faster. For the most part though, this is a limitation of using a fairly "standard" prefetching heuristic; a smarter or more aggressive prefetching scheme such as that used in Chapter 6. Equally, the system could be modified such that prefetch notification messages are passed through a different memory network, and hence experience much less interference, although this will of course have a higher hardware overhead.

The other limitation is that this technique requires a small "prefetch cache" to be utilised so that prefetches cannot displace data from the processor's cache which is required in the near future. While this requires extra hardware, the overheads are reasonably small. Ideally, this prefetch cache only requires as many entries as there are stream buffers inside the prefetcher; as the prefetch "hit" notification is only created when there is a prefetch hit, new prefetches for a stream will not overwrite old ones because the condition for the new prefetch being initiated is that the old prefetched data has been read. Of course though, if the hardware overhead must be minimised, this cache can be reduced to only hold the most recent prefetch, although this may cause other prefetches to be lost.

Despite these two limitations, this poses a valuable technique to improve the average-case execution time of real-time tasks, and should help to maintain system responsiveness in the face of rising memory latencies. While this technique has limited applicability for systems comprising of only hard real-time tasks, it may be useful in systems with both hard and non real-time tasks; by improving the execution times of all tasks in the system, there is potentially more time available to run the non real-time tasks.

By building a predictable and aggressive prefetcher and outlining a method to predictably generate "spare" bandwidth within the memory system, Chapter 6 then presents a method by which the prefetcher can be used to improve the worst-case execution time of a task. This again works quite well, with some benchmarks gaining a 20-30% increase to their worst-case execution times, depending on the amount of "spare" bandwidth available.

Of course, the limitation of this technique is that it requires an amount of spare bandwidth to operate which must be ascertained ahead of time. Clearly, this bandwidth could instead be shared amongst the current requesters rather than be used for prefetching, although as shown in the comparison of work-conservation and

prefetching towards the latter part of Chapter 6, different tasks benefit from different methods; as prefetching can be used to completely eliminate the time taken to access memory, it can potentially yield better performance. Moreover, leaving "spare" bandwidth in this manner is just one method by which prefetch slots can be generated; the further work in Section 7.3 will identify other sources of prefetch slots which could be explored in further work to allow the worst-case to be improved *without* leaving "spare" bandwidth.

Despite this limitation, the technique as a whole works well and can analytically improve the worst-case execution time of a hard real-time task. The rising memory latencies and increased memory contention is always going to be an issue, but a technique such as prefetch can, and has been shown to, take steps to mitigate this rising memory latency. Ultimately, this proves the thesis hypothesis to be true; a prefetcher has been constructed which can both be used in a real-time system without any performance detriment, and can also improve worst-case system performance under certain constraints.

7.3 FURTHER WORK

Of course, there are many avenues through which this work can be extended in future to further improve the results, or to analyse the current results even further.

The first of these should attempt to extend the new prefetching methodology further in order to also consider the data side prefetching when improve the worst-case execution time. This may be done in a few different ways; if a loop accesses sequential elements, value analysis may be used to determine whether the sequential elements can be prefetched and hence, given a worst-case time between two of the load instructions being executed can be ascertained, determine whether a prefetch for a data item will complete in time. Other techniques could use a variant of persistence analysis to predict the state of the prefetcher at any point within a task's execution, similar to how cache analysis works. This can then be used to ascertain which data accesses would be prefetched and thus further improve the worst-case execution time.

Secondly, in order to improve the worst-case execution time through data-side prefetching, a new data-side prefetching scheme should be developed in order to make it easier to predict what would be prefetched. There are many methods which can be used to improve the predictability of this; streams of data could be tagged ahead of time to inform the prefetcher of what can be prefetched, then a similar scheme to the presented code prefetching (i.e. start fetching all subsequent entries after the first has been hit) can be used to fetch sequential stream elements. Equally, software managed prefetching could be used to notify the prefetcher of what data could be fetched next, effectively creating an "asynchronous read" in-

struction within a processor. This can then be filled in by the compiler to fetch data ahead of time. These techniques may also be advantageous to improve the accuracy of code prefetching.

