Lossy Compression applied to the Worst Case Execution Time Problem

David Jack Griffin

PhD

University of York
Computer Science

September 2013

Abstract

Abstract Interpretation and Symbolic Model Checking are powerful techniques in the field of testing. These techniques can verify the correctness of systems by exploring the state space that the systems occupy. As this would normally be intractable for even moderately complicated systems, both techniques employ a system of using approximations in order to reduce the size of the state space considered without compromising on the reliability of the results. When applied to Real-time Systems, and in particular Worst Case Execution Time Estimation, Abstract Interpretation and Symbolic Model Checking are primarily used to verify the temporal properties of a system. This results in a large number of applications for the techniques, from verifying the properties of components to the values given variables may take. In turn, this results in a large problem area for researchers in devising the approximations required to reduce the size of the state space whilst ensuring the analysis remains safe.

This thesis examines the use of Abstract Interpretation and Symbolic Model Checking, in particular focusing on the methods used to create approximations. To this end, this thesis introduces the ideas of Information Theory and Lossy Compression. Information Theory gives a structured framework which allows quantifying or valuing information. In other domains, Lossy Compression utilises this framework to achieve reasonably accurate approximations. However, unlike Abstract Interpretation or Symbolic Model Checking, lossy compression provides ideas on how one can find information to remove with minimal consequences. Having introduced lossy compression applications, this thesis introduces a generic approach to applying lossy compression to problems encountered in Worst Case Execution Time estimation.

To test that the generic approach works, two distinct problems in Worst Case Execution Time estimation are considered. The first of these is providing a Must/May analysis for the PLRU cache; whilst common in usage, the logical complexity of a PLRU cache renders it difficult to analyse. The second problem is that of loop bound analysis, with a particular focus on removing the need for information supplied by annotations, due to the inherent unverifiability of annotations.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to dedicate this thesis to my family and my friends, be they in York or elsewhere.

And also to my supervisor, Professor Alan Burns, for giving me a chance on the basis that I was half a mathematician.

Declaration

The research presented in this thesis is original research carried out by the author unless otherwise indicated in the text. The work was undertaken during the period of October 2009 and September 2014 , at the University of York, under the supervision of Alan Burns. Any external sources are acknowledged by bibliographic referencing. The work in this thesis has not been previously submitted for award at this or any other institution.

Chapter 1

Introduction

1.1 General Background

Computer systems have become an important part of everyday life, and the pace of change has been staggering. Within the space of 40 years computer systems have gone from being a product only used in business to being embedded in a vast number of consumer electronic devices. Without computers, many modern day conveniences, such as the mobile phone or fly-by-wire in aeroplanes would simply not be possible.

However, the pace of change has caused problems in itself: many of the innovations created are hugely complex and difficult to verify that the system will always behave as expected. This manifests in the presence of bugs; a large amount of resources are dedicated to the testing of products to find and correct bugs before a new product is shipped. Even after a product has been shipped, bugs may still be found, necessitating some form of update.

The two innovations highlighted above, mobile phones and fly-by-wire systems, also require that temporal properties hold. Mobile phones must accurately synchronise with a base station in order to work, and a fly-by-wire system must monitor external conditions to ensure that the pilots instructions are carried out correctly. This gives rise to the concept of Real-time Systems, an area of research identified by Liu and Layland [70] in 1973.

Real-time systems are specifically defined as systems which are concerned with real-time (as measured by clocks, in seconds) rather than the normal definition of computer time (measured in processor cycles). Normally, the relation to real-time is expressed as a set of *deadlines* on the computational *tasks* that the system must execute, with consequences for failure to meet said deadlines. The specific consequence of failure determines a further categorisation. *Soft real-time systems* are capable of continuing to operate should a deadline be missed, although they will fail should deadline misses be too frequent. Further, a user may notice a degraded service when deadlines are missed. A typical example of a soft real-time system is a video playback application; in the event a video frame cannot be decoded in time, the user will notice the dropped frame (and normally, decoding problems until the next key frame), but provided that subsequent deadlines are not missed, normal playback will resume. This contrasts with *hard real-time systems*, for which no deadline can be missed[1]. An example of a hard real-time system is that of a mobile phones baseband radio, which has to synchronise with the base station. If a deadline is missed, the baseband radio will be unable to communicate with the base station resulting in phone calls etc. being dropped. Further, until the baseband radio re-establishes synchronisation with the base station, the phone will be unable to provide communication.

Two major areas of real-time Systems research are *Scheduling* and *Worst Case Execution Time (WCET) Estimation*. Scheduling deals with the allocation of resources to tasks, whereas WCET estimation attempts to place an accurate bound on the time a task needs to execute. If these problems can be solved sufficiently well for a given system, then its temporal properties can be verified.

Unfortunately, the problems of Scheduling and WCET Estimation are hard, as modern computer systems are incredibly complicated. For example, in a regular system, simultaneous processes compete for system resources via a scheduler [95] - with each competitor adding to the complexity of the system. Whilst processor design has taken this use case into account, with features such as multicore and large caches, these features are designed to enhance the average performance of a processor. However, as a result of the fact that "being too late" is always incorrect in a real-time system, one must take into account the Worst Case behaviour. As seen in Figure 1.1, these average performance boosting features typically result in a worst case that resides at the end of a long tail.

[1]In theory, no deadlines can be missed for hard real-time systems. In practice, there will likely be a certain amount of overprovisioning and acceptable quality of service which allow for some deadline to be missed. However, this should not be assumed in the analysis of hard real-time systems.

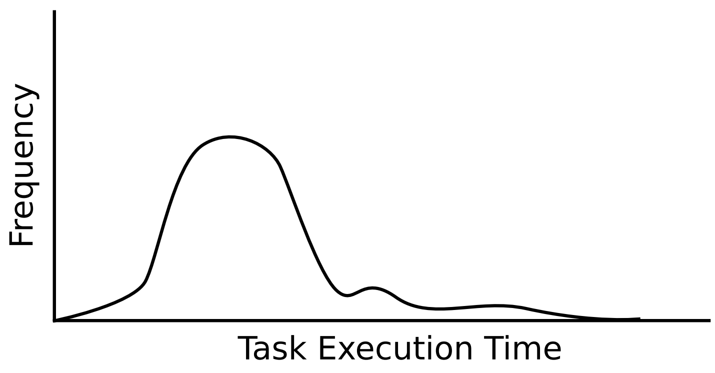


Figure 1.1: A typical graph of execution times of a task

The problem is compounded by the fact that many real-time systems are also *embedded systems*, or at least share the property of embedded systems that *resources are not abundant relative to the amount of work.* This rules out the simplest solution: more powerful hardware. Whilst using more powerful hardware would of course result in a reduction of worst case performance, providing more computing power increases resource costs in other areas. Perhaps the most critical of these is increased power consumption: with ever more systems in the mobile world, power usage is becoming a limiting factor - either due to heat dissipation or battery life. A second important reason is monetary cost: more powerful hardware will inevitably cost more - which manufacturers will be keen to avoid. Finally, the benefit is potentially very small: given the long tail of typical execution time distributions, more powerful hardware will often sit idle. If hardware sits idle, then the extra financial cost and power allocated to obtaining and using the more powerful hardware is wasted.

Hence a better method is to work around the complexity of computer systems, ensuring that computing resources are well utilised. This can be accomplished by *modelling.* In this context, modelling describes the process of taking observations of a system to construct an abstract representation of the system, the model, which can then be used to determine the properties of the system under given situations. This, of course, leaves a lot to be determined, such as how to collect observations, and how accurate the model needs to be in order to be useful.

Modelling in real-time systems is used frequently: continuing the example of scheduling, processor scheduling algorithms assume that all tasks have a well defined worst case execution time. Hence schedulers are not concerned with the content of the program, provided that it *can never exceed its worst case*. However, as the next section illustrates, modelling is in itself a hugely complex field.

1.2 Philosophy of Modelling

The first attempt to encapsulate the philosophy of modelling was made by Levins in 1966 [65], who argued that any useful model must make sacrifices in one of the three areas of *precision*, *realism* or *generality.* Whilst Levins caused much debate at the time of his work [78, 66], Levins philosophy is frequently cited as the basis for the inevitable tradeoffs involved in modelling.

Levins argument is that no useful model can exist that maximises the properties of *precision* (degree of accuracy of results), *realism* (faithfully replicating the phenomena being modelled) or *generality* (the applicability of the model to multiple situations). The key factor in this argument is the term “useful model”: as expanded upon by Odenbaugh in 2006 [77], a useful model must necessarily be *tractable*: that the use of the model for its intended purpose can be accomplished in the amount of time the user is prepared to invest. This in turn results in a fourth component in the argument: the amount of computation resources needed to use the model, or its tractability. This results in any model occupying a point in the diagram in Figure 1.2 - the more sacrifices one is prepared to make, the less computational resources are required.

As in any other case, models in real-time systems must be tractable, and hence sacrifices must be made. Depending on the type of model used, one or more of Levins desirable properties can be sacrificed - indeed, computer systems are often so complicated that it is necessary to sacrifice multiple properties to obtain a tractable model.

- *Generality*: Generality can be sacrificed by imposing *restrictions.* For example, a common restriction used in scheduling is *non-preemption* [9]. Assuming that once a task has been granted resources it cannot lose those resources until it has finished significantly reduces the effects of competition by restricting when competition can occur to before a

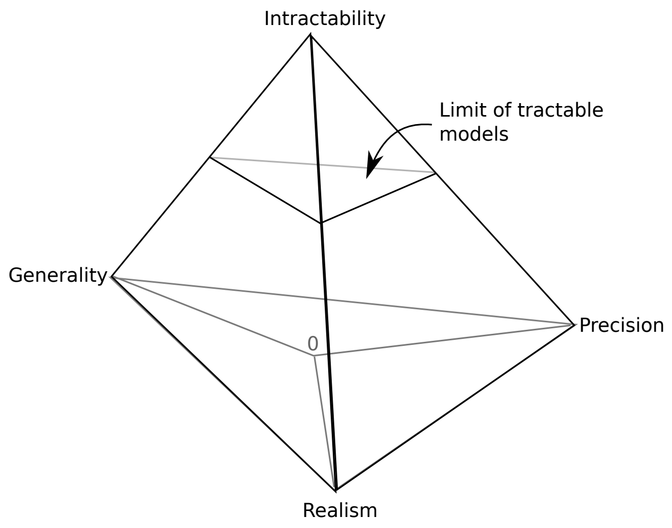


Figure 1.2: The extended form of Levins categorisation of models

task starts to execute. However, the enforcement of restrictions may result in solutions being missed. This can be seen as non-preemptive scheduling is considered to be non-optimal.

- *Realism*: Realism is commonly sacrificed, as real-time systems analysis is only concerned with behaviour and not the technical implementation. An extreme form of sacrificing realism can be seen in Statistical Analysis [42], which replaces the entire computer system with a simple statistical model.

- *Precision*: Precision is often sacrificed by allowing states which cannot be reached to be considered for the sake of a compact representation. For example, abstract interpretation [36] used in worst case execution time estimation [47] simplifies states in this manner, causing the result to be less precise than it could otherwise be.

Current state of the art techniques in modelling, such as Abstract Interpretation [36], use a combination of arguments to justify the specific sacrifices made for the sake of tractability; in real-time systems, these arguments typically include the safety of any simplification (as in Heckmann's bound on Pseudo Least Recently Used (PLRU) caches [56]), restrictions which also carry computational benefits for run time (as with online vs offline scheduling [40]), and ideally, simplification whilst maintaining optimality (as with Optimal Priority Assignment [5]).

However, all of the arguments for simplification rely on an innovative step, for which there is very little guidance. For example, Abstract Interpretation relies on identifying a property of the system which can be approximated [36]. This leads to the situation where research is primarily focused on identifying such properties. Further, even when such a property is found, there is no guarantee that the property yields a useful approximation. A more fruitful approach would involve rules which could give guidance, such that the innovative step can be designed to have the desired properties.

1.3 Information Theory and Lossy Compression

Information Theory [92] is the family of mathematics dealing with the transmission and storage of Information. A subset of Information Theory is Lossy Compression, which is used to create highly compact representations of information by discarding information that is not useful. Lossy Compression is used commonly, and to great effect, in audio and video compression. For example, the audio codecs MP3 [18] and AAC [16] or video such as h264 [110] all use lossy compression. Specifically, these codecs choose to discard information that humans would find difficult to perceive, and hence preserve the more useful information which lies in the perceptible range of humans.

Lossy Compression also exhibits a property where sacrifices are made in terms of tractability, to control the amount of data which is stored. A clear example of this can be seen in audio codecs; whilst codecs such as MP3 and AAC are specialised to sounds that humans hear, other codecs can go further. This can be seen in the SPEEX codec [103], which is specialised to compressing the human voice. Whilst it is unsuitable for general purpose audio, for the specialised domain of human voice compression it offers much higher compression than other approaches. This demonstrates that sacrificing *generality* can lead to less information being required for an acceptable result.

In principle, as the choices made in Lossy Compression are made for similar reasons, there is an argument that the approach used to devise a lossy compression method such as MP3 is also applicable to the problems of modelling encountered in real-time systems, and especially Worst Case

Execution Time Estimation, which uses multiple models to represent various components. This leads to the central hypothesis of this thesis.

1.4 Hypothesis

Problems encountered in finding appropriate models used in real-time systems problems, such as Worst Case Execution Time (WCET) Estimation can be thought of as a highly specific form of lossy compression. Hence, the approach used to design lossy compression algorithms can also be used to design an appropriate and effective model for use in WCET estimation.

1.4.1 Thesis Aims

To give evidence for the hypothesis, this thesis will aim to accomplish the following:

1. Introduce Information Theory and Lossy Compression in the context of Abstract Interpretation for WCET estimation.

2. Devise a general approach which applies lossy compression to devise a suitable model of a system.

3. Apply this approach to a problem for which there is presently no satisfactory solution: a complete PLRU cache analysis.

4. Apply the same techniques to the unrelated problem of Loop Bound Analysis, in order to reduce the amount of additional information required.

5. Evaluate how the approach was used in both problems to demonstrate the broad applicability of this technique.

1.5 Thesis Structure

Chapter 2 introduces current literature. First, real-time systems are introduced in general, followed by an in-depth review of techniques used in Worst Case Execution Time estimation. The specific problems of cache analysis and loop bound analysis are examined in detail, giving an overview of current state-of-the-art approaches.

Information Theory and Lossy Compression are introduced in Chapter 3. These approaches are examined with detailed examples of lossless and lossy compression algorithms. The cache analysis and loop bound analysis techniques from Chapter 2 are revisited, and examined in the context of lossy compression. Finally, the approaches are drawn together and summarised as a generic approach for devising a lossy compression method.

The generic lossy compression approach is given its first application in Chapter 4 on PLRU caches. The unique structure of a PLRU cache results in relatively high performance for a low cost but is difficult to analyse. The use of a more principled approach, as advocated in Chapter 3, allows the cache to be dissected and examined in greater detail to determine which parts are amenable to compression. Further, this approach is extended to the more general case of Hierarchical Not Most Recently Used (HNMRU) caches.

Chapter 5 applies the same generic approach to the unrelated problem of Loop Bound Analysis. In this case, the goal is to reduce the amount of additional information required by the analysis in the form of user annotations, by using lossy compression to guarantee the preservation of important information.

Finally, Chapter 6 presents conclusions by drawing together the approaches and presents an argument for the broad applicability of lossy compression to other problems encountered in WCET estimation. Limitations of the approach are discussed, along with further research ideas which would address these problems. Finally, the thesis aims and hypothesis are revisited to present concluding remarks.

Chapter 2

Current Techniques

This Chapter outlines current techniques and approaches to the problems concerning this thesis. Section 2.1 gives a brief overview of the field of real-time systems. This is continued in Section 2.2 which gives more detail to the problem of Worse Case Execution Time, and the two main approaches to this problem. The specific problems of Caches and Loop bound analysis, the main problem areas examined in this thesis, are detailed in Sections 2.3 and 2.4 respectively.

2.1 Overview of Real-time Systems

Real-time systems are computer systems which have to adhere to strict temporal deadlines. Whereas most systems can be measured with CPU or Logical time, the deadlines of real-time systems are defined in terms of the actual seconds that elapse between a job starting and producing output. There are two main types of real-time system, hard real-time and soft real-time [24]. Hard real-time systems are characterised by complete failure when a deadline is missed; these can include aircraft control systems or mobile phone baseband applications (controlling the synchronisation between a mobile phone, cellular tower, and other mobile phones). In either case, if correct temporal behaviour is not observed the system fails. In this scope, the consequences of failure are not considered (in the examples given, they range from potential fatalities to dropped phone calls), but instead delegated to discussion of safety critical systems. In soft real-time systems this property is relaxed, and the consequences for missing a deadline, while undesirable, are not serious and do not cause the system to fail. Examples of soft real-time systems would include video decoding, where a dropped frame degrades the experience of the user but causes no major problems. However, if many frames are dropped then the system becomes unusable.

Further to the definitions of hard and soft real-time Systems, a third classification is starting to gain traction, that of Mixed Criticality Systems (MCS) [23]. Current interest in MCS started as a result of the 2007 paper by Vestal [105] who wished to provide sharing of processor resources between tasks while giving greater guarantees of execution to more important tasks. For example, an unmanned aerial drone may have two tasks: flight control and data capture. While both tasks are functionally important, flight control can be said to have a higher criticality than data capture because if flight control fails the drone may be destroyed. MCS gives a method to encapsulate this type of requirement, allowing resources to be shared between tasks of different criticality levels while giving guarantees that tasks of high criticality will not be affected by tasks of lower criticality.

The study of real-time systems has two major aspects: Scheduling and Worst Case Execution Time (WCET) analysis. Scheduling [24] takes information on a number of tasks and attempts to produce a strategy such that all tasks will have the required resources when they execute. This can be accomplished either *offline scheduling*, where a set of parameters is calculated before the system is deployed; these parameters are then used to determine exactly when each task executes. A simple example of offline scheduling is the cyclic executive scheduling policy [8], where each task is allocated a fixed amount of time within a scheduling table. When the system is run, the cyclic executive simply looks up which task should be running at any given time in the scheduling table; when the end of the table is reached, the cyclic executive simply returns to the beginning of the table and starts again.

Alternatively, *online scheduling* utilises set of rules that on deployment of the system determine which tasks can execute. An example of online scheduling is the Fixed Priority scheme [24], where a fixed priority is assigned to each task. When the system is run, the set of tasks that are currently available for dispatch is evaluated and the task with highest priority is executed. In order to assign priorities, an approach such as Audsley's Optimal Priority Assignment (OPA) algorithm [5] can be used. For single processor systems, if there exists a priority assignment resulting in a schedulable system, OPA is capable of finding it by providing a set of priority assignments that hold for the worst set of task dispatches, the critical instant. While fixed priority scheduling is well studied on single processor systems, this is not the case on multiprocessor systems, where problems such as task migration costs and the lack of an easily identifiable critical instant[1]. Further research by various authors, and surveyed by Davis and Burns [40], has extended the fixed priority approach to multi-processor systems (e.g. via Partitioned Fixed Priority Scheduling), but there is currently no optimal priority assignment algorithm for multiprocessor systems.

The Worst Case Execution Time (WCET) of a task is defined as the maximum amount of time which that task may require to execute [24]. Worst Case Execution Time analysis is the problem of placing a bound on the execution time of a task. Literature uses the term *tightness* [24, 88, 31] to describe how accurate this upper bound is; a WCET estimate becomes tighter as it approaches the real WCET. A further term used by the literature is *safety* [24, 13, 81], which is used to indicate that the WCET bound is indeed an upper bound. However, due to confusion with the frequently overlapping field of safety critical systems, this thesis will use the alternative term *soundness*.

WCET bounds are not strictly necessary for scheduling, although scheduling typically requires a notion of maximum execution time for schedulability analysis to be possible. Similarly, while WCET analysis can be conducted without regard for scheduling, in certain circumstances knowledge of scheduling is desirable. For instance, if a scheduler causes a task to be preempted, then the execution time of that task will increase as shared resources, such as processor cache, are disrupted [4]. In addition, while the time allocated to the interfering task does not count towards the execution time of the disrupted task, it still causes the task to take longer to respond. Generally, the problems of inter-task interference, on both shared resources and the delay of preemption, are separated into the related problem of Worst Case Response Time Analysis [24]: determining how long a job will take from its task becoming available to its termination.

[1]Unlike in single processor systems. where the critical instant can be stated to be when all tasks are dispatched simultaneously, in multiprocessor systems the critical instant is not easily computable.

2.2 Worst Case Execution Time Analysis

This section gives an overview of current approaches to the WCET problem. In particular, thought is given to the idea of WCET as a testable property in Subsection 2.2.1. With this in mind, the two main approaches to WCET estimation, along with their respective advantages and disadvantages, are introduced in Subsections 2.2.2 and 2.2.3.

If one ignores that solving the WCET problem also solves the halting problem, the main difficulty with WCET analysis is that the many optimisations created for computers systems, are not designed with WCET analysis in mind. While each of these optimisations, such as caches or compiler optimisations, are individually simple, the problem of their combination is illustrated by Braitenberg in his book, Vehicles [15]. The vehicles which Braitenberg constructs are automata which possess sensors and some amount of logic. Braitenberg's goal is to create a toy world to learn more of psychology. With only simple logic it becomes apparent that his vehicles are capable of exhibiting complex behaviour. In particular, Braitenberg states that the only realistic way of finding out what logic the vehicle uses would be to open the vehicle and see directly; it is insufficient to use external measurements to analyse the vehicle. Conversely, designing the vehicle is quite easy, as it is merely adding or removing components. Further, design can be carried out by very simple rules, such as Darwinian selection. Braitenberg calls this the law of "uphill analysis and downhill design", to reflect the fact that analysing a system is much harder than designing one.

Bullock and Silverman [21] combine the observations of Braitenberg [15] with Levins [65] to come up with a simple observation: While a system may be designed to be easy to analyse, the effort for this is expended in the design phase rather than the analysis phase. Hence there is a tradeoff between effort required to design, and effort required to analyse. Unfortunately, for WCET analysis, this only leads to the observation that as the design of optimisations for computer systems is for better average case performance, effort has not been expended on the analysability of the optimisations, and hence analysis of such optimisations is problematic.