Better results still could be yielded by being able to generate prefetch slots without having to leave one of the arbiter's inputs un-connected. As mentioned within Chapter 6, it may also be possible to predict how many prefetch slots are generated due to a task not fully utilising its bandwidth bounds and instead use these prefetch slots to improve the worst-case execution time of a task. One method to do this could be to constrain prefetch slots to a specific requester, only allowing them to form a prefetch for those requesters. Using a technique to build a model of how many memory requests a task issues in a period of time (e.g. from [22, 23]), it is also possible to predict how many prefetch slots are generated. If it is acceptable to break composability, it may also be possible to allow these prefetch slots to improve the worst-case execution time of *all* tasks within the system, thus allowing low-priority tasks with a large amount of blocking to also benefit.

Finally, any technique which is using the "spare" bandwidth of an arbiter is always going to be contrasted against work conservation. The latter stages of Chapter 6 does perform a contrast between prefetching and work conservation under "worst-case" conditions, showing that both techniques can perform well for different task sets. This comparison could even be taken further, leading to a hybrid approach of work conservation and prefetching. As an example, each prefetch slot could also contain the details of a work-conserving access if one is available. The prefetcher can then attempt to split the "spare" bandwidth between work conservation and prefetching to investigate if the combination of the techniques yields a greater performance increase than using one of the techniques alone.

7.4 CLOSING REMARKS

Shared memory causes a great bottleneck in many systems; memory controllers are being shared between an increasing number of processing cores as the multi-core scaling of modern systems only continues. One major conclusion that can be taken from this observation is simply that shared memory is not the correct paradigm to use for modern systems, and that instead local memories and explicit communication should be used. Which such techniques will improve performance by reliving some pressure on memory, shared memory simply cannot be eliminated as the memory requirements of many modern tasks far exceed what can be provided with smaller, private memories. The cost of implementing larger core-local memories, or a DDR memory per core is simply not cost effective with modern manufacturing technologies.

Shared memory is still a very useful paradigm for tasks requiring large amounts of storage, or for many tasks which need to access the same, large bank of data. There are many methods by which the latency problem can be alleviated; some systems are moving towards integrating many DDR controllers on their chips (for example, Tilera [100] chips currently utilise four separate DDR controllers). By partitioning the memory between all requesters, the effective pressure on each memory controller is reduced. Other techniques can raise the request granularity to move more data in a single transaction, increasing the response time of each request but delivering much more data at once.

Importantly, these techniques are not mutually exclusive with the prefetching technique provided within this thesis. This thesis has shown that the ever-growing latencies associated with accessing a shared memory can be slowed by also including a prefetcher within the system. It can then also be combined with a partitioned memory controller, or a larger request granularity in order to further improve the performance and scalability of a multi-core system.

BIBLIOGRAPHY

[1] J. Hennessy and D. Patterson, *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann, 4th ed., 2006.

[2] R. Wilhelm, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, and R. Heckmann, "The worst-case execution-time problemâĂŤoverview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, pp. 1–53, Apr. 2008.

[3] Integrated Circuit Engineering Corporation, *Memory*. 1997.

[4] B. Jacob, S. W. Ng, and D. T. Wang, *Memory Systems: Cache, DRAM, Disk*. 2008.

[5] U. Drepper, "What every programmer should know about memory," *http://people.redhat.com/drepper/cpumemory.pdf*, 2007.

[6] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of Network-on-chip," *ACM Computing Surveys*, vol. 38, pp. 1–51, June 2006.

[7] Intel Corporation, "Even Higher Efficiency for Parallel Processing," 2015.

[8] B. Akesson, A. Hansson, and K. Goossens, "Composable Resource Sharing Based on Latency-Rate Servers," in *2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, pp. 547–555, IEEE, Aug. 2009.

[9] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens, "Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration," *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 3–14, Aug. 2008.

[10] M. Schoeberl, D. V. Chong, W. Puffitsch, and J. Sparso, "A Time-Predictable Memory Network-on-Chip," *14th International Workshop on Worst-Case Execution Time Analysis*, pp. 53–61, 2014.

[11] Micron Technology Inc, "1Gb DDR3 SDRAM Features," 2006.

[12] M. D. Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens, "A Generic, Scalable and Globally Arbitrated Memory Tree for Shared DRAM Access in Real-Time Systems," in *Proceedings 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 193–198, 2014.

[13] D. Frank, R. Dennard, E. Nowak, P. Solomon, and Y. Taur, "Device scaling limits of Si MOSFETs and their application dependencies," *Proceedings of the IEEE*, vol. 89, pp. 259–288, Mar. 2001.

[14] EZChip, "TILE-Gx72 Processor Product Brief," 2015.