There are exceptions to this observation, however. Examples of this include the JOP Java machine [89] which has a particularly notably feature in its "method cache". The method cache is a cache which loads entire Java methods into cache memory. Hence, instead of every instruction fetch being a potential cache miss, instruction cache misses can only occur when jumping into or out of a method, reducing the effect of state explosion in static analysis. In software development, the programming language wcetC [61] is a C based language which drops features that hinder WCET calculation and compels programmers to provide additional information. This forces additional effort to be expended during program design rather than analysis. However, these are exceptions rather than the norm, where difficult to analyse features are common.

2.2.1 WCET as a Testable Property

The usage case of WCET is to provide some guarantee of timing behaviour in a working system. This guarantee is similar in nature to any other guarantee on the correctness of a program function, and hence it is appropriate to treat the WCET as a testing problem. In "The art of software testing" [7], testing is defined as "the process of executing a program with the intent of finding errors." This definition is almost adequate to directly translate to the WCET problem apart from the use of the phrase "executing a program". The problem is that in many cases the system that the program is running on is not fully understood with regards to its precise timing behaviour. Hence testing must also explore the behaviour of the system that the program is running on in addition to the program itself.

The purpose of viewing WCET as a testing problem is two fold. The first reason is that in even the most stringent safety requirements, such as avionics certification [87], testing of the system does not need to prove that the system is entirely free of faults. Rather, it needs to provide evidence about the likelihood of faults occurring in a given time period. Present approaches to WCET, as will be seen in further sections, tend to be used to find absolute bounds even though an absolute bound is not necessary to meet the requirements of the end user.

The next reason to view WCET as testing is to attempt to integrate it within the general testing regime, alongside the property of functional correctness. This makes sense given that in a real-time system the execution time is a property which can cause the system to fail. Further, existing practices mandated within required testing procedures may be measured in additional ways to gain evidence about the WCET of a program. A prime example of this would be modified condition/decision coverage [7], which is required by the DO-178B / ED-12B standard [87]. Modified condition/decision coverage requires that for each variable in a conditional expression, the program must be tested such that changing that variable determines the outcome of the conditional expression. If these tests were conducted on the target hardware, then the tests could be measured to provide a large body of data on execution times, and there is no reason why this data could not be used to aid WCET estimation. This data would not be capable of replacing a full WCET analysis, as MC/DC (and functional testing in general) is not guaranteed to stress the worst case path. However, as all components of the worst case path will be tested, provided that sufficient detail is captured on the execution times of the portions of code tested, data from functional testing could be used as the basis of a measurement based technique which combines execution times from blocks of code. Ideally, such testing would take place on the target hardware, which would incur additional cost/effort during testing. However, assuming that the test hardware is a reasonable approximation of the target hardware, one could infer that it is likely that larger execution times on test hardware correspond to similarly increased execution times on the target hardware.

Returning to the argument that WCET should be considered a testing problem, a notable observation is that testing is split into two main categories, called *white box* and *black box* testing [7]. *White box* testing refers to testing methodologies which attempt to understand the system by dissecting the system and trying to identify areas of failure. For WCET, Static Analysis implements a white box approach to finding the WCET by constructing a model of the system under analysis. *Black box*, or *requirement led*, testing is the practice of determining if the system is functioning correctly by taking measurements and comparing them to the expected result. In WCET, this is implemented by measurement based testing, where the model used is constructed solely from measurements taken from the system.

An early review of testing real-time systems [50] identified that real-time software testing was of a significantly lower quality than conventional software testing. However the main reason was due to industrial practice of real-time systems developers, as they adopted programming techniques which were not conducive to testing for performance reasons. This has since changed, due to the increase in computation power and that safety critical

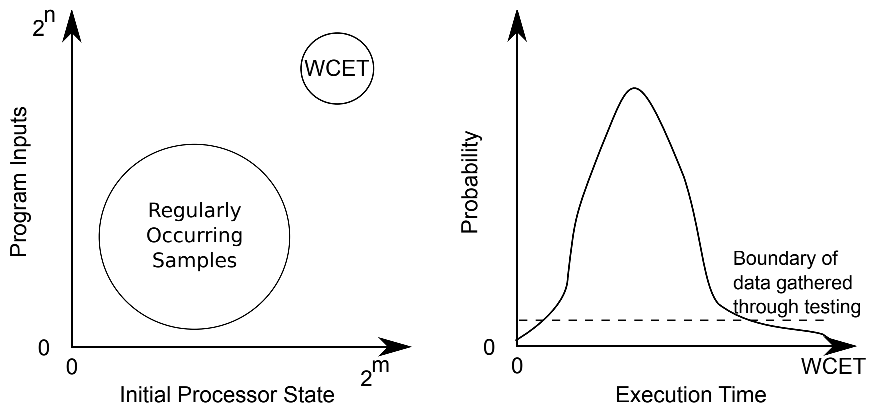


Figure 2.1: The problem facing measured WCET analysis

certifications which many real-time programs have to obtain (e.g. [87]). However, testing real-time systems is not without issue. An article by Butler and Finelli [25] on testing real-time systems argues that it is impossible for normal black-box testing methods to certify real-time software to an ultra-reliable level. This is simply due to the amount of resources such a certification would need in order to detect faults which occur at a rate of less than 10^{-7} faults per hour. While this might seem to mean that viewing the WCET problem as a testing problem is not a helpful idea, Butler and Finelli do not consider the WCET problem directly, and also do not consider more testing methodologies beyond observing for the presence of faults in simple black box testing.

2.2.2 Black Box Testing: Measurement Based Analysis

Measurement based analysis is the family of techniques which construct a model of the system by using measurements obtained during some form of testing. The most primitive form of measurement based analysis is simply to take the longest observed time from testing with a variety of inputs and increase it by a “fudge factor”. This method lacks realism, as it doesn’t seek to model the processes within the system, and precision as error bounds are not even defined. In general, this lack of realism is prevalent to a more or lesser degree depending on how much the internal workings of the system are analysed. However, as external measurements can be taken from any system, measurement based analysis techniques are highly general: a measurement based technique developed for one type of system is likely to be directly applicable to any type of system, even if the systems use differing components.

A common criticism of measurement based analysis approaches is that it is difficult to observe the WCET during testing, and hence the predictions made from measurement based analysis may be unsound [24, 112]. This is illustrated in Figure 2.1, which shows why this is the case; that the number of possible samples to test is exponential with respect to the inputs to the system, and hence will dwarf a practical number of test runs. In essence, the cause of this problem is similar to that of the state explosion problem faced by static analysis; that there is too much potential data to encounter. However just because the WCET is not observed it does not mean that the WCET can not be predicted via interpreting the data, although this may lower the confidence that such a result is correct.

Another issue encountered in measurement based analysis is simply obtaining the measurements. Hilary and Madsen [57] present an argument of how this can be more complicated than it at first seems. In Hilary and Madsens example, a program is tested by inserting additional code to take measurements, to measure the number of cache misses. The problem arises because the additional code is not isolated from the rest of the system, and will presumably have the potential to cause cache misses, which changes the number of cache misses in the program. Further, as the state of the cache is not known, it is not known how many additional cache misses are generated by executing the measuring code. Remedies proposed by the paper basically amount to sticking to non-intrusive measurement practices such as debugging hardware when possible. Hilary and Madsen cite this effect in other disciplines as either the measurement problem or the probe effect. One method not considered is the use of cache locking. By locking the additional code into cache, the code will have a constant effect over the lifetime of the program; such an effect is more manageable from an experimental point of view.

Multiple techniques for measurement based analysis exist. The TU Vienna Measurement research prototype, described in [112], builds upon measuring the end-to-end execution time for a system with a variety of inputs by introducing a strategy to select these inputs. The strategy used is to have a genetic algorithm attempt to maximise the execution time of the program by varying the program's inputs. Unfortunately the definition of "inputs" is not described, so it is unknown if this should be taken to mean inputs to the program under analysis or inputs to the analysis itself, which includes additional information such as the initial processor state. Further, the authors acknowledge that this approach cannot give a sound bound, as the genetic algorithm will approach the true WCET from below, and has no guarantee to find the true WCET.

Bernat et al. [12] present an argument on how to discover probabilistic dependencies between measured factors. This is accomplished by defining execution time profiles on segments of the program which specify the probabilities of executing within a certain time, as determined by measured experiments. The rules for combining such segments depend on if the segments are statistically independent of each other. Independent segments can be combined by simple convolution. Dependent segments require a new structure termed a joint execution profile which captures the nature of any dependencies, by using measurements of executing both segments together. Given that any blocks may be dependent, Bernat et al. detail statistical tests for dependence based on the results of joint execution profiles. Further they acknowledge that in some instances it may be impossible to prove independence of segments, but not possible to prove the nature of any dependence. The proposed remedy for this is a method for finding the most pessimistic joint execution profile for the two segments, which guarantees that the probabilistic WCET is sound, but possibly pessimistic. This approach is implemented in the pWCET tool [13].

Bernat et al. [11] later refined the idea of probabilistic dependencies by introducing the notion of Copulas [76]. Copulas are the general statistical tool used for describing the nature of dependencies between variables. By transforming the original work into this form is possible to utilise other techniques from statistics, in particular the methods used to create upper and lower bounds for copulas. This enables less pessimistic, but still sound, bounds on the dependence structure of observed variables to be found than in the previous work [12].

An alternative use of probabilistic techniques is the use of Extreme Value Theory (EVT) statistics [63] by Edgar [42]. Edgar's work utilises EVT to predict the probability of exceeding a deadline with a high degree of confidence, and a method to integrate this approach into scheduling. However, one criticism of the use of EVT is presented by Griffin and Burns [51], as the use of EVT makes the independent and identically distributed assumption (i.i.d.) on input data. Given that the majority of real-time systems produce side-effects, the i.i.d. assumption cannot be guaranteed to hold.

More recently, the field of Probabilistic Real Time Systems has enabled a revisiting of Extreme Value Theory. Given hardware with inherently random behaviour, such as the random replacement cache [38], EVT can be argued to be an appropriate tool to determine the tail of distributions. In particular, Cucu et al. [38] have proposed additional tests and assumptions which they argue can be used to ensure that the i.i.d. assumption holds, and therefore that the results of EVT are valid.

2.2.3 White Box Testing: Static Analysis

Early formal approaches to the WCET problem favoured static analysis, which is any WCET estimation technique which uses a model based on program source or assembly code and the behaviour of the hardware it is to run on. For example, take the 1989 work by Shaw [94] who proposed a general approach which uses an extended Hoare logic to calculate a program's WCET. Previous work had focused on "new" languages, which were generally extended forms of a subset of C, for instance MARS-C [82] which was released earlier in 1989. Puschner and Koza created MARS-C due to identifying the issue that in general it is impossible to bound a program's execution time, for this would solve the halting problem. MARS-C is a restricted form of C in that it does not permit constructs which are difficult to analyse like recursion. This is not enough to give tight WCET bounds, so Puschner and Koza introduced mandatory annotations to the language, which give additional information such as loop bounds or experimentally derived execution times for sections of code. While the latter of these is no longer used within static analysis, modern tools make heavy use of code annotations in order to achieve tight WCET bounds [100]. Further, while attention has shifted away from languages designed for WCET computation, such languages do still exist, with the most current example being wcetC [61].

As static analysis relies on breaking down the program structure, it is first necessary to define a notion of program structure. A fairly universal way of doing so is by defining the *control flow graph* (CFG) [3] of a program.

```
If x > 0:
  call function F
If x < 0:
  call function F
```

Figure 2.2: An example of pessimism in structure analysis

A CFG represents a program in terms of two structures:

- *Control instructions*: Any instruction which causes conditional branching of the program.
- *Basic blocks*: The maximal sets of consecutive non-control instructions.

Using these definitions, a CFG is defined as a graph with the nodes corresponding to control instructions, while the edges connecting two nodes correspond to the basic block which executes inbetween the two linked control instructions. The CFG may be extended with other notions as in the program dependence graph [48], an augmented CFG for compiler optimisations which describes data flow.

The first technique used in finding the WCET of a program by its CFG is structure analysis, as initially outlined by Shaw in 1989 [94]. The method outlined by Shaw is that the time a program takes to execute can be derived from the source code. The execution time of statements, or basic blocks, can be bounded by measured experiments, loops take time equal to the contents of the loop multiplied by the number of iterations and so on. Shaw proposed an extended Hoare logic to formalise this idea, which enabled automatic calculation of execution time for programs with known loop bounds. A failing of the method, which Shaw acknowledges, is that due to the effects of pipelining, caches and other hardware it is difficult to get tight bounds on the program. Shaw's work was refined by Lim et al. in 1995 [68] to try and address this. Lim et al. identified that there is a period of overlap between basic blocks; by calculating the effects of these periods of overlap, and how the basic blocks contest for resources, it is possible to achieve a tighter WCET bound.

There are several problems with the structural analysis approach. For instance, consider the program in Figure 2.2. Clearly, it is impossible for $x <$

0 and $x > 0$ to be true, and so F should be called at most once. However, as the `IF` statements are separate structural elements this information cannot be exposed to them, and so structure analysis calculates that the WCET corresponds to two executions of F, which cannot happen. Another issue is that structure information is most easily extractable from program source code, whereas safety regulations may stipulate the analysis must be done on machine code [87]. Given that compiler optimisation can have significant effects on the structure of a program [48] this introduces the problem of extracting the program structure of the machine code from the program source code. Attempts have been made to preserve structural information from the source code into the machine code [62, 45].

Given the flaws of structure based analysis, the family of path enumeration techniques was developed. Path enumeration techniques attempt to search through the feasible paths through the program's CFG. The two methods of path enumeration are explicit path enumeration techniques (EPET) and implicit path enumeration techniques (IPET). The only difference between the two is that EPET explicitly identifies each feasible path through the program before searching over them, whereas IPET implicitly identifies feasible paths while searching through them. The advantage of EPET over IPET is that more information is available; the disadvantage is much the same, in that more information is processed, requiring more time and memory resources. Both families of techniques can be proven to give tight WCET bounds, provided all constraints are known [83].

The primary method of performing IPET is to use integer linear programming (ILP). Li and Malik [67] developed this approach to combat, what they saw, as the wastefulness of explicit path enumeration techniques, and created the *Cinderella* tool from their research. Generally, linear programming (LP) is the mathematical problem of finding the best outcome given a set of linear constraints. Integer linear programming is the same as linear programming with the additional restriction that solutions must be integers, although this additional restriction makes the problem NP-hard. Li and Maliks approach is to derive a set of linear constraints on the execution time from a program's CFG. These constraints are then simply given to an ILP solver, which aims to maximise the execution time. Li and Malik do acknowledge that ILP is an NP-hard problem, but they present an argument that by analogy to previous work (Information Description Language by Park [79]), there is a guarantee that any ILP problem derived from constraints on a CFG corresponds to a LP problem, and hence a solution can be found in polynomial time.

An alternative IPET technique is presented by Marref [71]. Marref's proposal is named *predicated WCET analysis*, and is accomplished by analysing each basic block of code separately, and to express when that block of code can execute in terms of logical constraints. When a block of code executes, the present state is modified according to logical rules and the next block of code picked. As logical rules do not necessarily have an obvious representation in ILP, Marref proposed the use of constraint logic problem (CLP) solving to find the WCET when analysis is performed with this method. As CLP is the name given to any system combining constraint solving with logic programming, CLP is a more natural choice than ILP. The major issue in using CLP is that if the search order is not picked carefully, the complexity of the problem is exponential; Marref does propose a search strategy which avoids this issue. Marref also describes further performance enhancements from using a divide and conquer approach which decreases precision and realism in the model for additional tractability. This is achieved by removing the history of the search and recommencing the search at the next level with the cut off history, and making pessimistic assumptions should this history be required. While Marrefs system is an IPET system, it shares a lot with structural analysis based approaches in that the structural blocks are analysed separately and this data combined. The major difference is that the language used to describe the relationships between blocks in predicated WCET analysis is much richer than that used in structural analysis.

The main problem with IPET based analysis is state explosion [104]. While IPET is sufficient for simple systems, modern computer architectures feature many components which are not easily modellable as constraints due to lack of information. For example, the state of a cache or branch prediction buffer do not easily map into the constraints used by IPET, as the initial states of these components may not be known. Attempts to force such a mapping cause an IPET to have to consider an intractable number of potential paths through the program, to account for all possible states that these components may have. As such it is necessary to adopt additional techniques to counter the effects of state explosion.

2.2.4 Abstract Interpretation and Symbolic Model Checking

To combat state explosion, the two main techniques used in current approaches are Abstract Interpretation and Symbolic Model Checking. These techniques have the same goal of discarding irrelevant information to the problem at hand, but accomplish this goal using different and distinct methods.

Abstract interpretation is a general program testing technique developed by Cousot and Cousot [36]. The goal of abstract interpretation is to perform the minimal amount of work in order to prove correctness of a property for a program. The method for this is to create an abstract representation of the program which contains less information than the original program, but retains enough to gain a valid answer. For instance, the operation $-x \times y = -xy$, on positive x and y could be abstracted to $- \times + = -$, if the only required information was the sign of the result. Then, once the abstract representation has been created, *fixed points* can be identified or created. Fixed points are points within the abstract representation which are identical, or can be merged by a merging operator without impacting the validity of the model. By combining fixed points, the size of the model is reduced and the property is easier to prove. The level of approximation that the abstraction uses is not defined by Cousot and Cousot; instead this is left to the implementation and specific usage to pick an appropriate level. Hence Cousot and Cousots argument that all program analysis techniques can be realised by abstract interpretation is at least partially correct, as any level of approximation could be used. However, specific details regarding the abstract problem space may be easier to prove using alternative methods.

In usage in WCET estimation, abstract interpretation is commonly used to model hardware effects and paired with ILP to perform path analysis, an approach pioneered by Theiling and Ferdinand [99] in 1998. By determining the abstract states which the different parts of the program execute in, tighter bounds can be found on the execution time of the basic blocks of the program. This information is then used in generating the ILP problem which results in a tighter bound on the WCET of the program. Commercially, this approach is implemented in the aiT tool [46].

Model checking is the general technique of automatically constructing a model of a system by some rules, and determining if a particular property holds in that model. The pioneering work on model checking computer systems was carried out by Clarke and Emerson, who proposed a system of temporal logic to prove the correctness of parallel programs [29]. Temporal logic is an extension to normal logical formulas to enable reasoning about the order of application of logic formulas, and as such can be easily used to determine the WCET of a program by selecting the maximum execution time of all paths through the program. The use of model checking in path analysis results in an explicit path enumeration technique, as it explicitly details all paths through the program.

While model checking is particularly prone to the state explosion problem, there are a number of countermeasures. An important countermeasure is the use of binary decision diagrams (BDD), a tool detailed by Bryant [20]. Bryant proposed a system of transforming graphs of boolean systems such that the graph only contains the minimum information necessary to represent the boolean outcomes accurately, with such graphs being BDDs. Building upon Bryant's work, McMillan et al. [22] proposed an extension using Mu-calculus to define model checking on BDDs, and from this deriving the extensions needed to use temporal logic, as parts of a new technique termed symbolic model checking. Symbolic model checking represents the space to model check symbolically rather than explicitly, and by determining properties on symbols many concrete states can be examined simultaneously. In essence, this idea is the same as the abstraction phase of abstract interpretation; represent the space to be searched more compactly but with minimal impact on precision.

In usage on the WCET problem, Wilhelm argues that model checking is not a practical method for WCET calculation [111], due to the exponential size of the problem model checking has to solve. He argues that the abstract states of abstract interpretation mean that information can be more compactly represented, and the tradeoff in accuracy for this is acceptable given the amount of computational work saved. A strong rebuttal of Wilhelm's arguments is presented by Metzner [73], where Metzner demonstrates that model checking can be used to improve WCET estimation. This is accomplished by using symbolic model checking [22], therefore gaining the benefits of using an abstract model as in abstract interpretation. However, as abstract interpretation uses ILP, which can suffer from numerical instabilities which result in picking an invalid path through the program, Metzner argues that the explicit paths used in model checking can give a more accurate

WCET estimate. Metzner does not consider the cause of numerical instabilities in LP problems, which is the algorithm used. Hence it is possible to argue that by using different algorithms to solve the ILP problem, numerical instabilities could be avoided.

In practical terms in the WCET problem, there is very little to distinguish between the use of abstract interpretation and model checking. Abstract interpretation explicitly involves an abstraction step. Practical model checking is accomplished via knowing when to use techniques such as symbolic model checking which performs an abstraction. In making the abstraction, abstract interpretation creates the final set of abstract states implicitly by combining existing abstract or concrete states through combination operators. Model checking performs this step explicitly by definitions in the abstract model. Both techniques then rely on some other automated process - traditionally integer linear programming [111] for abstract interpretation, and varying [73] for model checking - to prove properties. In essence, the biggest difference between the two is how the abstract space is calculated.

2.2.5 Summary

While traditionally viewed as a separate problem to functional correctness, viewing the WCET problem as such yields useful insights. For example, the similarities between measured/static analysis and black/white box testing highlight the respective flaws of each approach. For measurement based analysis, this means a lack of confidence due to unobserved behaviours. Static analysis is similar in that much like a program must be simple enough to be understood for white box testing, simplifications must be made for static analysis to be tractable.

Having examined the general techniques used in WCET analysis, it now follows to inspect applications of these techniques in WCET analysis. Two such applications, Caches and Loop Bound Analysis, are inspected in the following sections.

2.3 Caches

Caches are a highly common form of local memory found on almost all modern computers. While caches, or some other form of local memory, are necessary to achieve anywhere near a CPU's maximum performance, a cache presents a considerable complication to WCET analysis. Specifically, a cache has a corresponding state, and this state must be modelled to accurately predict WCET. An overview of the history of caches, and their impact on WCET is found in Subsection 2.3.1. Subsection 2.3.2 examines in detail the LRU cache replacement policy, which is completely understood in terms of its impact on WCET. This is in contrast to PLRU and HNMRU, detailed in Subsections 2.3.3 and 2.3.4, where the understanding of the effect of cache is not as complete. This is in spite of the fact that the PLRU policy is a very common cache replacement policy due to its performance and the cost of implementing it in hardware when compared to LRU.

2.3.1 History and the impact of Cache on WCET

In the first computers, memory and the CPU ran with perfect synchronisation. However, such early computers were not fast. Indeed, a common fact used about the speed of progression for modern computers is that the Apollo missions managed to get to the Moon in the 1960s with less computational power than a 1990s calculator. Part of this increase in speed was the realisation that a CPU could operate at frequencies much faster than the memory. This can be observed in current consumer machines DDR3 memory runs at frequencies between 100 and 233 MHz, whereas a consumer CPU will run at speeds typically between 1 and 3 GHz. In addition memory controllers and the physical distance between memory and CPU add overhead to communications between memory and CPU.

The disparity in speeds of memory and CPU comes at a cost: without some mitigation, the CPU becomes bound on the data that it can process. Indeed, some statistics indicates that 75% of a program's execution time can be occupied simply in retrieving data and instructions from main memory [55]. The mitigation most commonly used is a CPU *Cache.*

A Cache is a relatively small local store of memory, either on the CPU die or physically close to it, that stores a subset of memory for fast access. Caches may be dedicated to a specific function; commonly separate instruction and data caches are used [55], rather than a single cache for both instructions and data. This is because in many applications the amount of data and instructions consumed by the CPU is not balanced, and this separation prevents either from displacing the other. Caches may also be

	Cache Level			
Processor	L1	L2	L3	Memory
150 MHz MIPS r4000	13ns	–	–	253ns
1.5 GHz Samsung Exynos 5250	2.6ns	17ns	–	127 ns
2.8 GHz AMD Phenom II X4 920	1.0ns	4.3ns	20ns	92.5ns
3.4 GHz Intel i7-4770	1.1ns	4.7ns	20ns	56ns
3.55 GHz IBM Power 7	0.5ns	2.3ns	6.7ns	120ns

Table 2.1: Cache/Memory Latency for various processors. Data from documentation of 7-Zip LZMA Benchmarks [33].

multi-layered, with the inner cache levels being faster for the CPU to access, but also smaller than the outer levels.

When a CPU is equipped with a cache, all appropriate memory accesses are routed via the cache. The cache then uses some predefined rules to determine what memory locations it should store [2]. When a memory access is received by the cache, it checks to see if the memory location has been stored in the *cache lines* stored in the cache. A *Cache Line* represents the minimum size of an element in the cache; this is normally not the same size as the minimum amount of memory a CPU can request in a single instruction. If the memory location is in the cache it results in a *cache hit*; the value of the memory location can be returned to the CPU immediately, without any additional expense. Otherwise a *cache miss* is recorded. In this case the cache passes on the memory access to the next cache level, or main memory in the case of the outermost cache level. For each additional cache level traversed the time taken by the memory access increases, with the final hop to main memory being very expensive relative to the innermost cache level. Examples of cache/memory latency for various processors are given in Table 2.1, which shows that the penalty for a cache miss can be expected to be between 50-200 times that of a cache hit.

Caches that hold data must also have a policy to handle writing to the cache[2]. There are two main methods of performing this: *write-through* and *write-back* [60]. A *write-through* cache is the simpler policy: whenever a memory address stored in the cache is written to, it simply forwards the write request through the cache straight away, as well as updating the value

[2]Ignoring the possibility of polymorphic code, an instruction cache can be seen as read-only.

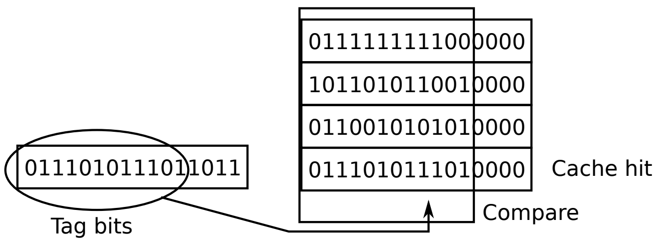


Figure 2.3: A fully associative cache; any cache line may contain any memory location

the cache holds. In essence, this ignores any caching behaviour for writes, and therefore can result in poor performance. A *write-back* cache avoids this by storing information on what data has been modified and only writing the data back to the main memory when necessary. This more complicated approach theoretically has improved performance, but is more expensive to implement in terms of the silicon necessary. The extra silicon may result in the cache running more slowly. Further, a write-back cache may need extra aid for peripherals which communicate through RAM, as such communications must be guaranteed to enter RAM for them to be effective.

The distinguishing feature of a cache is that the cache uses a predefined set of rules to determine what data it should store. This is in contrast to the main alternative, the scratchpad [108], which must be managed explicitly. Evidently, this means that the scratchpad requires additional work; either the programmer or the compiler must determine what should be stored within the scratchpad, whereas with a cache this is automatic. However it comes with a corresponding advantage: by explicitly declaring what is stored in the scratchpad, the contents of the scratchpad is known. Recent work by Whitham et al. [109] has shown that scratchpads, with appropriate resource allocation policies, are suitable for certain workloads with respect to analysable WCRT. Whitham et al. also make the point that with increased hardware support, the additional costs of scratchpads could be lowered making them more attractive. In contrast, caches have good hardware support, but the contents of a cache is not explicitly declared in advance, and by extension the effect on execution time cannot be calculated easily.

Cache architectures are highly variable. The simplest form of cache is a *fully associative* cache [55], illustrated in Figure 2.3. A *fully associative*

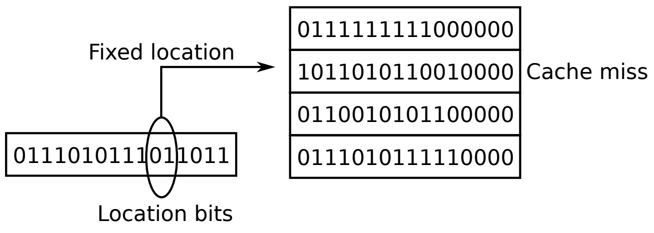


Figure 2.4: A direct mapped cache; memory locations mapped to specific cache lines

cache may store any memory address in any location of the cache. In the event that the cache is full, an eviction policy determines which element to remove. Fully associative caches work by computing a tag from the higher bits in the memory address, and on access uses this tag to determine if there are any matches. As every memory address within the cache has to be checked for a potential match, and this has to be done quickly, this normally means that fully associative caches are small. For speed, the tag checking is normally done in parallel, and this requires extra silicon for each additional line of the cache.

To combat the silicon costs of the fully associative cache, the *direct mapped* cache is an alternative [55], illustrated in Figure 2.4. The direct mapped cache specifies that each memory location may only reside in a single location in the cache. The location in the cache is typically given by taking the lowest n bits used in the address of the cache line (ignoring bits which are set to 0 as part of computing the cache line's address). As only a single location in the cache has to be checked, this results in much less silicon to be required than in a fully associative cache. However, the problem comes as a simple pathological case: if two memory locations which are frequently accessed reside in the same location of the cache, then the cache cannot store both simultaneously, and hence will not provide any performance benefit.

As a fully associative cache is too expensive for a large cache, and a direct mapped cache has an easy pathological case, the answer is a combination of the two: the *set associative* cache [55], illustrated in Figure 2.5. A Set associative cache behaves similarly to a direct mapped cache in that it uses the memory location requested to compute a location within the cache that the memory location can be stored in. However, unlike a direct mapped cache, the location computed by a set associative cache can store a set of

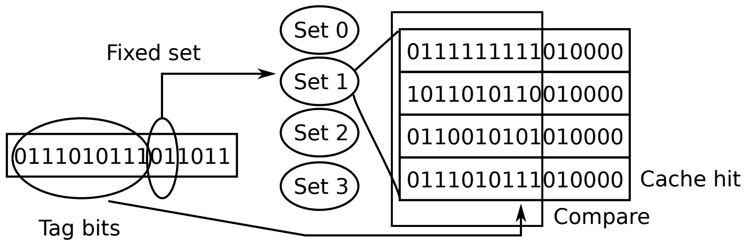


Figure 2.5: A set associative cache; cache lines mapped into specific set, whose elements can be any memory location

memory locations. The set is managed in much the same way as a fully associative cache: a tag determines if there are possible matches, and an eviction policy determines which memory address to remove from the set. The size of the set is referred to as the number of *ways* of the cache; a 4-way cache can hold 4 separate cache lines in each cache set.

The set associative cache avoids the pitfall of the direct mapped cache by allowing multiple cache lines which indicate they occupy the same location in the cache to exist in the cache simultaneously. This causes the size of the pathological case to require more memory accesses: whereas the direct mapped cache requires only 2 memory accesses for the pathological case, an n-way set associative cache requires at least $n + 1$ memory locations, as n locations can, by definition, coexist simultaneously. Further, as the number of ways of the cache is relatively small, the set associative cache also avoids the pitfall of the fully associative cache, as the amount of hardware required to check for the presence of a memory location in a cache set is relatively small. As such, the only missing component of the cache to explain is the eviction policy.

Eviction policies, as with other areas of the cache, can vary greatly. Three policies will be examined in detail in later sections; these are Least Recently Used (LRU, Section 2.3.2), Pseudo Least Recently Used (PLRU, Section 2.3.3) and Hierarchical Not Most Recently Used (HNMRU, Section 2.3.4). However many other policies exist. An intuitive eviction policy is the Least Frequently Used (LFU) policy [55], where the element of the cache least frequently used is discarded. The intuition behind this policy is that if a cache element is frequently used it is more likely to be used again. However, there are other less intuitive policies, such as Most Recently Used (MRU)

[55], which evicts the element last accessed. While unintuitive, in some use cases MRU will exhibit good performance; for example when streaming a large volume of data which is accessed precisely once, MRU will preserve other cache lines. Similarly unintuitive is the random replacement policy [38], which chooses the element to evict randomly. While one might consider a concrete strategy based on usage data to be more logical, the random strategy has a quantifiable probability of exhibiting poor performance. This contrasts with concrete strategies which typically have pathological cases, and if such cases cannot be ruled out by analysis, the presence of a pathological case can cause analysable performance to be low.

Traditionally there are two main aspects to consider when deciding on a cache policy. The first is performance; a cache is designed to improve performance, and so a cache policy that performs well is desirable. If the jobs to run on the processor are known far enough in advance it may be possible to pick a cache policy that works well with the given jobs. The second is the cost of implementation. A complicated cache policy may consume too much silicon on the chip to be implemented at a reasonable cost. Further, a larger chip consumes more power and produces more heat. In the case of embedded real-time systems power can be a concern [113]. For non-embedded systems, power draw contributes to running costs which are desirable to minimise, and extra heat from the cache will lower the amount of heat the main processor can generate, limiting its clock speed.

In addition to the traditional viewpoint, Thiele and Wilhelm [101] identify that for real-time Systems, analysability is a concern. Suppose that there are two systems A and B, such that the performance of A beats B for all tests. It is possible that the static analysis techniques that can be applied to A are of poorer quality than B, which can in turn lead to static analysis suggesting the performance of B is greater than A.

If it were the case that the impact of a cache on WCET was negligible, then analysability would not be a major concern. However, it has already been established, fetching data and instructions from main memory is a significant overhead when executing a task. A comprehensive analysis of cache size on performance on the SPEC CPU 2000 benchmarks, performed by Cantin and Hill [26], suggests that a reasonably sized cache, with respect to cost of implementation, can obtain a hit rate of around 90%. Such a high hit rate means that if there is a corner case when the cache is ineffective,

```
if cache.classify(memoryLocation) == Miss:
    cache.evictAndReplace(memoryLocation)
cache.touch(memoryLocation)
```

Figure 2.6: A general cache replacement policy

the impact on the execution time (and hence WCET) could be huge. Hence in addition to the absolute performance of a cache policy or its implementation cost, for statically analysed real-time systems, the methods of analysis available to the cache policy are a major concern.

In general, a cache policy can be described by the simple algorithm given in Figure 2.6. The first step is to $classify$ the memory access. If the memory access is a miss, then the memory location must be inserted into the cache by an *evict* operation which determines which element should be removed according to the cache policy. After this, or in the case that the memory location is already in the cache, a *touch* operation should be applied, updating the cache state such that the cache knows the memory location has just been accessed. While the classification operation is independent of policy, the evict and touch operations differ, as well as the data structures they operate on.

In 2000, Mueller [75] set out the methods which are still used for cache analysis today. Mueller identified that in a system there may be many ways of reaching the same load instruction. Therefore many different cache states could be possible for each instruction, with each needing to be considered to check for a hit or miss. Hence Mueller gave the following five types of classification:

1. *Must Hit*: A memory access is classed as a Must Hit if a cache hit is guaranteed

2. *Miss*: A memory access is classed as a Miss if it a cache miss is guaranteed

3. *First-Miss*: If a memory access is a miss on its first execution but subsequently may not be, it is a First-Miss

4. *First-Hit*: If a memory access is a hit on its first execution but subsequently may not be, it is a First-Hit

5. *May Hit*: If no other classification can be applied, the memory access is classed as a May Hit.

Mueller's approach references neither model checking or abstract interpretation. Mueller himself refers to an "abstract cache state", but this could be either abstract interpretation or model checking as both support similar notions of abstract states in this context, and both could be used to implement this approach. The classifications which Mueller advocates are particularly of use in modelling cache behaviour for program loops; given that a program will spend the majority of its execution time in loops, this is justified. Refinements to cache analysis for loops are presented by Martin et al. [72], who use a combination of abstract interpretation and compiler optimisation techniques. This is accomplished by using a method named virtual inlining and unrolling (VIVU) on loops. VIVU modifies the CFG presented to the WCET analyser so that the number of basic blocks reachable from any point in the code is reduced as much as possible; this is of particular benefit to nested loops where it is possible for a number of positions in the CFG to be reached after executing the body of the innermost loop. By performing VIVU on the CFG it is possible to restrict the number of possible states to reach, and therefore derive a tighter bound on WCET in a lesser amount of time. Other examples of schemes based on or similar to Muellers approach are presented in [47, 73].

Mueller also identifies that of these First-Miss and First-Hit contribute less information to the analysis than Must/May/Miss. For this reason typical analyses [91, 54] do not check for First-Miss of First-Hit. Instead, to better match techniques such as abstract interpretation [36], most analyses compute an "upper" and "lower" bound on the cache state. The upper bound is used to determine which memory locations must reside in the cache; any access to a location determined to be in the upper bound of the cache must result in a cache hit, and hence the upper bound is called Must analysis. The lower bound is used to determine which memory locations may reside in the cache, and for a similar reason is called May Analysis. The name May Analysis is slightly misleading; the important result from performing a May Analysis is to determine when cache misses are guaranteed occur. If a cache miss cannot be guaranteed to occur, then the analysis will have to consider the possibility of a hit or a miss. In turn, this may result in multiple states being considered, which has a performance impact on the

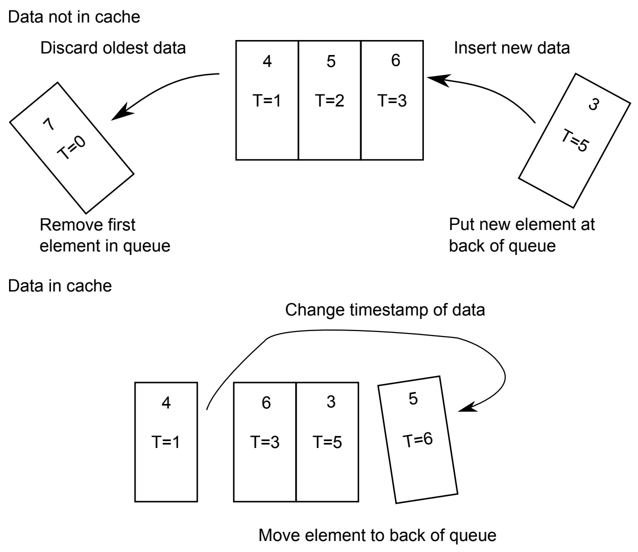


Figure 2.7: An LRU cache, represented as a list

analysis. However, if a Miss can be guaranteed, then a single successor state can be considered, which results in higher performance and less pessimism.

2.3.2 LRU Cache: A Solved Analysis

The Least Recently Used (LRU) cache policy is one the more obvious heuristics for a cache replacement policy, as mentioned earlier. LRU simply discards the least recently used element of the cache [55]. The assumption that this heuristic makes is that data which has been recently accessed is more likely to be accessed for a second time. While this heuristic fails for streaming data, which is accessed once and never again, it works relatively well for other types of data, for example the execution of program code. An exception for this case is when a loop of memory accesses is greater than the LRU cache size; for example a 2-way cache will not provide any caching for the sequence of accesses a, b, c, a, b, c.

Conceptually the simplest way to visualise an LRU cache is as a list, with position in the list representing the age of an element, as seen in Figure 2.7. With this representation, the evict and touch operations for the LRU cache can be defined as follows:

- EvictAndReplace(memoryLocation): replace the tail element of the cache with the new memory location.

- Touch(memoryLocation): move the indicated memory location from its current position in the list to the head of the list.

In practice there are many implementations of the LRU cache [98]. The simplest is the special case of the 2-way cache. In this case there is only a first and last element, hence it is sufficient to use a single bit of data to indicate the current last element. While this approach could be generalised to designate the tail of a list, in practice it is infeasible as when there are more than 2-ways an access may result in data having to be moved within the cache to updated positions in the list. Hence for higher associativities other methods are used.

One class of methods is to use age bits [98]. Age bits indicate the age of elements in the cache. The age bits are incremented when a cache element is demoted due to an access, and reset to zero when an element is accessed and therefore placed at the top of the cache. This method is efficient for relatively small associativity. When associativity is large age bits become inefficient, due to the difficulties in updating all age bits in each access.

For LRU caches with large associativity, methods such as a linked list become more efficient [98]. A linked list, unlike a plain array, has relatively efficient operations for moving elements within the list. However, the additional complexity of the hardware required to implement a linked list means that for small associativities other methods are more efficient.

A full Must/May analysis for the LRU cache was provided in 2007 by Sen and Srikant in [91]. The method exploits the conceptual list representation of the LRU cache. The touch operation of the LRU cache moves the specified element from its current location to the head of the list; this also has the effect of moving any elements inbetween the before the specified element and down the list by one. Using this information it is trivial to construct upper and lower bounds for each element in the list. It is then possible to construct a touch operator that works on the lists of upper and lower bounds as follows, by applying the following rules to each memory location m in the cache, when touching memory location t:

- Case 1: $m = t$: m is now at the head of the list; set its upper and lower bound to 0.

- Case 2: $lower(m) > upper(t)$: m will be moved down the list by t moving up; increment both upper and lower bounds by 1

- Case 2: Intervals $(lower(m), upper(m))$ $(lower(t), upper(t))$ intersect: m might be moved down the list by t, but the information present does not prove this. As such the lower bound remains the same, but the upper bound is incremented by 1.
- Case 3: $upper(m) > lower(t)$: m is already lower than t, so it will not be affected by t moving up; keep both bounds the same.

Here, the functions *upper* and *lower* denote the position of the cache line in the lists giving the upper and lower bounds respectively. They return ∞ when the cache line is not in the cache. To complete the analysis, the lists of upper and lower bounds can be used to classify cache accesses to element m as follows:

To complete the analysis, the lists of upper and lower bounds can be used to classify cache accesses to element m as follows:

- Case 1: $upper(m) < cacheSize$: m is guaranteed to be in the cache, so the classification is a $Must$.
- Case 2: $lower(m) < cacheSize < upper(m)$: m could be in the cache, but might not be. Hence the classification is a May
- Case 3: $cacheSize < lower(m)$: m cannot be in the cache, so the classification is a $Miss$.

The analysis of an LRU cache in this manner is very efficient, as all information on any number of concrete cache states can be represented by a single upper and lower bound. While the abstraction in this manner means that many impossible concrete states may be considered, these do not impact the accuracy of the analysis as the abstraction stores the best and worst cases accurately.

Unfortunately, even using the varied implementations of the LRU cache detailed in this section, the costs of implementing the LRU cache are deemed to be too high [55] for associativities greater than 2. Further, 4 or 8 way caches give much higher performance in benchmarks [26]. Hence it is necessary to use a different type of cache for high associativity. Even if an LRU cache is not feasible to implement for high associativity, the least recently used behaviour is a good cache heuristic in most use cases. Hence one method to keep the advantages of LRU while decreasing the cost in silicon is to approximate the behaviour or LRU.

2.3.3 PLRU Cache: Widely used, Not Fully Understood

The Pseudo Least Recently Used (PLRU) cache is an approximation of an LRU cache. In empirical testing, the PLRU cache scheme achieves a hit ratio almost as good as an equivalently sized LRU cache [55]. However, as the PLRU cache structure is designed to easily map onto a silicon implementation, the die area consumed by cache logic is greatly reduced when compared to the LRU cache for high associativities [55]. This in turn results in the cache requiring less silicon, and thus decreasing manufacturing costs and power usage. As these attributes are valuable to chip manufacturers, most high associativity caches are implemented using the PLRU scheme. As will be seen, the approximation of LRU results in behaviour which is more complicated to model. As a consequence, other than implementing the collecting semantics for the PLRU cache, current approaches are not able to fully analyse the behaviour of the PLRU: only partial static analyses are available. This means that static verification of the PLRU cache can only be accomplished by using the collecting semantics, which is computationally expensive.

Implementing a PLRU cache is accomplished by organising elements in a binary tree. Cache lines are stored on the leaves of the tree. Each node of the tree contains an additional bit of information that acts as a pointer. The pointers are used to approximate the least recently used element. This is accomplished by setting each pointer to point away from a memory location being updated during a touch operation; when the pointers are used to determine the element to evict, the pointers are simply followed and the element being pointed at is evicted. Hence the touch and evict operations can be defined specifically as:

- EvictAndReplace(memoryLocation): Take the path indicated by the pointers from the root of the tree to the indicated leaf; store the new memory location in the cache line on the indicated leaf.
- Touch(memoryLocation): For each pointer on the path between the root of the tree and the memory location indicated, set the pointer to point away from that path.

These behaviours are demonstrated in Figure 2.8, which shows a request for a memory location not in the cache, and hence results in an eviction

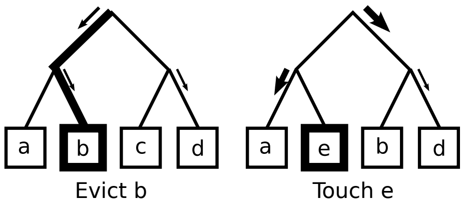


Figure 2.8: Showing the behaviours of a PLRU cache by demonstrating a cache eviction

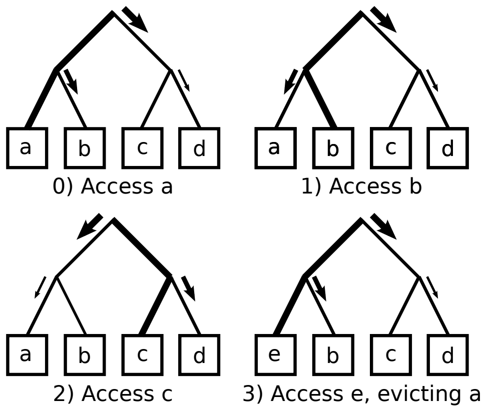


Figure 2.9: Illustrating the fastest a memory location can be evicted from a PLRU cache

operation (evicting the element which the pointers point to) followed by a touch operation (pointing the pointers away from the new cache element).