[15] Intel Corporation, "The Intel Xeon Phi Product Family," *Product Brief*, 2013.

[16] D. Burger, J. R. Goodman, and A. Kägi, "Memory bandwidth limitations of future microprocessors," *ACM SIGARCH Computer Architecture News*, vol. 24, pp. 78–89, May 1996.

[17] G. Moore, "Cramming More Components Onto Integrated Circuits," *Proceedings of the IEEE*, vol. 86, pp. 82–85, Jan. 1998.

[18] M. Bohr, "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper," *IEEE Solid-State Circuits Newsletter*, vol. 12, pp. 11–13, Jan. 2007.

[19] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. Jane Irwin, M. Kandemir, and V. Narayanan, "Leakage Current: Moore's Law Meets Static Power," 2003.

[20] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," 2009.

[21] R. Ho, K. Mai, and M. Horowitz, "The future of wires," *Proceedings of the IEEE*, vol. 89, pp. 490–504, Apr. 2001.

[22] R. Pellizzoni, A. Schranzhofer, J.-j. Chen, M. Caccamo, and L. Thiele, "Memory Interference Delay Estimation for Multicore Systems," tech. rep., 2009.

[23] D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee, "Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus," *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 1068–1075, Nov. 2011.

[24] A. Hansson, M. Coenen, and K. Goossens, "Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip," *Proceedings -Design, Automation and Test in Europe, DATE*, pp. 954–959, 2007.

[25] Micron Technology Inc, "DDR4 SDRAM Features," 2012.

[26] K. Kaplan and R. Winder, "Cache-based Computer Systems," *Computer*, vol. 6, pp. 30–36, Mar. 1973.

[27] R. Wilhelm, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat,

C. Ferdinand, and R. Heckmann, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, pp. 1–53, Apr. 2008.

[28] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*. Guildford and King's Lynn: Addison-Wesley, third ed., 2001.

[29] P. Puschner and A. Burns, "Guest editorial: A review of worst-case execution-time analysis," *Real-Time Systems*, vol. 18, no. 2-3, pp. 115–128, 2000.

[30] P. Cousot and R. Cousot, "Abstract interpretation," *ACM Computing Survey*, vol. 28, pp. 324–328, 1996.

[31] J. Engblom, *Processor Pipelines and Static Worst-Case Execution Time Analysis*. 2002.

[32] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm, "Reliable and precise WCET determination for a real-life processor," *Embedded Software*, vol. 2211/2001, pp. 469–485, 2001.

[33] C. Ferdinand and R. Wilhelm, "On predicting data cache behavior for real-time systems," in *Languages, Compilers, and Tools for Embedded Systems*, pp. 16–30, Springer, 1998.

[34] G. Bernat, A. Colin, and S. Petters, "pWCET : a Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Systems," pp. 1–18, 2003.

[35] W. Lunniss, S. Altmeyer, C. Maiza, and R. I. Davis, "Integrating cache related pre-emption delay analysis into EDF scheduling," *Real-Time Technology and Applications - Proceedings*, pp. 75–84, 2013.

[36] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[37] D. E. Kotecki, J. D. Baniecki, H. Shen, R. B. Laibowitz, K. L. Saenger, J. J. Lian, T. M. Shaw, S. D. Athavale, C. Cabral, P. R. Duncombe, M. Gutsche, G. Kunkel, Y.-J. Park, Y.-Y. Wang, and R. Wise, "(Ba,Sr)TiO_3 dielectrics for future stacked-capacitor DRAM," *IBM Journal of Research and Development*, vol. 43, pp. 367–382, May 1999.

[38] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A Predictable SDRAM Memory Controller," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis - CODES+ISSS '07*, (New York, New York, USA), p. 251, ACM Press, 2007.

[39] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero, "An Analyzable Memory Controller for Hard Real-Time CMPs," *IEEE Embedded Systems Letters*, vol. 1, pp. 86–90, Dec. 2009.

[40] B. Akesson, W. Hayes Jr., and K. Goossens, "Classification and Analysis of Predictable Memory Patterns," in *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 367–376, IEEE, Aug. 2010.

[41] B. Akesson, W. Hayes Jr., and K. Goossens, "Automatic Generation of Efficient Predictable Memory Patterns," in *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 177–184, IEEE, Aug. 2011.

[42] B. Akesson, P.-c. Huang, F. Clermidy, D. Dutoit, K. Goossens, Y.-h. Chang, T.-w. Kuo, P. Vivet, and D. Wingard, "Memory controllers for high-performance and real-time MPSoCs," in *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES+ISSS '11*, (New York, New York, USA), p. 3, ACM Press, 2011.

[43] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens, "Real-Time Scheduling of Hybrid Systems using Credit-Controlled Static-Priority Arbitration," tech. rep., NXP Semiconductor, Aug. 2008.

[44] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proceedings of the 27th International Symposium on Computer Architecture, 2000.*, pp. 128–138, 2000.

[45] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus IPC," *ACM SIGARCH Computer Architecture News*, vol. 28, pp. 248–259, May 2000.

[46] ARM, "AMBA Specification," 1999.

[47] Arm, *AMBA AXI and ACE Protocol Specification*. 2011.

[48] W. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pp. 684–689.

[49] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost, "HERMES: an infrastructure for low area overhead packet-switching networks on chip," *Integration, the VLSI Journal*, vol. 38, pp. 69–93, Oct. 2004.

[50] K. Goossens, J. Dielissen, and A. Radulescu, "Æthereal Network on Chip:Concepts, Architectures, and Implementations," *IEEE Design and Test of Computers*, vol. 22, pp. 414–421, May 2005.

[51] J. H. Rutgers, M. J. G. Bekooij, and G. J. M. Smit, "Evaluation of a connectionless NoC for a real-time distributed shared memory many-core system," *Proceedings - 15th Euromicro Conference on Digital System Design, DSD 2012*, pp. 727–730, 2012.

[52] Parallella, "Epiphany Architecture Reference," 2013.

[53] A. Agarwal, "The Tile Processor : A 64-Core Multicore for Embedded Processing Markets Demanding More Performance," 2007.

[54] Intel Corporation, "An Introduction to the Intel QuickPath Interconnect," 2009.

[55] D. Stiliadis and A. Varma, "Latency-rate servers: a general model for analysis of traffic scheduling algorithms," *IEEE/ACM Transactions on Networking*, vol. 6, no. 5, pp. 611–624, 1998.

[56] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 1997.

[57] K. Tindell, A. Burns, and A. Wellings, "Mode changes in priority preemptively scheduled systems," in *[1992] Proceedings Real-Time Systems Symposium*, pp. 100–109, IEEE Comput. Soc. Press, 1992.

[58] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis, "Weighted round-robin cell multiplexing in a general-purpose ATM switch chip," *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 8, pp. 1265–1279, 1991.

[59] B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems*.

[60] G. Plumbridge, J. Whitham, and N. Audsley, "Blueshell : A Platform for Rapid Prototyping of Multiprocessor NoCs and Accelerators," in *Proceedings HEART Workshop*, University of York, 2013.

[61] J. Garside and N. Audsley, "WCET Preserving Hardware Prefetch for Many-Core Real-Time Systems," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems - RTNS '14*, (Versailles, France), pp. 193–202, ACM Press, 2014.

[62] J. Tendler, J. Dodson, and J. Fields, "POWER4 system microarchitecture," *IBM Journal of*, vol. 46, no. 1, pp. 5–25, 2002.

[63] Intel Corporation, "Intel 64 and IA-32 Architectures Optimization Reference Manual," 2015.

[64] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," *ACM SIGARCH Computer Architecture News*, vol. 19, no. 2, pp. 40–52, 1991.

[65] J. Lee, H. Kim, and R. Vuduc, "When Prefetching Works, When It Doesn'âĂŹt, and Why," *ACM Transactions on Architecture and Code Optimization*, vol. 9, pp. 1–29, Mar. 2012.

[66] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pp. 364–373, IEEE Comput. Soc. Press, 1990.

[67] J.-L. Baer and T.-F. 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, New York, USA), pp. 176–186, ACM Press, 1991.

[68] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," *ACM SIGMICRO Newsletter*, vol. 23, pp. 102–110, Dec. 1992.

[69] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *Proceedings of the 24th annual international symposium on Computer architecture - ISCA '97*, (New York, New York, USA), pp. 252–263, ACM Press, 1997.

[70] K. Nesbit and J. Smith, "Data Cache Prefetching Using a Global History Buffer," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pp. 96–96, IEEE, 2004.

[71] A. Roth and G. Sohi, "Effective jump-pointer prefetching for linked data structures," in *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, pp. 111–121, IEEE Comput. Soc. Press, 1999.