One slight complication in this description is the handling of invalid cache lines (the state of a cache line which contains no information, such as when the cache is initially turned on). Two policies exist in this case *sequential fill* and *tree fill* [54]. In the tree fill scheme, the pointers on the nodes of the tree always determine which cache line should be evicted. However, in the sequential fill scheme, if the cache contains an invalid cache line the first invalid cache line (using an arbitrary ordering) is selected. Sequential fill can cause complications in analysis as if it were not for this behaviour, left and right subtrees of a node in the PLRU tree containing the same abstract representation could be considered equal in all cases. With this behaviour, left and right subtrees cannot necessarily be considered equal if they contain two or more invalid cache lines.

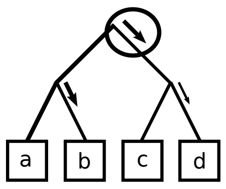


Figure 2.10: As an access to a sets the circled pointer, b can be unintentionally protected

The behaviour of a PLRU cache means that from each access to a cache element, it is guaranteed than the element will persist for at least $log_2(cache\_size) + 1$ accesses [56]. The worst case scenario that is used to find this bound is illustrated in Figure 2.9. The worst case can be proven by noting the fact that a memory access can only set a single pointer on the path to the required value; specifically, the pointer where the path to element a and the path to the element being accessed diverge. For all other pointers on the path to element a, the pointers will by definition be set to point away, as the paths are the same. Hence, to get all pointers on the path to element a to point to a, at least $cache\_height = log_2(cache\_size)$ memory accesses are necessary. Finally, an additional memory access is required to evict a, and hence the bound is $log_2(cache\_size) + 1$.

Unfortunately, unlike in the case of an LRU cache where a fixed number of misses guarantees an element is evicted, there is no similar bound for the PLRU cache. This is illustrated by Berg [10], who shows that it is trivial to construct a case where an element in the cache can still be resident in the cache after an arbitrarily high number of misses. This is illustrated in Figure 2.10, and exploits the tree structure of the PLRU cache. Specifically, where cache lines a and b reside in the same subtree, frequent accesses to cache line a will protect cache line b from eviction, by setting the pointers in the common part of the path of a and b to point away from b.

One of the interesting properties of a PLRU cache is that multiple cache states can exhibit the same logical behaviour. This can be observed by applying *subtree flipping*: a pointer in the tree is chosen, the pointer is flipped and the two child subtrees of that pointer are swapped; this does not change the cache lines being pointed at, but does change physical locations[3]. To

[3]Although this does not hold in the case of a sequential fill cache containing invalid cache lines, as in sequential fill the physical position of an invalid cache line is significant.

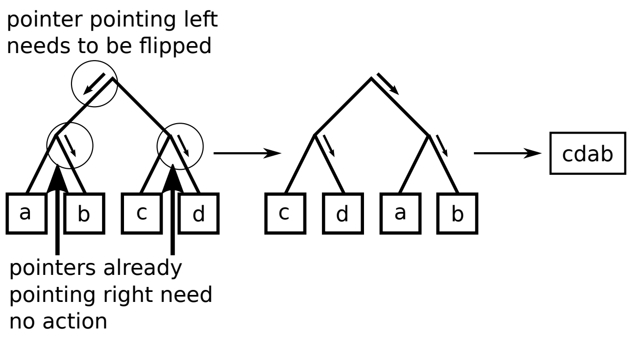
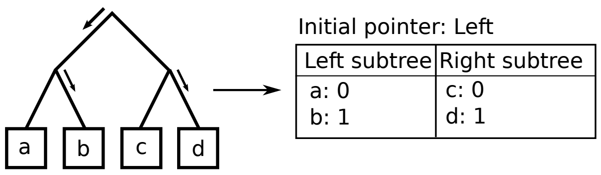

pointer pointing left needs to be flipped
a
b
c
d
c
d
a
b
cdab
pointers already pointing right need no action

Figure 2.11: Assigning a name to a tree fill PLRU cache

determine which states exhibit the same behaviour, a current technique is to use subtree flipping to ensure that all pointers have a specified orientation, as illustrated in Figure 2.11. As all cache states which exhibit the same behaviour are guaranteed to have the same name, and hence duplicates can be easily discarded; hence assigning names allows abstract interpretation to determine which states can be discarded. This scheme also enables a more compact representation of the cache state, as all pointers have a fixed orientation, and so do not need to be stored explicitly. Using the logical behaviours of a PLRU cache as a method of analysis implements the technique of collecting semantics for PLRU [37].

Initial work on providing a Must analysis of a PLRU cache was carried out in 2003 by Heckmann et al [56], and further elaborated on by Grund and Reineke [84] in 2008. As previously stated, any cache element requires at least $log_2(cache\_size) + 1$ memory accesses from its last memory access to be evicted. Hence, it can easily be inferred that a Must analysis on an LRU cache of size $log_2(k) + 1$ elements will provide a sound estimate of the elements that would be in the PLRU cache. The main problem with this approach is that it is a partial analysis which does not scale well with the size of the cache. Hence, while a cache of size 4 can have 3 elements analysed by this approach, doubling the cache size to 8 results in only 4 elements of the cache being analysed. Further, this approach does not yield any useful data about cache Misses, and hence cannot provide a May analysis.

More recently, in 2010, Grund and Reineke [54] provided an improved PLRU cache Must analysis, by utilising a metric named subtree distance. Subtree distances quantify the link between elements in the cache. By ap-



Initial pointer: Left

Left subtree	Right subtree
a: 0 b: 1	c: 0 d: 1

Figure 2.12: Mapping a PLRU cache state to Grund and Reinekes representation

proximating the subtree distance of elements within the cache to be either maximal or non-maximal, the analysis is able to have some knowledge of how access to specific elements affect other elements, thus enabling the exclusion of additional elements from eviction when compared to the previous work.

The implementation of Grund and Reineke's approach considers abstract cache states with 3 components: the root pointer and representations of the left and right subtrees. Elements in the subtrees are grouped by the maximum number of pointers that could be pointing to each element in the subtree, as in Figure 2.12. When the maximal number of pointers reaches $log_2(cache\_size)$ these elements have been considered for eviction, and hence fail the Must test.

By analysing the root pointer of the cache tree separately, Grund and Reinekes analysis effectively decreases the size of any cache trees which are approximated by 1. This results in the good results seen by Grund and Reineke for a 4-way cache [54], as the cache trees approximated have height 1, and hence behave identically to an LRU cache. However, while using the maximal number of pointers pointing to a cache element is a better approximation than using the Must analysis of an appropriately sized LRU cache, the same problem of scalability applies. Grund and Reinekes results for an 8-way cache indicate that only 6 elements of the cache can be analysed, as opposed to the entire cache in the 4-way case. This leads Grund and Reineke to conclude that their technique is limited to analysing $2log_2(cache\_size)$ elements, and hence this is still only a partial analysis. Further, the approach still does not yield any useful information for a May analysis.

These techniques will be revisited in Chapter 3.4.1, where they will be analysed from an information theoretic point of view. Chapter 4 will then perform a ground up evaluation of the PLRU cache to construct a new form of compression.

2.3.4 HNMRU Cache: A New Cache

The Hierarchical Not Most Recently Used (HNMRU) cache, proposed by Roy in 2010 [86] was created to further decrease costs for implementation of an effective cache policy while retaining the advantages of an LRU-like policy. The main principle of the HNMRU cache is a further generalisation of the PLRU cache policy; where a PLRU cache uses a binary tree, HNMRU uses a tree with a varying branching factor. An example of an HNMRU cache is illustrated in Figure 2.13. The cache shown is an HNMRU 4-2 cache, as the branching factor at the root is 4 and on the second level of the tree is 2.

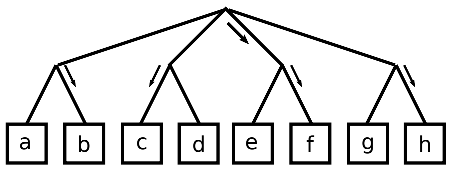


Figure 2.13: An example of an HNMRU 4-2 cache

HNMRU pointers differ from PLRU pointers in that they indicate the most recently used path, rather than the least recently used. In the case of an eviction, an element is chosen by starting from the root of the tree and randomly picking a subtree that is not pointed at; this applies the Not Most Recently Used policy at every step of the tree hierarchy, which gives the HNMRU policy its name. Roys implementation uses linear feedback shift registers (LFSR) [30] to implement the random number generation used for path selection. This choice is because LFSR is an RNG policy that can be implemented in a small amount of die area, and the main benefit for using HNMRU is greater control of the tradeoff between die area and performance.

Due to the fact that HNMRU is a recent development, the only analysis of HNMRU appears to be in Roys paper [86]. Roys analysis uses traces from a variety of benchmarks to simulate expected use of the HNMRU cache. These benchmarks show that the expected performance penalty of using an HNMRU policy over PLRU is approximately 5% for an 8-way cache. This is favourable as HNMRU can deliver approximately a 50% reduction in storage necessary for an 8-way cache. However, Roy's analysis does not take into account worst case execution time, as it is focused on expected performance rather than worst case. Hence Roy's analysis does not give information that would be required for the use of HNMRU in real-time systems. HNMRU would be desirable in real-time systems however, as many real-time systems also operate under energy constraints, and as the HNMRU policy uses substantially less die area for implementation there is a corresponding decrease in the amount of energy the cache requires.

HNMRU caches will be revisited in Chapter 4.6, where a method for their analysis will be proposed based on a generalisation of the corresponding method for PLRU caches.

2.3.5 Summary

Caches are inherently necessary to modern computer systems, due to the difference in speed between RAM and CPU. However, the presence of a cache complicates timing analysis, as the state of the cache must be modelled. If the model is too pessimistic, then the effect on estimated WCET can be relatively large, due to the frequency with which the cache is used. While an adequate analysis exists for the LRU cache, the same cannot be said of the more complicated PLRU and HNMRU cache schemes.

Current cache analysis techniques are revisited in Chapter 3.4.1, and a new method for PLRU and HNMRU cache analysis is given in Chapter 4.

2.4 Loop Bound Analysis

Loop bound analysis is the general problem of determining the number of times each loop in a program can execute. In general, this cannot be solved as loop bound analysis can be trivially equated to the halting problem. However, real-time systems are typically not the general case; as real-time systems are programmed with time constraints in mind, additional restrictions on the programming techniques used can be applied [82] such that it is possible to determine if the program halts. By extension, this means that upper bounds can be placed on the number of times each loop executes.

Loop bound analysis is a critical part of determining the worst case execution time of a program. As the previous section on caches demonstrated, a large amount of die area is given to cache. Instruction cache is primarily to speed the execution of loops [55]. While certain loops are trivial to place bounds on, for example those which have a fixed number of iterations, the conditions for a loop can also be complex and depend on multiple variables.

While techniques such as IPET (as mentioned in Section 2.2.3) provide bounds on loops by definition, such techniques either do not produce tight bounds or do not scale to large programs. This is because IPET techniques that do not perform loop bound analysis can either enumerate every possible path through the loop or make pessimistic assumptions about variables. In the former case the problem is intractable due to the difficulty of expressing loop invariants as ILP problems [111]; in the latter case bounds are not tight.

2.4.1 Value Analysis and Abstract Interpretation

In order to circumvent the problems of an exhaustive search via IPET, it is necessary to use more intelligent methods. As variables can be used in loops, Value Analysis plays a key role in Loop Bound analysis. Value analysis refers to techniques that attempt to predict the possible values that variables can take within the program. The benefits of value analysis are considered by Ermedahl and Gustafsson [43], who show how a value analysis can provide information which both enhances the precision of the WCET estimate and eliminates the need for some annotations. The first issue they address is eliminating false paths through the program, by using conditional statements to force conditions on the dependent variables for the remainder of the program. For example, if a program branch is executed conditional on $x > 20$, x should be assumed to be greater than 20 until the variable leaves scope. By extension this method can also apply to finding loop bounds by expressing the unrolling of the loop and expressing it as a series of `if` statements. Providing that the loop terminates, this analysis will terminate and find the bound on the loop because the conditions on the loop variable will continually tighten.

Unfortunately, performing Ermedahl and Gustafsson's analysis [43] in this manner produce a large number of states to evaluate. For this reason, they discuss the necessity of merging states, but note a distinction between states. Some states may be redundant and not contribute to the analysis; these states can be merged with no penalty. States which are not redundant may be merged as well, but in this case there is a penalty in the precision of the WCET estimate. This leads to using techniques to reduce the number of states to consider.

Of the current approaches to WCET estimation that implement value analysis, abstract interpretation is the favoured method ([100, 112, 47]). A reason for this is given in the analysis of the Coldfire processor by Ferdinand et al. [47]: values of program variables are typically unknown, but it may be possible to calculate intervals which the value is guaranteed within. This corresponds well to abstract interpretation, as the intervals are easily thought of as abstract states. Further, merging is easily thought of as a merging operator by simply picking an interval encompassing the two abstract states to be merged. Operations can then be defined on intervals which make sense; for instance, two intervals can be added together to find the interval which the result will lie within. The only issue that Ferdinand et al. identify with this approach is that these operations become undefined if the interval contains overflow values, because it is unknown if an overflow will actually happen in the concrete representation of the program.

Abstract Interpretation applied to Value Analysis is very close to the purest form of abstract interpretation, as defined by Cousot and Cousot [36]. In this case the process is iterative; information about variables is propagated through the statements of the program via a series of rules. The rules are not equivalent to executing the program; instead information is extracted about the possible values of a variable at each point. Typically, this is used to first compute a lower bound, and then an upper bound. After this step, abstract interpretation has found a *fixed point.* However, the information in the first fixed point found may not be adequate; for example, an upper bound may have been calculated as infinity. This can be corrected by iterating the process, using information from the previous iterations. Such information can be used to improve the tightness of subsequent fixed points, with the aim of removing overly large amounts of pessimism.

The information propagated in abstract interpretation can come from a variety of sources. An important first source is the initial assignment; in the case of a variable from an outside source, the range of values that the variable may take must be known. Further to this, statements inside the program can reveal information about possible values in certain segments of the program. For example, if a basic block is reached by traversing the conditional statement *if* $x > 0$, it can be inferred that for the duration of that block x is a positive number.

Many tools available depend on the use of additional information from the user, supplied in the form of annotations [82, 61, 100]. These annotations are used to provide information which is difficult to infer otherwise, such as the values that variables will take or loop bounds. Unfortunately, as annotations are supplied by the user, they are susceptible to human error, a point made by Prantl et al. [81]. Prantl et al. argue that due to this potential error it is necessary to verify annotations, in a similar way that the program itself may be tested. This is accomplished by exploiting the fact that it is possible to determine if a program is guaranteed to terminate within a given time. Hence if a WCET bound is given which involves the use of annotations, it is possible to verify it. Prantl et al. acknowledge one core failing of this approach: it is computationally expensive. To mitigate this, Prantl et al. state that the program analysis should be simplified first by another method, such as abstract interpretation.

However, many implementations of value (and by extension, loop bound analysis) do not verify annotations, and assume correctness. This can be seen in comparisons of such tools [112], where variations of the phrase "once the source code has been annotated" feature heavily, but with no mention of verification of such annotations, or mention of features to detect errors in annotations. Attempts have been made on the verification of annotations; notably Chapman et al. [27] intended to use the submit annotations in SPARK Ada to a theorem prover to check that annotations were correct, although this work was not completed. Also of note is that in benchmarks where some source code is not present, for example in the case of a shared library, the majority of tools compared in [112] exhibit critical problems, and leads to significant loss of tightness in others.

2.4.2 Summary

Loop bound analysis is a critical part of finding the worst case path through a program, and hence critical to WCET estimation. By extension, this means that Value Analysis is also critical. Currently, a weakness of common techniques is the use of unverified annotations. As unverified annotations may introduce incorrect information to the analysis, there is an argument that it would be desirable to reduce the dependence on such annotations.

An examination of current techniques in the context of information theory is presented in Chapter 3.4.2. Loop bound analysis itself will then be revisited in Chapter 5.

2.5 Common Problems

The problems encountered in WCET analysis fundamentally relate to the flow and preservation of Information. Information is critical to the success of any WCET analysis technique. For example, in LRU cache analysis [91] all relevant information can be easily condensed to a compact representation, due to the rules by which the information is treated. In the case of the PLRU cache, which has more complicated rules, information cannot be condensed in the same way. However, due to the state explosion problem [91], it is impossible to consider all information without some sort of compression. This leads to potentially valuable information being lost, as demonstrated in the techniques presented by Grund and Reineke [54], which cannot analyse all elements of an 8-way PLRU cache due to loss of information.

Information is again, a limiting factor in loop bound analysis. The fact that source code and detailed annotations are necessary to the implementations of static analysis tools [112] indicates that information is not being propagated through the analysis correctly. This is because regardless of optimisations performed, a compiler is expected to produce a functionally equivalent program. Provided that this assumption is true, a compiled program should contain as much information as the source code. Hence the question remains: why is the source code required? If the compiled program represents the same information, it should be possible to use it to derive the same properties.

Further, reliance on annotations which may not be correct [81], can introduce incorrect information to analysis. If possible, information guaranteed to be correct, such as valid possible values for input variables, should be relied upon instead. It can be theorised that if such information is propagated through a program analysis correctly, then additional information should not be necessary.

Given that Information is the limiting factor, it makes sense to question if frameworks exist that focus on the management, representation and compression of information. This is investigated in the next chapter, on Information Theory.

Chapter 3

Information Theory and Compression

As indicated in Chapter 2, a limiting factor in WCET analysis is the use of information; specifically, the representation and compression used in methods such as Abstract Interpretation and Symbolic Model Checking. These techniques utilise a form of lossy compression in that they discard information that is deemed uninteresting to the analysis. However, there is no formal method to identify what information should be considered “uninteresting”, instead relying on research to find what should be discarded. In contrast, the formal framework of Information Theory sets out how to classify different types of information, and existing compression methods provide insights on representation and how to identify information of low value to a specific goal. Hence this chapter introduces Information Theory and existing compression methods, and then examines how these concepts are used in existing techniques used in PLRU Cache analysis and Loop Bound analysis.

Information Theory [102] is the branch of mathematics dedicated to the analysis, handling and storage of Information. Historically, Information Theory was developed by Shannon in 1948 for use in analogue signal processing [92]; specifically, analogue signals which are prone to interference. Using Information Theory it is possible to specify precisely how much interference a signal can tolerate before it is unusable.

A prominent early example of Information Theory applied to compression is Huffman Encoding Subject [58]. Huffman encoding makes the assumption that the data it compresses can be modelled as a series of independent and identically distributed (i.i.d.) random variables, and further assumes that the distribution of the characters to be compressed is known in advance. Subject to these caveats, it is possible to prove that Huffman encoding is optimal [19] by applying Shannon's source coding theorem [92]. Unfortunately, most data for compression will not meet the i.i.d. assumption[96], as the i.i.d. assumption is specifically stated as follows:

- Independent: No sample affects another sample
- Identically Distributed: The chance of any result does not change between samples, i.e. $P(x = a) = P(y = a)$

Obviously, for many systems, the i.i.d. assumption will not hold. For example, the probability of the next word of a sentence is conditional on previous words, as some combinations of words do not make valid sentences. For example, given the beginning of the sentence "The quick brown fox jumped over the", the likelihood of the next word being "fence" is higher than the word "the", given that adding the word "the" doesn't make a valid sentence. However, if one were to analyse the words assuming that all words were i.i.d., one would come to the conclusion that the word "the" were more likely than "fence", given that the word "the" has already appeared twice in the sentence, and therefore evidence suggests that it appears more frequently than other words.

In practical terms, Information Theory also provides the language needed for the techniques given in this chapter [102]. Firstly, Information Theory distinguishes between "information", the abstract representation, and "data", a concrete instantiation of information (for example, the difference between a language and a sentence of that language). The next important definition is that there can be many different types of information in a system, where each type of information is represented by its own non-overlapping alphabet (for example, two different languages are different types of information). Finally, Information Theory (in particular data processing), allows notions of value to be placed on data (for example, shouting the word "Fire!" (usually) has a more important meaning than the word "Sheep!").

Although compression is typically applied to streams, the results are still used in other situations. In many cases when data can be indexed arbitrarily, treating the data as a stream makes problems more tractable. For example, many compression systems (e.g. .ZIP [41]) compress streams of data even if all the data is known in advance. As this thesis is primarily concerned with the application of compression, and in particular lossy compression, the remainder of the chapter will focus on such examples.

3.1 Lossless Compression

Lossless compression is the family of techniques that take information and reduce the size of its representation. The critical point for lossless techniques is that even though they reduce the size of the representation, the original can be reproduced entirely faithfully. This has many practical usages; for example, archiving data.

One important family of lossless compression techniques is the Lempel-Ziv (LZ) family. Developed in 1977 and 1978 respectively, LZ77 [114] and LZ78 [115] have had a large influence on compression techniques since then, in particular forming the basis of the compression used in the GIF format, and the DEFLATE algorithm used in both .ZIP and PNG formats. A particularly important variant on the LZ77 algorithm is the Lempel-Ziv Markov chain variant, LZMA [80]. LZMA's importance is that it is one of the currently leading general purpose compression algorithms, able to achieve much higher compression ratios than algorithms such as LZ77.

3.1.1 LZ77, Huffman and DEFLATE

The LZ77 [114] algorithm is particularly simple. As LZ77 advances through a file, it maintains a window of previously encountered data. If data in encountered which matches data held in the window the new data is encoded as a copy of data from the window, as illustrated in Figure 3.1. Hence the compressed file is encoded as a series of two types of block. The first type of block represents raw data, and the second represents data that should be reconstructed by copying previously decompressed data.

The representation of a copy block is a simple *(offset, length)* pair. The offset specifies how far back in the decompressed data to go to start copying the number of bytes specified by the length. One non-obvious benefit of this is that it allows the efficient representation of repeated data by specifying a length greater than the offset; this technique is illustrated in Figure 3.2.

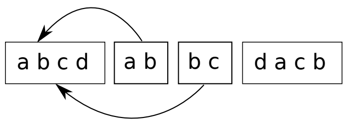


Figure 3.1: Finding patterns in a sliding window

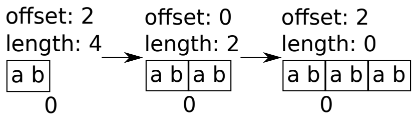


Figure 3.2: Representing patterns in LZ77

The principal is that as the decompressor works on a bytewise basis, when the length exceeds the offset the decompressor will initially copy the first instance of the repeated data in the normal manner, and then copy the subsequent instances from the previous instance.

A disadvantage of LZ77 is that as the size of the sliding window increases, finding repeated data becomes more computationally expensive. This is seen in that the standard LZ77 algorithm specifies sliding window sizes of up-to 32kB of data. If a sliding window is sufficiently small in comparison to the size of the input, then there is a high chance of patterns which are too long to be compressed, and hence compression is inefficient. A second disadvantage is that LZ77 provides only sequential access to compressed data; it cannot provide random access. This is overcome in some implementations by using LZ77 to compress blocks of data which are either small enough that complete decompression is inexpensive or simply do not require random access in the given use case. For example, .ZIP files split input data into a number of blocks and compress each block in isolation [41]. Each block is small enough that the loss of random access to the data in the block does not add a large overhead, and by performing this splitting the blocks can be accessed randomly and hence circumvent the sequential access requirement of pure LZ77.

As previously mentioned, when combined with Huffman encoding [58], LZ77 forms the basis of the DEFLATE algorithm [41]. Huffman encoding is a simple method of mapping a given alphabet to an alphabet which can be represent the same list of symbols more efficiently; approaches of this type are called *entropy encoding*. This is accomplished by knowing the probabilities of each symbol in advance and representing the more common symbols with fewer bits. The main problem which Huffman encoding overcomes is avoiding ambiguity in the resulting bitstream, which is accomplished by finding the smallest possible prefix-free alphabet. Hence any symbol represented by n bits cannot be a prefix of a symbol represented by $m > n$ bits, and hence the resulting encoding is unambiguous.

The actual implementation of DEFLATE [41], once LZ77 and Huffman encoding are defined, is very simple. The input data is split into a number of blocks. Each block is first compressed with the LZ77 algorithm with a dictionary size of up to 32B to remove duplicate data, and then compressed using Huffman encoding. Typically, the Huffman encoding used is specialised to the block; as all data in the block is known, it is possible to create a frequency table for the characters used to represent the block and construct the optimal Huffman encoding. However, it is also possible to use a predefined Huffman table if necessary, or if the additional space required to store the Huffman table is greater than the space the custom table would save.

The advantages of the DEFLATE algorithm are mainly historical: it is widely implemented and an accepted standard. Indeed, it is used within many other standards (e.g. PNG [14]). Whilst it does provide a useful amount of compression, the relatively small dictionary size used in the LZ77 stage of the algorithm limits its effectiveness. Further, the LZ77 portion of the algorithm requires a relatively high amount of string comparisons, a relatively expensive operation.

3.1.2 Legacy of LZ77

In the subsequent year, LZ77 was enhanced with the proposal of the LZ78 algorithm [115], which requires far fewer comparisons during compression and is therefore much faster. LZ78 was again enhanced with the proposal of the Lempel-Ziv-Welch (LZW) algorithm [107], which provides a simple improvement and is the compression method employed in the GIF file format [59]. Further, specialised variants such as LZMW [74], LZAP [97] and LZWL [28] have been created.

One of the current state of the art compression methods is LZMA, created by Pavlov [80] in 2001[1]. For many data sets, LZMA achieves the greatest compression when compared with other state of the art methods [80]. LZMA employs three major innovations over LZ77, the first of which is a much larger dictionary/sliding window when compared to LZ77. Secondly, in LZMA a character can be any length of bits rather than being fixed to 8 bits, meaning patterns across byte boundaries can be detected and compressed. Finally, the algorithm implements localised caching, such that recently detected patterns can be stored in fewer bits. All of these additions increase the computational complexity of LZMA, but due to the advances made in the 24 years between LZ77 and LZMA, modern computers are able to use LZMA in a practical amount of time.

3.1.3 Summary

Lossless techniques for compression are effective at reducing the size of the representation of information, without destroying information. The price for this is that compressed information tends to be slower to access, as additional computation must be performed when data is written or read to a compressed resource. However, lossless compression has limits [92], and whilst it can provide a moderate level of compression, higher levels of compression may be desirable. Importantly however, lossless compression demonstrates that the representation of data is important for the application of compression, as seen in the enhancements of LZMA over LZ77: enabling characters to have an arbitrary rather than fixed length improves compression.

3.2 Lossy Compression

The previous sections have dealt with Lossless encoding, but such approaches have limits on the amount of compression that can be achieved. These limits can either be mathematical limits such as the Shannon Source Coding Theorem [92] or simply given by the amount of resources available to find patterns for compression. However, common compression algorithms, such

[1]Unfortunately, LZMA is not the product of academic research, and as such there is no paper describing the algorithm. Indeed, the algorithm is poorly documented, and to the best of my knowledge, the only human language overview of the algorithm is found in an article on Wikipedia.

		Lossless		Lossy
	Raw	Archive (LZMA)	Audio (FLAC)	Audio (AAC)
Size (MB)	3053	1247	1327	61
Ratio	1.0	2.4	2.3	50.0

Figure 3.3: Compression of the Open Goldberg Variations [6]

as those used in audio and video, deliver very high compression ratios without having to expend significant resources. An example of this is seen in Figure 3.3, where a recording of the Goldberg Variations [6] is compressed, with two lossless techniques and the more common audio format, AAC. Whereas lossless compression manages to approximately halve the file size of the recording, AAC manages a reduction by a factor of fifty.

The trick used to increase the compression ratio so dramatically is that of lossy compression [106]. Whilst every piece of information is important, quite often some pieces of information are more important than others. Lossy compression takes advantage of this principal to discard information of low value to a particular result. However, due to this, lossy compression is not useful in every case. For example, in the case of archiving input files have to be recreated faithfully from the archive: in this case, all input information is of equal value, and hence it is not appropriate to discard any information.

When applying lossy compression, the value of discarding information is twofold. Obviously, information that is discarded does not have to be stored, and this will increase compression. However, in addition to this, it is possible that the information to be discarded can actually be substituted instead [106]. If the information is substituted, then patterns can be introduced into the information stream that can be exploited by a lossless compression technique. This can result in even less information being stored than by simple discarding; by introducing patterns that a compression algorithm can effectively exploit, large sections of the stream only have to be stored a single time, thus further reducing the amount of information to store.

Determining the value of information to discard via lossy compression is highly dependent on the specific application of the compression. This section will examine perhaps the most famous applications of lossy compression techniques, audio and video compression. In both cases there is an abundance of information which can be discarded, but very different principals apply to determining what should be discarded.

3.2.1 Audio Compression: MP3, AAC

Audio compression, such as MPEG Layer 3 (MP3) [18] or Advanced Audio Codec (AAC) [16], rely on psychoacoustic masking for compression [106]. Psychoacoustic masking describes the limits on human hearing, and specifically when the presence of one frequency of sound masks another.

First conceived of in 1979, psychoacoustic masking was independently developed by Schroeder [90] and Hill and Krasner [64], and was an initial failure for general purpose audio compression. Hill and Krasner succeeded in developing hardware that compressed the sound of the human voice, but was not useful for other sounds. Meanwhile, the more influential Schroeder was only able to prove negative results on the viability of the technique. It was not until later that psychoacoustic masking became a viable technique, with implementations of psychoacoustic codecs being well publicised in 1988 [17].

Psychoacoustic masking occurs due to how the human ear receives sound. In humans, the cochlea is the specific part of the ear which converts sound waves to nerve signals. To do this, it is lined with thousands of tiny hairs. Each of these hairs vibrates at a specific frequency, and when they do the human perceives sound of that frequency. Psychoacoustic masking describes how sounds can interfere with this process, by stopping the hairs of certain frequencies vibrating.

There are two main forms of psychoacoustic masking. The first of these is Simultaneous Masking [106]. Simultaneous masking occurs when the ear is unable to differentiate between multiple frequencies of sound. This results in only one of the frequencies being detected, with the others being lost. This occurs because there is a mismatch in the number of sound frequencies (uncountably infinite) and the number of hairs in the cochlea (finite). By necessity, this means that each hair in the cochlea detects a band of sound frequencies. If two sounds fall within the same band, then only one sound can be perceived. As evolution has specialised the human ear, these bands are not uniform. Instead, the bands are narrowest in the range of "human" sounds, and greater for "non-human" sounds. In particular, this means that at low frequencies the bands are large, and hence higher or lower frequencies are more difficult to distinguish. Further, as humans grow older the cochlea will likely experience damage from various sources, most commonly loud noise. This damage will further increase the size of the bands of sounds.

The second main form of psychoacoustic masking is Temporal Masking [106]. Temporal masking is an effect where sounds at different times mask each other. This is not the same as the deafness that occurs immediately after hearing a loud sound, which is caused by acoustic reflexes attempting to shield the ear. Instead, temporal masking occurs at a much smaller scale, and results in the inability to distinguish lower intensity sounds that occur immediately *before or after* a louder pulse of sound. The reason for this is due to the positioning of hairs within the cochlea; some sounds will disrupt the vibration of hairs in the cochlea either before or after they are heard, causing the masking effect.

The MP3 standard [18] was the state-of-the-art compression format for audio for a significant amount of time, and has become a de-facto standard. Designed by the Moving Picture Experts Group, MP3 employs the theory of psychoacoustic masking extensively to reduce the amount of data to compress, without compromising perceived audio quality. The same is true of the successor to MP3, the AAC standard [16].

Both MP3 and AAC do not define an encoding algorithm; only the decoding algorithm is specified, with various optional features [18, 16]. This is due to the fact that both codecs must make heuristic decisions, as an exhaustive search for the best compression is infeasible. Hence by only specifying the decoder, a number of encoders can exist, all producing compatible output. The various encoders can have different properties, for example being specialised to work on different source material.

In place of an encoding algorithm, MP3 and AAC specify various reference tables of the psychoacoustic effects the algorithms should use when selecting data to discard [18, 16]. These tables are given as they are only possible to calculate as the result of extensive, and qualitative, research. Every human ear is different, and hence constructing tables of psychoacoustic response is only possible by measuring the effects on a sample of people and selecting an “expected” masking profile, which would be reasonable to assume.

With the psychoacoustic profiling information available, the main remaining problem is to decide which information should be kept, and what should be discarded. The first problem to solve is the conversion of a sound wave to digital information; this process is called *sampling*, in an audio specific context, and more generally *quantisation* [49]. Assuming that sound is represented as a series of perfect wave functions, the Nyquist-Shannon Sampling Theorem [93] states that for a wave function containing frequencies of up to x hz, the wave function can be perfectly described with a sampling frequency of $2x$ hz. Whilst the mathematical assumptions made may not match reality, this result gives a useful description of what sampling frequencies are appropriate for different applications. For example, encoding low frequency sounds, such as whale songs, requires a much lower sampling frequency that higher pitched sounds. Hence, it is possible to compress lower frequency sounds much more effectively than high frequency sounds, as less information is needed to describe the sound.

The second parameter in determining what information to discard is the user deciding how much space they are willing to allocate to the compressed audio. As stated before, the MP3/AAC algorithms use heuristics; these heuristics will vary depending on how the user specifies the quality of the compressed audio. The specification gives three options: a constant bitrate (CBR), constant quality (CQ), and a hybrid approach of average bit rate (ABR) [18, 16]. The CBR approach attempts to maximise perceived quality given fixed resources, whilst the CQ approach behaves similarly, maximising compression at fixed quality. The ABR approach attempts to achieve a balance between the two, producing output with predictable quality and resource usage. However, in each case the heuristics used must vary. For example, CBR and CQ approaches do not have to anticipate the difficulty of encoding future segments of audio which is necessary for the ABR approach.

Finally, after information has been discarded from the audio stream, lossless compression can be applied. In the MP3 and AAC codecs, this amounts to applying Huffman encoding [58] to the stream. An additional benefit of discarding information that cannot be perceived by human ears, is that the amount of unique symbols decreases, and the number of repeated symbols increases. This in turn leads to Huffman compression becoming more effective.

Whilst MP3 and AAC codecs attempt to discard information which the human ear cannot hear, this is not always the case. In particular, when there is not enough capacity to compress the audio accurately, compression artifacts will be observed, as studied by Liu et al. [69]. A common example of this is applause; applause is essentially random noise and therefore largely incompressible. If there is not enough capacity to represent the applause accurately, then artifacts will be introduced as the least important information is still perceivable. In the case of applause, this is a distinct metallic ringing sound due to Simultaneous masking being applied in excess. In the case of Temporal Masking being applied excessively, an “echo” of the sound can be heard, although as with Temporal masking working in both directions of time, the echo may be heard before or after the sound which caused it.

The main things to observe in the relatively simple case of audio compression are the transformation of input data to a form more amenable to compression and that there is a clear scientific grounding for choosing the information which is discarded. These properties are continued in the vastly more complex case of video compression.

3.2.2 Video Compression: h264

Another well known use of lossy compression is video compression, with a current example being the h264 algorithm [110]. Whilst the problems faced by video compression are similar to those in audio compression, video streams contain far more data than audio streams; an audio stream is merely a 1-dimensional sequence of samples, whereas video data comprises 3-dimensions: x and y for position within frames, as well as time to advance between frames.

The vastly increased amount of data has two implications: the first is that much more data must be discarded in order to reach an acceptable compressed data size. However, the second is that there are more strategies available to discard information; the increased dimensions enable patterns to be found on a single piece of data in more dimensions.

In practice, searching three dimensions for patterns is computationally infeasible. Due to this the types of compression implemented by the MPEG and h264 algorithms are actually quite restricted. To perform compression over time, three video frame types are specified: I-frames, P-frames and B-frames [110]. I-frames are the simplest and serve as a point of reference; they simply store a compressed image taken from the source material.

By contrast, P-frames make use of information from the previous frame, encoding the difference with techniques such a motion vectors [110]. The image is divided into fixed sized blocks, and these blocks are compared against the previous frame, finding the closest matching area. The difference between the block and the area in the previous frame is taken, and then compressed. This results in compression as frequently, for example is scenes where the background is static, many blocks will not change and hence do not require space to represent in the P-frame. B-frames are similar to P-frames, with the difference being that a B-frame can also reference the subsequent frame.

To encode each type of frame, the MPEG and h264 codecs specify a number of transforms that have been picked to minimise perceived image quality loss. However, when video frames are in their basic form, it is not easy to either specify transforms or pick which transforms are most applicable. To combat this, the source frames are first transformed, primarily using transformations based on the Discrete Cosine Transformation (DCT) developed in 1974 by Ahmed et al. [1]. The DCT transform is not itself a lossy transform; instead it represents portions of the frame as the sums of weighted cosine functions at different frequencies.

Once the blocks are DCT encoded, they are in a form more amenable to compression. One significant technique employed, as with audio, is quantisation [49]. However, whilst quantisation in audio is a necessary process to convert an analogue signal to digital, in this application quantisation is used to trim the amount of space required to store the DCT encoded blocks. By stripping out the contributions of cosine functions with low weights, the DCT block can be represented using a smaller amount of data. Further, as the cosine functions removed have a low weight, they have a low contribution to the overall picture and hence can be argued to be not as valuable as those with a high weight. This process does have caveats attached: specifically, the contribution of low-weight cosine functions is necessary to reproduce subtle details. Hence removing the low-weight cosine functions introduces visual artifacts, as illustrated in Figure 3.4.

The h264 codec goes a step further than MPEG by attempting to minimise the appearance of artifacts. This is accomplished by a mandatory deblocking filter [85], which minimises the visual appearance of artifacts. Whilst this filter does not stop the artifacts from appearing, it minimises the visual impact of the artifacts by replacing clearly visible hard artifacts with artifacts that are less visible, as seen in Figure 3.4. As the compression artifacts are made less visible by the deblocking filter, the h264 codec

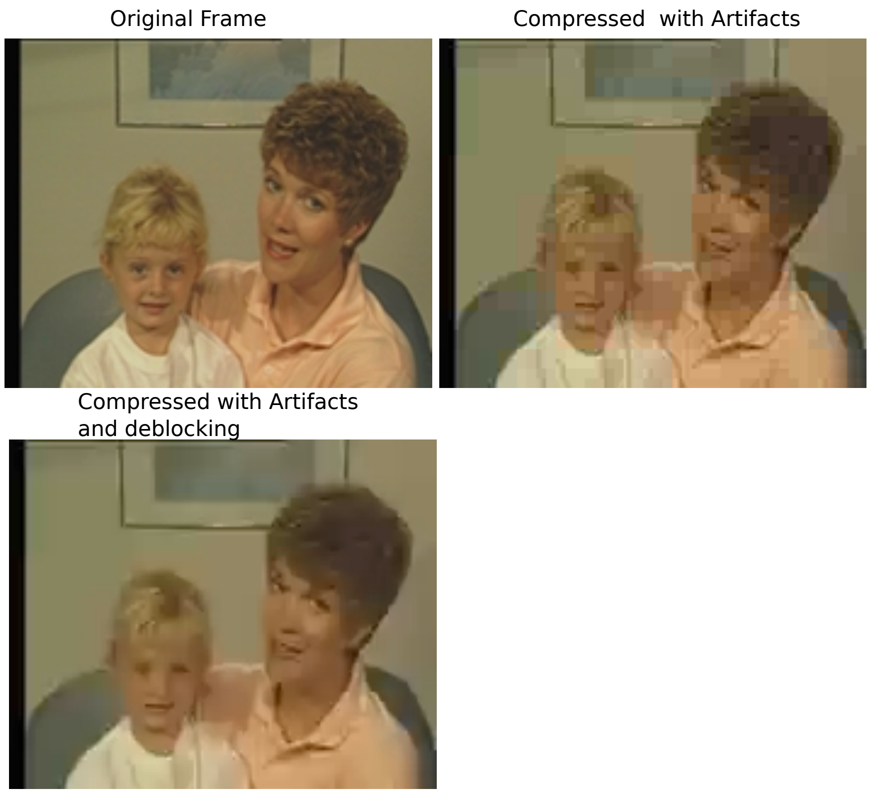


Figure 3.4: Blocking artifacts in video compression. Images from Richardson [85]. The differences between these images are much clearer when viewed in colour.

is able to be more aggressive in the quantisation step , resulting in higher compression with fewer visible artifacts.

In contrast to audio compression, the vastly increased amount of data in video means that visual artifacts are almost inevitable, even when discarding information which according to scientific analysis is less likely to be noticed. Hence in addition to the transformation of input and using scientific method to determine what to discard, video compression introduces the notion of a recovery strategy: in this case, substituting data likely to be noticed as compressed with an alternate representation which is less noticeable.

3.2.3 Summary

Lossy compression can exceed the limits of lossless compression by discarding unnecessary or unimportant data. The drawback to this is that it is impossible to faithfully recreate the original source. However, if done correctly, this is irrelevant: by discarding low value information, it should be impossible, or at least difficult to detect that changes have occurred.

3.3 Common Themes in Formal Compression Approaches

The examples of compression given illustrate a variety of approaches to reducing a given quantity of data. Lossless compression revolves around finding a compact representation for data. This can be accomplished either by adjusting the alphabet of the data, as in Huffman encoding [58], or by exploiting patterns in the data as in Lempel-Ziv encoding [114]. Further, these approaches are not mutually exclusive, and can be combined, as in the DEFLATE algorithm [41].

In lossy compression, a common theme is to discard information that is deemed to be least valuable. In the case of audio encoding, this is information describing audio which cannot be heard due to psychoacoustic effects [18, 16]. Discarding useless information obviously results in less information to store, and hence higher compression. Further, it is sometimes necessary to transform the data before it becomes obvious what is of least value, such as the DCT transform commonly employed in video encoding [110].

Further, lossy compression approaches must also be able to deal with the fact that discarding information can introduce artifacts that must be dealt with. This is seen in the deblocking filter used by the h264 codec [85], which attempts to reduce the visual impact of artifacts introduced by the encoding process. Whilst such a filter is strictly speaking unnecessary for compression, it reduces the effect of information loss, and hence more information can be lost.

In summary, this leads to four main techniques used in general compression:

- **Changing Representation:** Changing the representation of information such that it either takes less space, or enables another technique to be applied. Examples: Huffman Encoding, DCT Transformation in Video Encoding
- **Pattern Compression:** Finding and then efficiently encoding patterns within a stream of symbols. Examples: Lempel-Ziv encoding
- **Lossy Compression / Discarding Information:** Discarding information so less information needs to be stored. Examples: Audio / Video Compression
- **Recovery:** Ways of handling the absence of information that was discarded during the encoding, such that the effect of information loss is minimised. Example: Deblocking in h264

Some of these techniques may appear familiar: this is because these techniques are already in use, but are not formally identified. The next section will revisit the existing approaches to the problems of PLRU cache analysis (Section 2.3.3) and loop bound (Section 2.4), and identify how these techniques have been used in previous approaches.

3.4 Compression in Existing Techniques

Whilst applying the principals of lossy compression to the problems faced by static analysis has not, to the best of my knowledge, been done before, existing techniques do use many of the same ideas. This section examines how abstract interpretation and symbolic model checking make use of lossy compression, and identifies weaknesses where a formal approach could help.

In principal, Symbolic model checking [22] has a very simple map to the principals of lossy compression. Symbolic model checking works by defining a map between the concrete state space and a symbolic state space, which loses precision to reduce the number of states to check. This map can be thought of in much the same way as how lossy compression maps between the source and the compressed output. The step needed to see symbolic model checking as lossy compression is to realise that rather than considering individual states, all states at any step of symbolic model checking are compressed together. This allows the merge operation to be applied to states which are similar to each other.

Abstract Interpretation [36] maps onto lossy compression in a similar way. To reduce the number of states to consider, abstract interpretation makes use of an abstraction function. This function takes the current set of states and outputs a simplified version. The exact simplification made is left to the implementer of the technique and determined by the type of problem being considered. Hence whenever the abstraction function is used to reduce the number of states, lossy compression occurs as the simplification will, by definition, lose information.