[72] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism," *ACM SIGPLAN Notices*, vol. 37, p. 279, Oct. 2002.

[73] J. Collins, S. S. Sair, B. Calder, and D. M. Tullsen, "Pointer Cache Assisted Prefetching," in *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pp. 62–73, 2002.

[74] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 63–74, 2007.

[75] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "APOGEE : Adaptive Prefetching On GPUs for Energy Efficiency," in *22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013.

[76] I. Hur and C. Lin, "Memory Prefetching Using Adaptive Stream Detection," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pp. 397–408, IEEE, Dec. 2006.

[77] I. Hur and C. Lin, "Feedback mechanisms for improving probabilistic memory prefetching," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pp. 443–454, IEEE, Feb. 2009.

[78] P. Yedlapalli, J. Kotra, E. Kultursay, M. Kandemir, C. R. Das, and A. Sivasubramaniam, "Meeting Midway: Improving CMP Performance with Memory-Side Prefetching," in *22nd International Conference on Parallel Architectures and Compilation Techniques*, pp. 289–298, 2013.

[79] L. A. R. Yang Chia-Lin, "Push vs. pull: Data movement for linked data structures," *Proceedings of the International Conference on Supercomputing*, pp. 176–186, 2000.

[80] F. Dahlgren, M. Dubois, and P. Stenstrom, "Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors," in *1993 International Conference on Parallel Processing - ICPP'93 Vol1*, pp. 56–63, IEEE, Aug. 1993.

[81] D. M. Tullsen and S. J. Eggers, "Limitations of cache prefetching on a bus-based multiprocessor," *Proceedings of the 20th annual international symposium on Computer architecture - ISCA '93*, pp. 278–288, 1993.

[82] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture - Micro-42*, (New York, New York, USA), p. 316, ACM Press, 2009.

[83] F. Liu and Y. Solihin, "Joint Exploration of Hardware Prefetching and Bandwidth Partitioning in Chip Multiprocessors," *eecs.umich.edu*.

[84] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, 2015.

[85] J. Gaisler, "A portable and fault-tolerant microprocessor based on the SPARC V8 architecture," *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pp. 409–415, 2002.

[86] R. Pellizzoni, A. Schranzhofer, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," *2010 Design, Au-*

tomation & Test in Europe Conference & Exhibition (DATE 2010), pp. 741–746, Mar. 2010.

[87] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "CoMPSoC: A template for composable and predictable multi-processor system on chips," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14, no. 1, p. 2, 2009.

[88] Xilinx, "MicroBlaze Processor Reference Guide Embedded," 2003.

[89] M. Lee, S. Min, C. Park, Y. Bae, H. Shin, and C. Kim, "A dual-mode instruction prefetch scheme for improved worst case and average case program execution times," in *1993 Proceedings Real-Time Systems Symposium*, pp. 98–105, IEEE Comput. Soc. Press, 1993.

[90] J. Carter, W. Hsieh, L. Stoller, M. Swanson, E. Brunvand, A. Davis, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama, "Impulse: building a smarter memory controller," in *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, pp. 70–79, IEEE, 1999.

[91] U. Guide, "VC709 Evaluation Board for the Virtex-7 FPGA," 2015.

[92] J. Muttersbach, T. Villiger, and W. Fichtner, "Practical design of globally-asynchronous locally-synchronous systems," in *Proceedings - International Symposium on Asynchronous Circuits and Systems*, pp. 52–59, 2000.

[93] Xilinx, "7 Series FPGAs Memory Interface Solutions," 2011.

[94] "TACLeBench," 2013.

[95] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen WCET benchmarks: Past, present and future," in *10th International Workshop on Worst-Case Execution Time Analysis*, pp. 136–146, 2010.

[96] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pp. 3–14, 2001.

[97] C. Lee, M. Potkonjak, and W. H. Mangione-smith, "MediaBench : A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," in *MICRO 30 Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pp. 330–335, 1997.

[98] J. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems," in *[1992] Proceedings Real-Time Systems Symposium*, pp. 110–123, IEEE Comput. Soc. Press, 1992.

[99] H. Cassé and P. Sainrat, "OTAWA , a Framework for Experimenting WCET Computations," No. January, pp. 1–8, 2006.

[100] Tilera, "Tile Processor User Architecture Manual," 2011.