3.4.1 Lossy Compression in PLRU Cache Approximation

As introduced in Chapter 2, two main forms of analysis for PLRU caches exist; the LRU-approximation given by Heckmann et al. [56] and the Potential Leading Zero (PLZ) approach given by Grund and Reineke [54]. In addition, the practice of considering logical cache states rather than physical cache states, the Collecting Semantics [37] of a PLRU cache, also discards information. However, as the only information discarded by the Collecting Semantics is the precise concrete states encountered, by instead keeping track of all behaviours encountered, one can conclude that the information discarded by Collecting Semantics has no value. In contrast, as the LRU and PLZ approaches do not produce useful results for PLRU caches with associativity of at least 8, it can be inferred that these approaches discard information of significant value.

In the approached used by Heckmann et al. [56], the loss of information is by necessity: Heckmann et al. do not propose rules for tracking when the information could be evicted, and hence the only possible method is to discard that information and make no decisions using that information. The

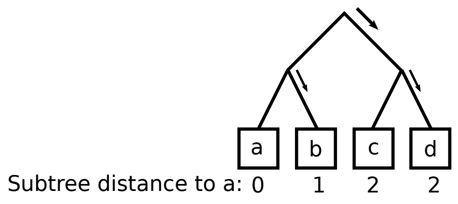


Figure 3.5: Subtree distances to element a in a 4-way cache

loss of such a large amount of information is that the analysis is incomplete, and hence has high pessimism: even for small caches a large number of definite hits will be classified as "May Hit".

Grund and Reineke's Potential Leading Zeros [54] approach to PLRU Must analysis is provided as a form of abstract interpretation, and hence as previously stated discards information by using its abstraction function. Specifically, an approximation of the number of pointers that point to each element of the cache is made (the "potential leading zeros" that the analysis is named for), as previously shown in Figure 2.12. In order to merge states, similar states are identified by the subtree distances of the elements within the state (Figure 3.5, which provides a partial representation of the tree structure).

Grund and Reineke's algorithm approximates the subtree distances between two cache lines as being either 0, non-maximal, maximal or unknown. In the case of Figure 3.5, a has subtree distance 0 to a itself, a non-maximal subtree distance to b, and maximal subtree distance to c and d. Effectively, this partitions the cache into two subtrees, which allows the analysis to deduce that an access to a cache line within the left subtree of the root node will not impact the right subtree, and vice versa. Any cache states which have the same subtree distance approximations for their elements can be merged, as seen in Figure 3.6, with the number of potential leading zeros being upper bounded. When the number of potential leading zeros for a cache line reaches the height of the subtree it resides in, then the analysis considers that the cache line could be evicted at the next eviction.

The main problem with Grund and Reinkes approach is that it is incomplete: it can not analyse all elements of caches with at least 8 ways. Hence even though the approach succeeds in reducing the number of states to be considered, it does not provide a full cache analysis. This can be

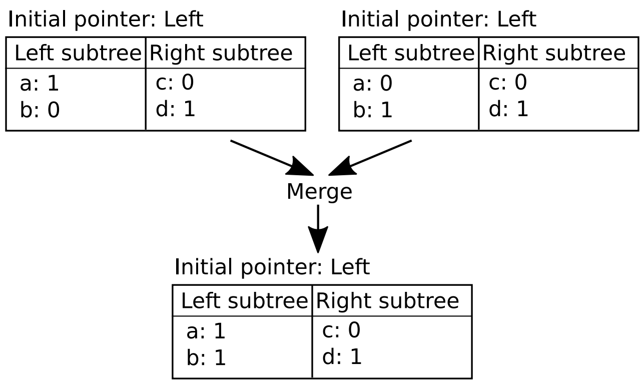


Figure 3.6: Merging states in Grund and Reinekes Potential Leading Zeros analysis

viewed as discarding too much information; Grund and Reineke's primary motivation for targeting the number of potential leading zeros for approximation appears to be soundness. This is because the behaviour of Grund and Reineke's approach is similar to the technique of Heckmann et al. of using an LRU cache to provide a Must analysis [56]. This is observed as in Grund and Reineke's algorithm, each subtree can be equivalently modelled as an LRU cache of size $log_2(N/2)$, for an N-way cache, which is the same as performing Heckmann et al.'s analysis on each subtree. Hence the property of soundness is easily proved.

Whilst soundness is a desirable property, the consequence of discarding too much information is the appearance of *compression artefacts*: the discrepancies between the compressed representation and the actual system. In the case of Grund and Reineke's approach [54], compression artifacts lead to considering additional states that could never be encountered. The major concern is that Grund and Reineke's approach can never determine that any element has been evicted for a cache with $N \geq 8$ ways. This immediately causes state space explosion: if no element can be provably evicted, then trivially after accessing k unique memory blocks, all possible combinations of those cache elements must be considered. Hence for k memory blocks, this quantity is expressed as ${}^kC_N$, and is equal to $\frac{k!}{N!(k-N)!}$. However, as each cache is represented as two subtrees, the number of states considered will likely be greater[2]. Regardless of the exact quantity, this leaves a lower

[2]Experimentally, it rises at a rate of $O(2^k)$.

bound on the number of states considered that rises at least as fast as $\Omega(k^N)$. Given that the number of unique memory locations considered is expected to be large, this is clearly an undesirable property.

As too much information is discarded by Grund and Reineke's approach, and this leads to state explosion, it could be argued that considering the problem from an explicitly information theoretic point of view could help. By determining the effects of removing each type of information contained in the cache state on the size of the state space considered, it should be possible to devise an analysis which is both sound and doesn't lead to state explosion. This could be obtained by retaining enough information to perform an effective May analysis, thus allowing elements to be provably evicted and hence reducing the number of cache states to be considered.

In contrast to the approaches of Heckmann et al. [56] and Grund and Reineke [54] approaches, using the Collecting Semantics [37] of a cache does not discard valuable information. It does however discard information: specifically, when cache states have different physical representations but all exhibit the same logical behaviour, only one of these cache states need be evaluated and the rest discarded. This information is not relevant to the analysis as the analysis is not concerned with which physical states may be reached, but the logical behaviours that may be observed. Hence, in PLRU analysis, there does exist a form of lossy compression which only discards information that has no value to the analysis. Unfortunately, the technique of Collecting Semantics does not discard enough information to be tractable. Hence any form of tractable analysis must discard information which has value: the main question is in determining which information is of least value and how it may be discarded without impacting the analysis.

3.4.2 Lossy Compression in Loop Bound Analysis

Loop bound analysis is typically accomplished by either symbolic model checking [22] or abstract interpretation [46]. As mentioned previously, both of these techniques use lossy compression to reduce the size of the state space. In the case of loop bound analysis, the lossy compression takes the form of merging states which are deemed to be identical or close-enough.

The main problem comes in defining "close-enough". If too many states are considered, then the analysis becomes intractable; Too few, and the analysis will be overly pessimistic. This problem is characterised by Bullock and Silverman [21], who identify that in any computational simulation there is a trade off between the tractability of the problem and accuracy[3].

It can be argued that one of the problems caused by excessive compression of the state space is the need for annotations to provide information that the compression could have inferred but discarded, as observed by Ermedahl and Gustafsson [43]. Hence, it could be argued that provided that the bounds of input variables are well defined, it should be possible to propagate this information through the analysis of the task code, avoiding discarding useful information. In this case, annotations would not be required as any information from an annotation could be inferred automatically, rendering the annotation unnecessary. On the other hand, for an analysis technique that requires annotations, it can be inferred that useful information was not propagated successfully at some point in the analysis, and therefore discarded. Hence the reason for the required annotations is to reintroduce information lost as a result of excessive compression applied by the analysis technique.

Hence there could be two benefits to considering this problem from a more formal approach. The first of these is that the usage of information within the program can be analysed; for example, to distinguish between variables which take a small set of discrete values and those that occupy ranges. If the analysis is capable of determining these properties, then appropriate representations can be picked which allow for the most compression whilst keeping the amount of pessimism low.

The second benefit to a formal approach is that by introducing a measurement for the amount of information in the analysis at any given point, it is possible to keep track of the information within the system. This can aid the development of analyses, by identifying the points at which an approximation discards information which could later be useful. If these points are known, then the algorithm can be altered to attempt to avoid or minimise the information lost.

An additional argument can be made that current approaches may be too general. Abstract Interpretation, as applied to Loop Bound Analysis by [46], attempts to prove the existence of loop bounds. Trivially, this is impossible in general as it would solve the halting problem. Hence, it

[3]Also the generality and realism of the simulation, as identified by Levins [65], but these properties are non-negotiable for WCET analysis.

is expected that the program obeys conventions or rules, in the form of restricted programming practices, that guarantee that a loop bound exists [61]. This in turn means that it is unnecessary for loop bound analysis to prove the existence of loop bounds; it would be acceptable to assume the existence of loop bounds. If assuming the existence of loop bounds, then the problem of finding the loop bounds becomes computable [39], and does not violate the halting problem. While the assumption of the existence of loop bounds is a major strengthening on the requirements for the input program, it also allows a corresponding relaxation to be made on the requirements for testing: specifically, annotations are now no longer required.

3.5 Goals for using a more formal approach

As illustrated by the analysis of lossy compression in current techniques, the ideas of using lossy compression to reduce the state space of an analysis are critical to current approaches [56, 54, 37, 46]. However, current approaches to defining how to implement said compression are not formal; the basic idea used appears to be for an interesting quantity to be identified and then approximated. Whilst this may yield good behaviour, there is no guarantee. For example, Grund and Reineke's PLRU-PLZ [54] approach was selected because the number of potential leading zeros does give an accurate description of when a cache element can be selected for eviction. However, the approximation selected does not yield an accurate analysis due to important information being lost.

In contrast, lossy compression in fields such as audio [18, 16] or video compression [110] chooses to loose information for which there is an argument that the information is not important. For example, audio which cannot be perceived by the human ear, or detail which cannot be detected in motion. Once a sufficient amount of information has been discarded, the information is further compressed using a form of lossless compression [58]. Of note is that these approaches demonstrate a formal approach, where information is classified according to value and the least valuable information is discarded.

In principal, there is no reason why such a formal approach could not be applied to problems such as those found in static analysis of Real Time Systems, in particular those which rely on exploring a state space. These problems are typically found in modelling computer systems for the purpose of finding an estimate on the WCET of a task. Such an approach would be comprised of two main parts: first the representation of the information can be chosen such that information is stored efficiently: a type of lossless compression. Once the representation of the information is decided, information can be tracked throughout the analysis, and only low value information can be chosen for being discarded: a form of lossy compression. By only discarding low value information, and having an appropriate recovery strategy, the goal would be to reduce the number of states to consider to a level acceptable to the end user.

3.6 Outline of a Formal Approach

Given the usage of lossy compression in traditional techniques, such as audio and video compression, and the current implicit usage in abstract interpretation and symbolic model checking, it is possible to provide a general outline of how a more formal approach can be used to find appropriate simplifications. This can be given as the following steps, with corresponding examples from MP3 compression:

1. *Types of Information*: Firstly, it is necessary to identify the distinct types of information within the system being modelled. Types of information are determined by constructing the minimal alphabets needed to represent unambiguously all data within the concrete system.
    - Example: finding frequencies in a sound.
2. *Value of Information*: After types of information are found, it is necessary to argue for the value of each type of information as used within the system. This can be accomplished by considering the use of the information within the system. Such an argument should take into account the frequency of use of the information, with less frequently used information being less valuable, the amount of information stored in each instance of data, and the consequences of the information being inaccurate. Simple experimentation, by trivially discarding information of a certain type, may be useful here to provide evidence that an argument is correct.

- Example: Perceptibility of frequencies in a sound at a given time due to hearing range or psychoacoustic effects.

3. *Overall Strategy*: Having decided on which types of information are least valuable, the next step is to decide how much information should be discarded by the lossy compression. All data of no value should automatically be discarded, and in addition enough low value data also needs to be discarded to make the analysis tractable. These decision on what to discard will then shape the choices in the remaining steps.

    - Example: Strategy of discarding all imperceptible frequencies when they occur, approximating the remainder using MDCT.

4. *Representation*: Once an overall strategy has been determined, a suitable representation should be found. The representation used should be picked such that it is computationally efficient to discard any information marked for removal by the strategy.

    - Example: Represent sound by applying quantisation and MDCT to the source.

5. *Approximation Operator*: Next, an approximation operator must be defined, which successfully implements the discarding of low value information.

    - Example: Discarding imperceptible frequencies and removing low value components of the MDCT transformation.

6. *Recovery Strategy*: Finally, if information that has value to the analysis is discarded, it is necessary to implement a recovery strategy to cope with the information loss. Such a strategy has to ensure that the property of soundness still holds in the analysis, despite information having been discarded.

    - Example: Replacing sounds which have been discarded with white noise of an appropriate volume during playback.

Having outlined an overall strategy for devising new analyses with a more formal approach, the subsequent two chapters use this approach to implement new analyses. Chapter 4 applies this approach to the PLRU and

HNMRU caches, and Chapter 5 applies the same principles to Loop Bound analysis.

Chapter 4

Lossy Compression for Tree Based Caches

A PLRU cache is a commonly used type of cache due to its relatively good performance in comparison with LRU and reduced circuit complexity. However, the logical behaviour of a PLRU cache is more complicated than the LRU cache, which results in fewer obvious abstractions being available. As introduced in Chapter 2, previous attempts to analyse the behaviour of a PLRU cache have resulted in incomplete analyses [56, 54], unable to analyse the cache completely.

As previously mentioned in Chapter 3, lossy compression exists in current techniques for modelling PLRU caches. Even utilising Collecting Semantics [37], a relatively exhaustive process, on a PLRU cache implements a form of lossy compression by not considering every physical state. Instead, only unique behaviours are considered; any cache state encountered which duplicates behaviour already seen is discarded. This is illustrated in Figure 4.1, which shows an intuitive method of assigning a name to the logical

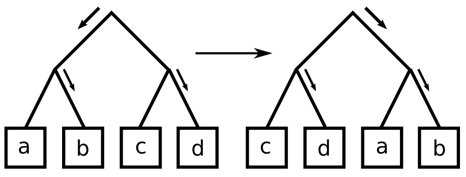


Figure 4.1: Putting all pointers to the right results in assigning a name to the logical state of a cache

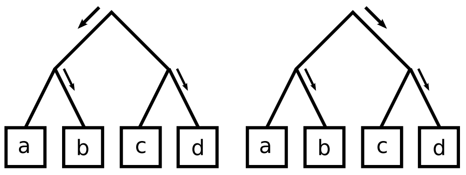


Figure 4.2: Two cache states that differ by a single bit, but have significantly different representation in the normalised form

behaviour of a tree fill cache state - by using subtree swapping to set all pointers to a given direction. Clearly, as cache analysis is only concerned with the behaviour of a cache, the fact that there are multiple ways to trigger the same behaviour is not useful to the analysis, and hence can be discarded without consequence.

In a tree fill PLRU cache, implementing the Collecting Semantics uses the fact that each pair of subtrees in the cache are logically equivalent. Hence it is possible to flip subtrees around their shared pointer, provided that the value of the pointer is also changed. One scheme for this is to flip subtrees until every pointer points the same way, thus removing the need to store pointers and generating a more compact representation.

To further improve on this compression, it is necessary to look inside cache states for information which can be discarded. This technique is used in the approaches of both Heckmann et al. [56] and Grund and Reineke [54]. Both of these analyses perform an approximation of each cache state considered, and hence are able to merge cache states exhibiting similar behaviour. However, in both cases, too much information is discarded, resulting in analyses which are unable to completely analyse a PLRU cache.

The difficulty of compression in this form is illustrated in Figure 4.2. This shows two cache states which differ only by a single bit, but have significantly different representations when using the intuitive naming method illustrated in Figure 4.1. Evidently, such differences in representation will make any compression harder to accomplish as similarities between the states are more difficult to find. Further, a form of natural recovery is possible in PLRU cache analysis because each cache access overwrites portions of the cache state; pointers are overwritten by touch operations, and memory blocks by evictions. If all uncertainties in a cache state can be overwritten, then the cache state has recovered without penalty. Hence it would be desirable that any merged state only discards the absolute minimum information necessary; in the example of Figure 4.2, this would mean only discarding the single bit of difference. In order to address these problems, the steps outlined in Chapter 3.6 will be applied, so that a more effective representation can be found.

Overview

The previous chapter presented the argument that abstract interpretation, as seen in PLRU cache analysis, is a form of lossy compression, and also showed that more common instances of lossy compression (such as audio and video) use a more formal approach to devising compression than commonly seen in abstract interpretation. Therefore, it can be argued that approaching the problem of abstract interpretation for PLRU caches with a more formal lossy compression based approach would be beneficial. Specifically, it would be desirable to determine what information within a PLRU cache state can be discarded with little penalty. This is explored in Section 4.1, which makes predictions based on the principles of lossy compression about what information can be discarded. This is continued in Section 4.1.2, where a simple experiment is conducted to verify that the predictions are indeed valid.

In Section 4.2, conventions and notation are outlined, which are used throughout the definition of the new technique. Section 4.3 gives the details of an algorithm which implements a scheme based on the results of Sections 4.1 and 4.1.2 and the methods used to manipulate a tree-fill PLRU cache; this is formalised in Section 4.4. Continuing this, in Section 4.5 a proof is constructed which shows that the proposed algorithm is sound.

Further, as also introduced in Chapter 2, a relative of the PLRU cache exists, in the form of the HNMRU cache [86]. As HNMRU caches have no analysis available at present, but are similar to the PLRU cache, it would be desirable to adapt the PLRU analysis to accept HNMRU caches. This is accomplished for a modified form of HNMRU cache in Section 4.6.

An experimental evaluation of the new approach is carried out in Section 4.7. The first part of the experimental evaluation compares the new algorithm with the previous algorithms available, while the second part examines the relative analysable performance of PLRU and HNMRU caches of the same size. Finally, the work in this chapter is summarised in Section

4.8.

4.1 Information Theory and PLRU Caches

4.1.1 Information in a PLRU Cache State

The first step is to determine the different *types of information* used to represent a PLRU cache. A type of information is defined by the minimal alphabet that is needed to represent it. Information using a different alphabet is counted as a different type. In the system of a tree-fill PLRU cache, there are two quantities that affect this encoding: the number of cache ways N, and the set of possible memory blocks that the cache could contain, L. With this information, it is possible to identify three distinct types of information:

- *Pointers*: The pointers on the nodes of the cache tree. These are represented by the alphabet of $\{0, 1\}$, and hence consume a single bit of information. In total, for an N-way cache, there are $N - 1$ pointers, meaning $N - 1$ bits of information are used for their representation. Pointers are used to determine which cache line to evict.

- *Cache Lines*: Each cache line contains either a memory block or is invalid. Hence they are represented by the union of the set of L possible memory blocks and a single invalid state. As each cache line is drawn from the pool of possible memory blocks, for a cache of size N there are ${}^{|L|}C_N = \frac{N!}{|L|!(|L|-N)!}$ sets of possible cache lines. Cache Lines are used to determine if a given access is a hit or a miss.

- *Tree Structure*: The tree structure describes the position of each cache line and pointer within the cache, and hence is represented by an alphabet of orderings. Hence the tree structure is comprised of two parts: the ordering of pointers (which, given $N - 1$ pointers, has a size of ${}^{N-1}P_{N-1} = (N - 1)!$) and the ordering of cache lines (which, given N cache lines, has a size of ${}^{N}P_N = N!$), and therefore has an alphabet size of $(N - 1)!N!$. However, in most representations the tree structure will be implicit, as both pointers and cache lines will be represented with an implicit notion of ordering (e.g. as in a list).

The tree structure is used in both evictions and touch operations, to determine which cache line to evict/touch respectively.

As would be expected, the number of ways in the cache is the single biggest contributor to the complexity, although not the size of the state space. Increasing the size of the cache increases the number of pointers, the number of cache lines and permutations in the tree structure. For example, increasing the number of ways in the cache from 4 to 8, using 10 memory blocks, increases the number of states from 1680 to 5760, assuming an implicit representation of tree structure. By contrast, the number of memory blocks only increases the number of potential cache lines in the state space, but does so at a very fast rate; for an 8 way cache, doubling the number of memory blocks from 10 to 20 increases the number of states from 5760 to 16124160. However, while a memory block can persist in a PLRU cache indefinitely, given a specific sequence of accesses, this can not be true for all memory blocks in the cache, and hence the number of memory blocks under consideration at a given point in the analysis is typically low, which means that the effective size of $|L|$ is low.

Of these items, the pointers and tree structure control the discrepancy between physical and logical states of the cache. As previously mentioned, flipping subtrees such that all pointers point in the same direction can be used to assign a logical cache state to a physical cache state. The information lost in this transformation is the distinction between different physical states which have the same logical behaviour, which clearly has zero value with respect to the analysis. However, by discarding this information, each pointer has a specified orientation, and this reduces the size of the state space by a factor of 2^{N-1}; this still leaves a state space of size $\frac{N!}{(|L|-N)!}$ (due to the implicit representation of the tree structure), which is still too large for efficient analysis.

Using these types of information, it is possible to evaluate their relevance with respect to the PLRU cache algorithm. It is possible to divide cache analysis into three steps: a classification step, which determines if an access is a hit, miss or uncertain, an evict step that determines possible successor states in the event of a miss, and a touch step that sets the pointers in the cache appropriately, and updates the tree structure. Each of these steps uses information, and hence can be affected by uncertain information. Classification uses the contents of the cache; in the case that this

	Step of analysis		
	Classify	Evict	Touch
Pointers		Used	Overwritten
Cache Lines	Used	Overwritten	
Tree Structure	Used	Used/Overwritten	Used/Overwritten

Table 4.1: Information usage in cache analysis

is uncertain, classification will be unable to determine cache hits or misses. Similarly, Eviction is degraded if a pointer is unknown, as in this case both possibilities an unknown pointer represents must be considered, and hence multiple successor states will be produced. However, it is also important to note that these steps also modify the cache state, and hence can remove uncertain information as well. An eviction operation can potentially replace an unknown memory block, while a touch operation can overwrite unknown pointers. By combining this overall description of the analysis with the description of types of information present in a PLRU cache state, as in Table 4.1, it is possible to determine the value of information for each step.

As can be seen, the tree structure is critical to all steps of the analysis, and hence it can be inferred that performing lossy compression on the tree structure could cause more problems than on other types of information. This can be observed in previous techniques; In Heckman et al's [56] technique of approximating the PLRU cache as a LRU cache, the tree structure is only preserved in the case of caches of size 2, when an PLRU cache behaves exactly as an LRU cache. For caches of a higher size, sequences of memory accesses greater than $log_2(k) + 1$ in size result in all accesses being classed as misses. For cache sizes above this, the tree structure is not preserved and the analysis suffers. In Grund and Reineke's analysis [54] it can be shown that the tree structure is preserved on caches of up to size 4, but that parts of the tree structure may be lost in caches of a higher size. This also results in there being a constant for which a sequence of accesses of greater length will result in all accesses being classed as misses; Grund and Reineke suggest this to be $2(log_2(k) - 1)$ based on empirical evidence obtained from experiments.

The cache lines of a cache state are used to determine if a cache access would be a hit or a miss; this in turn determines whether or not an eviction should occur. Pointers are used to determine what element of the cache should be evicted, if an eviction is necessary. It can therefore be inferred that the information presented in the cache lines is more valuable than the information in the pointers, because for each memory access it is necessary to classify the memory access, whereas evictions may not happen based on the result of the classification. Further, in actual use cache hit rates are engineered to be high, as the penalty for a high number of misses is severe; this is illustrated by Cantin and Hill [26], who show that for the SPEC CPU 2000 benchmark the miss rates for LRU associative caches typically range between 10% and less than 2%, depending on the size of the cache. Assuming that this represents acceptable performance for a cache, it would follow that for every evict operation that occurs, between 10 and 50 classification operations would occur. This leads to the conclusion that information used during classification (the contents of the cache lines) is much more valuable than information used during eviction (the contents of the pointers). In the case that this does not hold, and evictions happen frequently, then discarding pointers will cause pessimism; however, this pessimism will only occur when the cache is largely ineffective (due to a high miss frequency). Hence there is an argument that the use of the cache is inappropriate, and would increase WCET, due to the latency the cache introduces. Therefore, in the case where discarding pointers introduces significant pessimism, there is an argument that the use of the cache in inappropriate.

It is also important to consider that any operation will result in some information being overwritten; if uncertain information introduced by lossy compression is overwritten, then the lossy compression will not have an effect on the precision of the analysis. Hence the likely frequency of overwriting data is a consideration in determining which data should be discarded in compression. As a touch operation will be performed in each access, touch operations occur with the same frequency as classification operations, which is expected to be much greater than the eviction operation. The logical conclusion of this is that as the pointers of the cache are more frequently overwritten than the cache lines or tree structure, the impact of a pointer containing unknown information will be less than that of cache lines or tree structure.

Finally, it is necessary to consider the impact of uncertain information. As previously discussed, uncertain information may cause additional states to be considered, thus reducing the precision of the analysis. Any single

	Usage freq.	Overwrite freq.	Maximum Uncertainty impact
Pointers	Low	High	k on evict
Cache Lines	High	Low	2 ($2k$)
	(can trigger pointer and tree structure uncertainty)		
Tree Structure	V.High	High	k on touch

Table 4.2: Properties of Information in a PLRU cache

piece of uncertain information will not reduce the precision of the analysis however, as every decision in the analysis is binary, and hence a single piece of uncertain information corresponds exactly to the two possibilities of that binary decision. Hence, in the case of a single piece of uncertain information resulting from a merge, the analysis will have to consider two successor states. In the case of multiple pieces of uncertain information however, more successor states will result. Therefore, to gauge the relative impact of uncertain pieces of information it is necessary to examine how multiple pieces of uncertain information interact to produce states which could not have occurred in a precise analysis.

In the case of multiple uncertain pointers, each uncertain pointer encountered during an eviction results in two successor states. In the worst case, for a cache with N-ways, where every pointer in the cache is uncertain this will result in N possible successor states, as the analysis would have to consider the possibility that any cache line could be evicted. For the tree structure, uncertain information results in a cache line having an unknown position within the cache tree. In the worst case (i.e. the cache line is known to be in the cache, but its' position is not known), this results in N possible successor states for the touch operation, as the analysis would have to consider that the cache line being touch could reside at any location in the tree structure. However, this is mutually exclusive with the additional states created by an uncertain eviction, as eviction will select an exact location within the cache. Finally, in cache lines uncertain information can cause a classification to conclude that an access to a memory block cannot be classified, resulting in two successor states - one to consider a hit and one to consider a miss. However, this is not exclusive with either of the above behaviours; uncertain pointers will be triggered in the miss scenario, whereas uncertain tree structure will be triggered in the hit scenario.

Summarising this information (Table 4.2) it can be concluded that pointers are a useful target for lossy compression. The tree structure is not an appropriate target because it is used in every operation and uncertainty in the tree structure results in a large number of successor states. Cache lines are argued to be unsuitable as they are used frequently, overwritten infrequently, and an uncertain cache line can potentially trigger uncertain pointers and tree structure to generate additional successor states. Whilst the potentially large number of successor states might appear to make pointers undesirable, their expected low usage and high overwrite frequency combined mean that in typical usage it would not be expected that this would cause a problem.

4.1.2 Experimental Verification

Having constructed an overall argument that cache pointers are the least useful piece of information within a cache, this hypothesis should be tested by experiment. These experiments should aim to discard precisely one form of information from the cache states modelled, to determine the effect that this has on a simple benchmark. The benchmark picked is a simple looping test, as used by Grund and Reineke [54], as such a looping test stresses the cache algorithm. Further, as a 4-way cache already has a complete Must analysis in the form of Grund and Reineke's PLRU-plz algorithm, an 8-way cache set will be used for the benchmark.

For a chosen form of information to be considered amenable to discarding, discarding that information would have relatively little impact on accuracy of the analysis. In the case of a Must analysis, this would be represented as the number of hits that the analysis finds. Conversely, it would be expected that if more critical information were discarded, then the effect on the number of hits found by the analysis would be significant. In the case of a May analysis, the number of misses found represents the accuracy of the analysis.

The test strategies for discarding information are as follows:

- **S1: Cache Lines**: Pick a cache line; whenever a memory address could reside in that cache line, discard this information.
- **S2: Tree Structure**: Pick a pair of cache lines; consider any memory address that could reside in one cache line as if it could reside in the other cache line as well.

- **S3: Pointers**: Pick a pointer in the cache state; discard the value of that pointer.

It would be expected that these "blanket" discard approaches would likely yield poor analyses, as they discard information regardless of whether or not such information would yield compression. However, the blanket discard approach is able to determine the impact of discarding a certain piece of information. To determine the degree of inaccuracy, it is necessary to compare against the accurate results, specifically those derived by the Collecting Semantics [37], implemented as a simple state exploration approach. The results from this experiment are summarised in Figure 4.3, which shows the $loop(n)$ benchmark, a simple loop of memory blocks $1..n$ repeated 16 times. In this case, the only negative result is NC, or not classified, as this represents a failure of the analysis to reach a conclusion. If the hypothesis that discarding pointers loses the least information is true, then S3 should yield the best results.

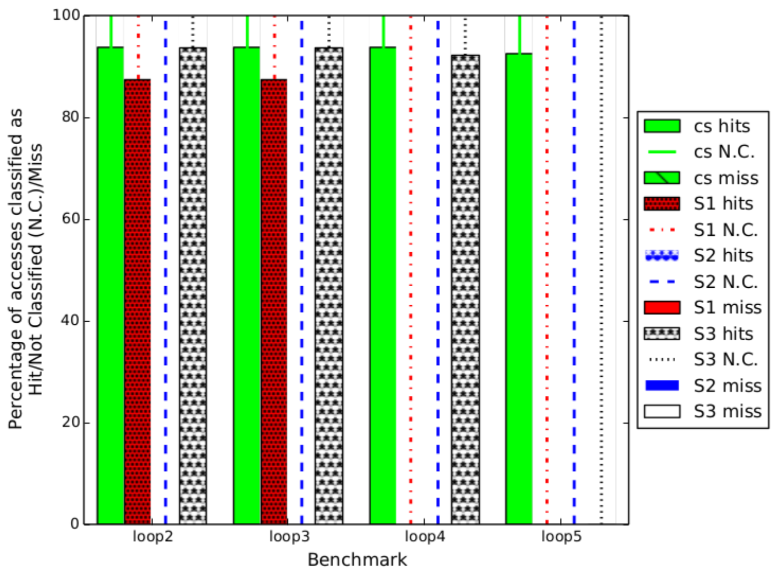


Figure 4.3: Results from the benchmarks, comparing the methods $S1$, $S2$ and $S3$ to the Collecting Semantics cs

As expected, none of these approaches is a workable analysis. Whilst analysis time and memory usage are not presented here, none of these strategies performs better than using the accurate Collecting Semantics. However, these results confirm the hypothesis stated in the previous section: that discarding information from cache lines or tree structure is much less preferable than discarding information from pointers, given that when pointer information is discarded longer loops can be analysed. Also of note is that tree structure is by far the worst type of information to discard - if tree structure is discarded, not even a loop of 2 memory locations can be analysed.

Having verified the hypothesis that pointers are the best form of information to discard, it is necessary to construct the representation, merge operator and recovery strategy. This is accomplished in the next section.

4.1.3 Discarding the Least Valuable Information and Recovery

As has already been established, multiple cache states that exhibit the same behaviour, in that any sequence of accesses to memory blocks results in the same actions being taken, can be discarded (Figure 4.1), but doing just this does not result in sufficient compression. As pointers have been identified as the type of information contained within a cache state that has the least bearing on the outcome of the analysis, it is necessary to revisit how logical states are constructed. The goal of this is to determine a method for constructing logical states such that the information stored in pointers can easily be compressed, while preserving the cache lines and tree structure of the states.

The present method [37] finds logical states by using pointers to rearrange the tree structure so that all pointers point the same direction, and is referred to as *cache naming.* Cache naming is unsuitable as it becomes complex to check that two cache states have the same tree structure, as seen in Figure 4.2. Instead, an alternative method of transforming cache states is necessary, such that for cache states with the same cache lines it is easy to determine if they also have the same tree structure. This can be accomplished by using the cache lines to manipulate the tree structure, as opposed to the pointers, and will be referred to as the *cache signature.* For compactness, a cache state which has been adjusted so its name can be read off (i.e. all pointers are set to a predetermined direction) will be referred to as a *named state*, and similarly a cache state which has been adjusted so that the signature can be read off will be referred to as a *signature state.*

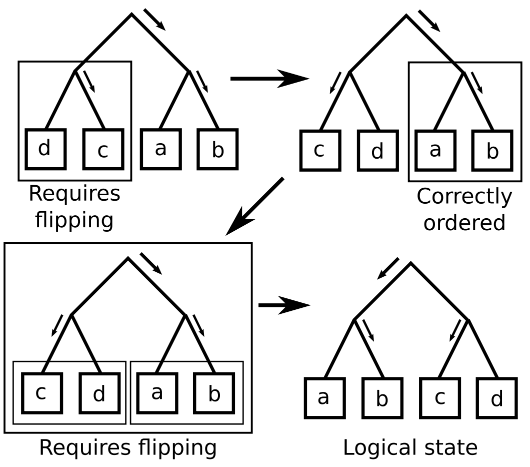


Figure 4.4: Illustrating how to find a cache signature, using alphabetical comparison, by moving from a physical state to an appropriate logical state without using pointers

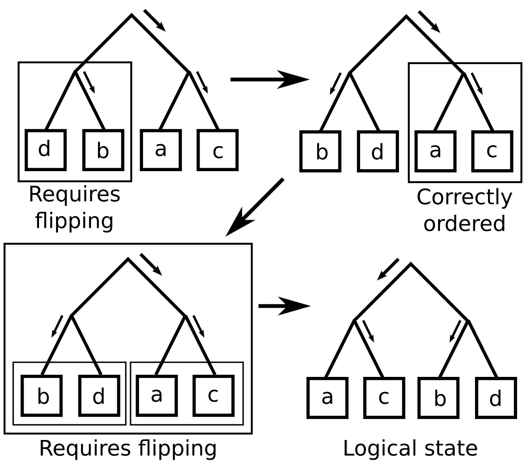


Figure 4.5: A second of finding a cache signature. As the tree structure differs from the first, the logical state reached is significantly different.

A method of computing the signature, using cache lines to manipulate the tree structure, is to recursively sort the cache tree such that for each node of the tree, the leftmost element of the left subtree is less than the leftmost element of the right subtree, using a given ordering (assuming that no memory block can be in the cache twice). With caches in this arrangement, as the arrangement given by cache naming which uses pointers, it is trivial to check if two caches containing the same cache lines have the same tree structure, by simply reading the elements of the cache in left-to-right order. As pointers are not used to compute this ordering, the only way for two cache states containing the same memory blocks to have a different ordering of their elements is for their tree structures to differ. Hence it accomplishes the goal of determining when two cache states have the same cache lines and tree structure, and thus enables pointer information to be analysed.

An example of this is given in Figures 4.4 and 4.5, using an alphabetic ordering on the memory blocks a, b, c, d contained in the cache states. In Figure 4.4, first the deepest subtrees are sorted, resulting in the left hand subtree being flipped but the right hand subtree remaining the same. Then the next deepest subtrees (in this case, the entire tree) is sorted in the same manner, by comparing c and a; as $a < c$ the subtrees swapped, resulting in the signature state. Figure 4.5 demonstrates the same sequence of actions, but on a cache with a different tree structure. The resulting logical cache state is significantly different, with the final ordering of elements changed; crucially however, when using this method the only way to obtain such a difference is through cache states with different contents or tree structure - the contents of the pointers only affects the pointers in the signature state, and not the positioning of memory blocks in the signature state.

An additional advantage of removing the impact of pointers on the positioning of elements in the signature states is that the signature states become more efficient to update than named states. This is because if no eviction/replacement is carried out, a signature state remains in its signature representation. By contrast, as the touch operation operation changes pointers, in a named state which relies on pointers for its ordering, then subtree flipping must occur. Hence, by removing the impact of pointers, the efficiency of analysis is increased.

The proposed method for lossy merging is to allow each pointer to occupy three states: left-pointing, right-pointing, and uncertain. When performing

Cache	For each set of N cache lines, number of		
Associativity	Physical Sates	Logical States	Compressed Signature States
2	2	2	1
4	192	12	3
8	5160960	40320	315
16	$\approx 10^{18}$	$\approx 10^{14}$	$\approx 10^{9}$
N	$2^{N-1}N!$	$N!$	$N!/2^{N-1}$

Table 4.3: Compression from going from physical states to logical states

merging, each cache state is first transformed into its logical representation by applying the sorting method outlined above. Then, when multiple cache states have the same cache content and tree structure, indicated by reading the contents of the logical cache state from left to right, these cache states are merged. Each pointer in the merged cache state is set to be left or right pointing if for all cache states being merged, the pointers in the same location are all left or right pointing. Otherwise, the pointer is set to be uncertain, as it must represent both values to be sound.

The size of the state space when applying this additional compression can be computed by first observing that pointers no longer contribute anything to the size of the state space, hence obtaining a compression factor of 2^{N-1}, where N is the number of cache ways. Then by observing that for each subtree in the cache tree structure, that subtree can only be arranged in a single configuration due to the conversion to a logical state, thus obtaining an additional compression factor of 2^{N-1}. Hence, applying these compression factors to the size of the exhaustive state space, the size of the state space is $\frac{N!}{2^{N-1}}$, multiplied by the number of combinations of memory blocks that can be in a cache state (${}^{k}C_{N}$ for k distinct memory blocks), giving a total size of $\frac{N!k!}{2^{N-1}N!(k-N)!}$. Table 4.3 illustrates the benefit of moving to this representation, as caches of size 8 become feasible to analyse. However, the caveat of this approach is that due to preserving the most complex element of the cache state, the tree structure, the size of the state space still grows at $O(N!)$, thus meaning that analysis of a 16 way PLRU cache remains infeasible.

Next it is necessary to identify a recovery strategy. As has been mentioned before, a PLRU cache access will overwrite data, giving a form of natural recovery. However, in the case that uncertain data is required, it is necessary to consider all possible states; to do otherwise would violate the requirement of soundness. Hence the recovery strategy is simply to consider all possible values of the uncertain data.

Having decided on the type of information that is of low value to cache analysis and how to discard this information, it is now possible to define an algorithm to implement this method. Before doing this, the notation used is given in the next section. Following this, first an informal outline of the technique when compared to Collecting Semantics is given in Section 4.3 , which is then followed by a formal definition.

4.2 Full Tree Analysis: Conventions and Notation

Before beginning the definitions of the algorithm, it is necessary to introduce the notation and conventions used in these definitions. The first, and most important piece of notation is as follows: $\perp$ is a symbol which extends the alphabet used to represent pointers, and represents an *unknown* pointer. As pointers have been evaluated to be the least important piece of information in a cache state, $\perp$ is necessary such that should the value of a pointer be discarded, $\perp$ can be used to represent this fact.

Next it follows to give the representation of PLRU tree. To aid in the definition of the algorithm, which will be recursive, such PLRU trees are specified recursively. Hence, a subtree of a PLRU cache is defined to be a triple $\langle \mathbf{l}, v, \mathbf{r} \rangle$ where:

- v is the value of the data stored at this point in the tree. For a tree of height 0, this is a set of possible cache line identifiers that may reside at this cache location. For other heights, this is a pointer which can contain the values $\{0, 1, \perp\}$. The value 0 refers to a left-pointing pointer and the value 1 refers to a right-pointing pointer.

- $\mathbf{l}$ and $\mathbf{r}$ are the left and right subtrees respectively. For a tree of height 0, these are empty (denoted as $\emptyset$). Otherwise, subtrees are represented as triples of this form.

And, assuming that all subtrees are symmetric the height of a subtree is trivially defined as:

$$height(\langle \mathbf{l}, v, \mathbf{r} \rangle) = \begin{cases} 0 \textbf{ if } \mathbf{l} = \emptyset \\ height(\mathbf{l}) + 1 \textbf{ otherwise} \end{cases} \tag{4.1}$$

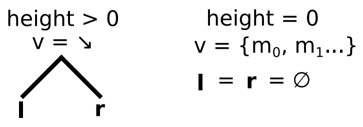


Figure 4.6: The subtree form a PLRU cache

Subtrees are illustrated in Figure 4.6, which shows subtrees for heights > 0 and 0.

For functions accepting a tree as an argument, the forms $f(\langle \mathbf{l}, v, \mathbf{r} \rangle)$ and $f(\mathbf{t})$ (where $\mathbf{t} = \langle \mathbf{l}, v, \mathbf{r} \rangle$ represents a subtree) are used interchangeably. In algorithms, tuple unpacking is used such that $\mathbf{l}, v, \mathbf{r} \Leftarrow \mathbf{t}$ assigns the left subtree, value, and right subtree of $\mathbf{t}$ to the variables $\mathbf{l}$, v and $\mathbf{r}$ respectively.

In a PLRU cache, a cache subtree of size N has a root node with height $ln_2(N)$. It always follows that subtrees of a cache of height h have height $h - 1$.

A subtree is said to contain the sets of memory locations contained by any of its subtrees. So for example with of a height 2 tree $\mathbf{t} = \langle \mathbf{l}, v, \mathbf{r} \rangle$, $\mathbf{l} = \langle \mathbf{l_l}, v_l, \mathbf{l_r} \rangle$, $\mathbf{r} = \langle \mathbf{r_l}, v_r, \mathbf{r_r} \rangle$ and all other subtrees are empty, $\mathbf{t}$ contains the v_l and v_r. While this is a slight abuse of notation, the $\in$ operator is used to denote the containing relationship described above, and hence the statement $\mathbf{v_l} \in \mathbf{t}$ is true. A similar abuse of notation is used to denote an item being a member of a tuple, i.e. $x \in \langle v_0..v_i \rangle \Rightarrow \exists n \in [0, 1] | v_n = x$.

It is assumed that an ordering is defined over all sets of cache line identifiers; the order is arbitrary, but must be fixed. An example ordering could be given by sorting the set of cache line identifiers and comparing the resulting lists.

Finally, a dashed variable refers to a modified form of the undashed variable (e.g. x' is a modified form of x). However, note that in some pseudocode some variables are modified in-place, which means that the dashed variable convention is not appropriate.

Having defined the conventions and notation used, it follows to give the informal definition of the Full Tree analysis algorithm.

4.3 Full Tree Analysis: Informal Definition

Having decided that pointers are to be discarded without touching the contents of the cache, it now follows to outline how this can be accomplished. The key idea is to merge cache states which have the same contents and tree structure, but differ in their pointers; when pointers that differ are merged, they are replaced by $\perp$, which represents an unknown pointer which could hold any possible value. In principle, this can yield a compression factor of 2^{N-1} for a cache with N ways, by ensuring that all states which only differ by pointers are represented in a single state.

Full Tree analysis is presented as a series of modifications to the Collecting Semantics of a PLRU cache [36, 54]. The Collecting Semantics describes the logical behaviour of the PLRU cache, and provides a useful starting point as it is correct and sound, although computationally expensive. This section will introduce the operations which need to be modified in order to implement Full Tree analysis. In addition, the Functions Sort and Next give a pseudocode overview of the approach.

The first step is to define a new way of determining how to represent the behaviour of a cache. The previous method of setting all pointers to a single direction (Figure 4.1) does not have a way to easily determine if the cache contents and tree structure are the same between two states. Hence, a new naming method, the cache *signature* is required. The cache signature utilises subtree flipping to ensure that the left-most element of the left subtree is less than the left-most element of the right subtree, for all subtrees in the cache tree. This can be accomplished efficiently be starting from the smallest subtrees; the technique is illustrated in Figure 4.4, and a pseudocode implementation of this *sort* function is given in Function Sort. Once this property is established, the memory blocks in the cache are ordered by the tree structure, and not the pointers. Hence the signature can be defined as the tuple of cache lines ordered from left to right in the *sort*ed state.

With the method to calculate a cache's logical state changed from cache naming to the *signature*, merging becomes trivial. As the signature of a cache state represents both the cache contents and the tree structure, the two items which merging must preserve, two states can be merged if and only if they share the same signature. Hence, for two states which have

```
Function Sort($\mathbf{s}$)
    if $height(\mathbf{s}) > 0$ then
        $\mathbf{l}, v, \mathbf{r} \leftarrow \mathbf{s}$
        $\mathbf{l}' \Leftarrow sort(l)$
        $\mathbf{r}' \Leftarrow sort(r)$
        if leftmost element of $\mathbf{l}'$ < leftmost element of $\mathbf{r}'$ then
            if $v = 0$ then
                $v' \Leftarrow 1$
            else if $v = 1$ then
                $v' \Leftarrow 0$
            else
                $v' \Leftarrow \perp$
            end
            return $\mathbf{s}' \Leftarrow (\mathbf{r}', v', \mathbf{l}')$
        end
        else
            return $\mathbf{s}$
        end
    else
        return $\mathbf{s}$
    end
end
```

Function Sort($\mathbf{s}$), which converts a cache state according to the cache signature rule

```
Function Signature($\mathbf{s}$)
    if $height(\mathbf{s}) = 0$ then
        $\mathbf{l}, v, \mathbf{r} \leftarrow \mathbf{s}$
        return $[v]$
    end
    else
        $\mathbf{s}' \leftarrow sort(\mathbf{s})$
        $\mathbf{l}, v, \mathbf{r} \leftarrow \mathbf{s}'$
        return Signature($\mathbf{l}$) + Signature($\mathbf{r}$)
    end
end
```

Function Signature($\mathbf{s}$), which returns the signature of cache state $\mathbf{s}$

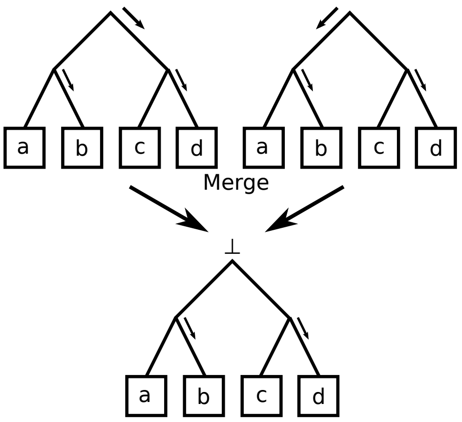


Figure 4.7: Merging two cache states which have been sorted

been *sort*ed, merging is a simple matter of examining the pointers in the states and setting the pointers to the unknown value, $\perp$, if they conflict, and otherwise keeping the original value (Function Next Line 25). This is illustrated in Figure 4.7.

Given a set of cache states, merging the contents of the set is accomplished by first applying the sorting operation to each element of the set. Next, this set is split into subsets such that all elements of each subset share the same signature, and finally each subset is merged to a single state. As merging does not change the memory blocks within the cache states, classification works in the manner that it normally would (i.e. if a memory block is present in all possible cache states, the access is a hit, if it is not present in any it is a miss and if it is present in some states but not others it is not classified). Similarly, as the touch operation does not require the use of pointers, it too works in the normal manner (i.e. setting all pointers on the path to a touched memory block m to point away from m). However, the evict operation does require the use of pointers, and hence must be modified appropriately. Whenever an eviction operation encounters an unknown pointer, it must consider both possibilities that the pointer could represent, as shown in Figure 4.8. This approach is given in Function Next at Lines 12-15.

It is of note that in the worst case, the behaviour of eviction will result in substantial pessimism, as each eviction can potentially result in 2^{N-1} successor states (where N is number of cache ways). However, the risk of

```
1  Function Next(S, m)
2      if $\forall s \in S, m \in s$ then
3          $classification \Leftarrow$ hit
4      else if $\forall s \in S, m \notin s$ then
5          $classification \Leftarrow$ miss
6      else
7          $classification \Leftarrow$ not classified
8      end
9      $T \Leftarrow \emptyset$ (used as an intermediate set where all eviction operations
       have been carried out)
10     for $\mathbf{s} \in S$ do
11         if $m \notin \mathbf{s}$ then
12             $C \Leftarrow$ the set of concrete states that are represented by $\mathbf{s}$
13             for $\mathbf{c} \in C$ do
14                 $\mathbf{c}' \Leftarrow \mathbf{c}$ with the cache line indicated by the pointers
                   replaced with $m$
15                 $T \Leftarrow T \cup \{\mathbf{c}'\}$
16             end
17         else
18             $T \Leftarrow T \cup \{\mathbf{s}\}$
19         end
20     end
21     $R = \emptyset$ (used to construct the final set of successor states with
       merged cache states)
22     for $\mathbf{t} \in T$ do
23         $\mathbf{t}' \Leftarrow \mathbf{t}$ with all pointers on path to $m$ set to point away from $m$
24         if $\exists \mathbf{r} \in R$ where $signature(\mathbf{r}) = signature(\mathbf{t}')$ then
25             $\mathbf{m} \Leftarrow signature(\mathbf{t}')$ with all conflicting pointers with
               $signature(\mathbf{r})$ set to $\perp$
26             $R \Leftarrow (R \cup \{\mathbf{m}\}) \setminus \{\mathbf{r}\}$
27         end
28         $R \Leftarrow R \cup (signature(\mathbf{t}))$
29     end
30     return $classification, R$
31 end
```

Function Next(S, m) which computes the classification and successor states from applying memory access m to the set of abstract states S

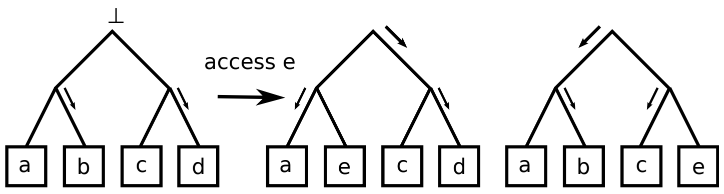


Figure 4.8: Evicting with an unknown pointer

observing this behaviour is low as there is likely to be significant overlap in pointers between cache states with the same signature and that the evict operation is assumed to have a much lower frequency than the touch operation, which decreases the number of unknown cache pointers. As seen in Function Next, other than the changes involving cache signatures, merging and eviction, the algorithm proceeds in much the same way as the Collecting Semantics would, in that it explores the states presented to it and assigns each memory access a classification.

Taking into account these changes, the complete algorithm is applied to a program in the same way that a Collecting Semantics approach would be. Specifically, an initial set of cache states is supplied (which may correspond to the unknown cache state). Then, for each memory access the access is first classified as either a hit, miss or no classification (NC), based on the current set of cache states under consideration. After this classification, the set of cache states under consideration is updated by applying the evict and touch operations as appropriate. Instead of computing which cache states have the same behaviour by cache naming, the merging step outlined above is then applied, which reduces the number of cache states under consideration at the next step. To handle a branch in the program, each branch is considered separately, and the cache state after the paths of the branch have merged back together is defined to contain all states that are possible at the end of each branch.

Having outlined the functions which need to be specified, it remains to define a formal definition and prove its correctness.

4.4 Full Tree Analysis: Formal Definition

This section gives a formal definition of the Full Tree Analysis algorithm. Using the conventions from Section 4.2, this definition is defined such that it is possible to construct a proof of correctness, by examining the portions which differ from the Collecting Semantics, a technique which is known to be correct.

4.4.1 Concretisation

In order to demonstrate that the formalisation is relevant to the concrete PLRU cache, it is necessary to define a concretisation function which provides a map between the abstract states used in the formalisation and the concrete states that they represent. The concretisation function will map abstract states onto logical behaviours, as per the convention on naming cache states used by Collecting Semantics [36]. After this, the proof of the correctness of Collecting Semantics can be used for the final step of the correctness of this proof. For an abstract cache state to correspond to exactly one logical cache behaviour, the following conditions must hold on all subtrees $\mathbf{s} = \langle \mathbf{l}, v, \mathbf{r} \rangle$:

- For all subtrees of height greater than 0, v represents a cache pointer, and so v must not be $\perp$
- For all subtrees of height 0, which represent a cache line, v represents the set of possible memory blocks and so must have a single value i.e. $|v| = 1$
- For all pairs of subtrees $\mathbf{s_0}, \mathbf{s_1} \in \mathbf{s}$ where $\mathit{height}(\mathbf{s_0}) = \mathit{height}(\mathbf{s_1})$, for all memory blocks m, $m \in \mathbf{s_0} \Rightarrow m \notin \mathbf{s_1}$.

When mapping an abstract state to a set of states that are present in the Collecting Semantics, it is necessary to enforce all of these conditions. Hence, to enforce the first condition, it is necessary to define a function which maps an abstract pointer to the set of pointers it represents while taking into account the possibility of an unknown pointer. The function is simply defined as follows:

$$pointerValues(p) = \begin{cases} \{0,1\} \ \textbf{if} \ p = \perp \\ \{p\} \ \textbf{otherwise} \end{cases} \tag{4.2}$$

Whilst the second condition can be enforced by simply splitting the set v into its individual elements, enforcing the condition that a cache line identifier can only be in a single subtree is more tricky. Hence the recursively defined concretisation function c must explicitly enforce this condition by removing any combination of concrete subtrees which have a memory location in common. Therefore it is necessary to define a *contents* function which returns the memory locations that are present in a cache subtree, as follows:

$$contents(\langle \mathbf{l}, v, \mathbf{r} \rangle) = \begin{cases} \langle v \rangle & \textbf{if} \ height(\langle \mathbf{l}, v, \mathbf{r} \rangle) = 0 \\ \langle a_0 ... a_m \rangle & \textbf{where} \ \langle a_0 ... a_n \rangle = contents(\mathbf{l}) \\ & \langle a_{n+1} ... a_m \rangle = contents(\mathbf{r}) \ \textbf{otherwise} \end{cases} \tag{4.3}$$

Using this function, the concretisation function c can be defined as follows.

$$c(\langle \mathbf{l}, v, \mathbf{r} \rangle) = \begin{cases} \{\langle \mathbf{l}, v', \mathbf{r} \rangle \big| \ v' \in v\} \ \textbf{if} \ height(\langle \mathbf{l}, v, \mathbf{r} \rangle) = 0 \\ \{\langle \mathbf{l}', v', \mathbf{r}' \rangle \big| \ v' \in pointerValues(v), \mathbf{l}' \in c(\mathbf{l}), \mathbf{r}' \in c(\mathbf{r}), \\ m \in contents(\mathbf{l}') \Leftrightarrow m \notin contents(\mathbf{r}') \\ \forall m \in contents(\langle \mathbf{l}, v, \mathbf{r} \rangle)\} \ \textbf{otherwise} \end{cases} \tag{4.4}$$

The function c maps an abstract state to a set of states which have all the properties defined previously. In particular, note that v takes different values depending on the height of the subtree. In case 1, where the height of the subtree is 0, v is a set of potential memory blocks. In case 2, v represents the value of a pointer. Hence each of these states represents a single logical cache state, and by the correctness of Collecting Semantics [36] a set of concrete cache states, and therefore the formalisation is relevant.

As it is expected that the algorithm will have to deal with multiple cache states, for proving soundness it is useful to define *cSet*, the concretisation function applied to a set of cache states S, as follows:

$$cSet(S) = \bigcup\{c(\mathbf{t})|\mathbf{t} \in S\} \tag{4.5}$$

4.4.2 Cache Operations

The first set of functions to be defined concern the normal operation of the cache. As previously stated, a typical cache algorithm can be broken up into three operations:

1. *Classification*: Determining if a cache access is a hit or a miss
2. *Eviction and Replacement*: Determining a cache line to evict and replace the contents with a new cache line. For a PLRU cache, this is implemented by following the pointers in the cache tree.
3. *Touch*: Determining the actions needed to prolong the life of an element in the cache. For a PLRU cache, this is implemented by setting each pointer on the path to the selected element away from the element.

However, in addition to the normal definition for a PLRU cache, these operations must be defined to work on the abstract representation, where both cache line index location and pointer values may be uncertain. The next sections define the relevant formal functions.

Classification

Using the concretisation function, it is possible to construct a classification function cs. The classification function simply determines if a memory access is a hit, miss or unclassifiable, and does this by inspecting all concrete states that the cache may occupy. Hence the classification function is defined as follows:

$$cs(\mathbf{t}, m) = \begin{cases} hit \textbf{ if } \forall \mathbf{t}' \in c(\mathbf{t}), m \in \mathbf{t}' \\ miss \textbf{ if } \forall \mathbf{t}' \in c(\mathbf{t}), m \notin \mathbf{t}' \\ notclassified \textbf{ otherwise} \end{cases} \tag{4.6}$$

Evict And Replace

The eviction and replacement operator takes a cache state and evicts the cache line as indicated by the pointers. In the case of a pointer which is unknown, the eviction operator must take into account both possibilities that arise. This may result in many successor cache states from a single cache state; however, this is not a problem as later compression is used to reduce the number of states. Further, the eviction operation must also take into account that it might not be known if a cache line is in a cache state, and hence eviction must update the cache appropriately; this can be accomplished by exhausting all possible locations for the cache line and creating appropriate copies of the state where each of these assumptions is true. To do this, it is necessary to define the *purge* function, which removes all references to a cache line from a subtree. Note that this also handles the case that the location of a memory block is unknown (e.g. from an initial unknown state), where the cache state will not correspond to a single behaviour and hence sets of potential memory blocks have a size greater than 1.

$$purge(\langle \mathbf{l}, v, \mathbf{r} \rangle, m) = \begin{cases} \langle \mathbf{l}, v \setminus \{m\}, \mathbf{r} \rangle \ \textbf{if } height(\langle \mathbf{l}, v, \mathbf{r} \rangle) = 0 \\ \langle purge(\mathbf{l}, m), v, purge(\mathbf{r}, m) \rangle \ \textbf{otherwise} \end{cases} \tag{4.7}$$

Note that the purge function deletes information, and is only usable if all possible locations of m are considered.

Given the requirements, it is possible to define the insertion function *insert*, which takes an abstract state and maps it to the set of states given by inserting m at the location given by the pointers.

$$insert(\langle \mathbf{l}, v, \mathbf{r} \rangle, m) = \begin{cases} \{\langle \mathbf{l}, \{m\}, \mathbf{r} \rangle\} \ \textbf{if } height(\langle \mathbf{l}, v, \mathbf{r} \rangle) = 0 \\ \{\langle insert(\mathbf{l}, m), v, \mathbf{r} \rangle\} \ \textbf{if } v = 0 \\ \{\langle \mathbf{l}, v, insert(\mathbf{r}, m) \rangle\} \ \textbf{if } v = 1 \\ insert(\langle \mathbf{l}, 0, \mathbf{r} \rangle, m) \cup insert(\langle \mathbf{l}, 1, \mathbf{r} \rangle, m) \ \textbf{if } v = \perp \end{cases} \tag{4.8}$$

As the *insert* function does not check on the presence of m within the cache state, it can only be applied in the case of a guaranteed miss. Othewise, an insertion of memory block m is not guaranteed to result in a definite location within the cache. Hence the *evict* function is defined as follows, which also provides the necessary logic to apply evictions as necessary.

$$evict(\mathbf{s}, m) = \begin{cases} \{\mathbf{s}\} \textbf{ if } cs(\mathbf{s}) = \text{ hit} \\ insert(\mathbf{s}, m) \textbf{ if } cs(\mathbf{s}) = \text{ miss} \\ \{\mathbf{s}\} \cup insert(purge(\mathbf{s}, m), m) \textbf{ otherwise} \end{cases} \tag{4.9}$$

The cases of the function ensure that the eviction is only applied in the cases when it is necessary; in the first case this ensures that eviction is not applied if the access is guaranteed to be a hit. In the second case, a guaranteed miss, *insert* is used to get the appropriate set of cache states where m has been inserted to the locations the pointers indicate. In the third case, where the access is not classified, the result is the union of both the previous cases, but *purge* is used to construct a state where the assumption of a miss is true.

Touch

The touch operation sets all pointers to be pointing away from the path to a given cache line index. The difficulty in this function is that the abstract representation may not be able to give a precise location for a cache line index. If at a given stage, it is not certain which subtree a cache line index, m, is contained in, it is necessary to consider both possibilities. This is accomplished in a similar manner to the *evict* function, by utilising the *purge* function to assert where the possible location could be. Hence, it is possible to define the *touch* function, which maps an abstract cache state to a set of a successor states as follows, assuming that the touched cache line index m is indeed a member of the cache, and utilising the classification function cs:

$$
touch(\langle \mathbf{l}, v, \mathbf{r} \rangle, m) = \begin{cases} \{\langle \mathbf{l}, v, \mathbf{r} \rangle\} \textbf{ if } height(\langle \mathbf{l}, v, \mathbf{r} \rangle) = 0 \\ \{\langle touch(\mathbf{l}, m), 1, \mathbf{r} \rangle\} \textbf{ if } cs(\mathbf{l}, m) = hit \\ \{\langle \mathbf{l}, 0, touch(\mathbf{r}, m) \rangle\} \textbf{ if } cs(\mathbf{r}, m) = hit \\ \{\langle touch(\mathbf{l}, m), 1, purge(\mathbf{r}, m) \rangle, \\ \langle purge(\mathbf{l}, m), 0, touch(\mathbf{r}, m) \rangle\ \} \textbf{ otherwise} \end{cases} \tag{4.10}
$$

Note that the *touch* function exploits the fact that for two cache subtrees S_0, S_1 of the same height, which belong to the same cache state, $cs(S_0, m) = hit \Rightarrow cs(S_1, m) = miss$ as if m is definitely in S_0 it cannot be in S_1 as well. Hence the final case only occurs when $cs(\mathbf{l}, m) = cs(\mathbf{r}, m) = notclassified$. Also note that in cache operation, whenever *touch* is called it is guaranteed that the cache line m will be in the cache at some location, even if that location is not precisely known.

4.4.3 Compression

Having defined how normal operations are performed on the abstract cache state, it remains to show how compression is accomplished. Compression is implemented by two steps:

1. *Signature and Sorting*: Cache states are converted to a standardised form by a sorting procedure; this form exposes the cache signature.
2. *Merging*: Cache states with the same signature are merged, reducing the number of cache states being considered.

The following sections describe these operations.

Signature and Sorting

With the operation of the cache defined, it is necessary to define the compression. The first step in this is to find a cache state's signature by using sorting. As shown before in Figure 4.4, this operation involves sorting the subtrees of the cache with height greater than zero from the bottom up. As the same operation is applied to each subtree, it is possible to define the function recursively, broken down into three functions. The first function, *inv*, simply encapsulates how to invert a pointer whilst taking into account that the precise value of the pointer may not be known.

$$inv(p) = \begin{cases} 0 \textbf{ if } p = 1 \\ 1 \textbf{ if } p = 0 \\ \perp \textbf{ otherwise} \end{cases} \tag{4.11}$$

An important property of the contents function is that for subtrees with the same height, the length of the tuple returned by contents will be the same. Hence, with the previously mentioned assumption of an ordering between sets of possible cache line identifiers, it is possible to define a comparison between tuples returned by contents for subtrees of the same height as follows:

$$lte(\langle \mathbf{a}_0...\mathbf{a}_n\rangle, \langle \mathbf{d}_0...\mathbf{d}_n\rangle) \begin{cases} \text{True} & \textbf{if } \mathbf{a}_0 = \mathbf{d}_0 \text{ and } n = 0 \\ \text{True} & \textbf{if } \mathbf{a}_0 < \mathbf{d}_0 \\ \text{False} & \textbf{if } \mathbf{a}_0 > \mathbf{d}_0 \\ lte(\langle \mathbf{a}_1...\mathbf{a}_n\rangle, \langle \mathbf{d}_1...\mathbf{d}_n\rangle) & \textbf{otherwise} \end{cases} \tag{4.12}$$

With these operations defined, and using the *contents* (4.3) function from before, it is now possible to define the sort operation. This is a formal definition of the sort function defined in the pseudocode Function Sort.

$$sort(\langle \mathbf{l}, v, \mathbf{r}\rangle) = \begin{cases} \langle \mathbf{l}, v, \mathbf{r}\rangle \textbf{ if } height(\langle \mathbf{l}, v, \mathbf{r}\rangle) = 0 \\ \langle sort(\mathbf{l}), v, sort(\mathbf{r})\rangle \textbf{ if} \\ \quad lte(contents(sort(\mathbf{l}), contents(sort(\mathbf{r})))) \\ \langle sort(\mathbf{r}), inv(v), sort(\mathbf{l})\rangle \textbf{ otherwise} \end{cases} \tag{4.13}$$

The sort function is broken up into three cases. The first case is the recursive base case; if the height of the cache is zero, then $\mathbf{l} = \mathbf{r} = \emptyset$ and no sorting is necessary; hence the subtree is returned unmodified. The second case checks to see if, when subtrees are sorted, the left subtree does indeed belong on the left in the sorted tree. If this is the case, the only action required is to sort both subtrees. The final case deals with the remaining possibility: that in the sorted tree, the current left subtree belongs on the right. Hence the subtrees are flipped around and the pointer inverted.

With the sort function defined, the *signature* function (previously defined in the pseudocode Function Signature) can be defined trivially as the sets of potential cache line indexes read from left to right on the sorted cache state. Hence the signature function is simply:

$$signature(\mathbf{t}) = contents(sort(\mathbf{t})) \tag{4.14}$$

4.4.4 Merge

With the signature and cache sort operations defined, all that remains is to define a *merge* function. The *merge* function considers the current set of states and merges all which have the same signature. This implies that they contain the same memory blocks, tree structure. In order to simplify merging, it is assumed that all cache states to be merged are sorted by the *sort* function, an assumption that is enforced by the algorithm in (4.18) and (4.19).

The first function to define is *mergeStates*, a function to merge two states which have the same signature and are sorted. If both subtrees have height 0, then by virtue of their signatures matching they are identical and no action is necessary. If the height is not zero, then the pointers are compared; if they match then the pointer is kept, otherwise it is set to be unknown. After the pointers are merged, the left and right subtrees of the cache states are merged as appropriate. As the two cache trees are initially of the same height, all subtrees merged in this manner will be of the same height. Hence a case where the heights of the subtrees are unequal cannot happen.

$$mergeStates(\begin{matrix}\langle \mathbf{l_1}, v_1, \mathbf{r_1}\rangle, \\ \langle \mathbf{l_2}, v_2, \mathbf{r_2}\rangle\end{matrix}) = \begin{cases} \langle \mathbf{l_1}, v_1, \mathbf{r_1}\rangle \textbf{ if } height(\langle \mathbf{l_1}, v_1, \mathbf{r_1}\rangle) = 0 \\ \langle mergeStates(\mathbf{l_1}, \mathbf{l_2}), v_1, \\ mergeStates(\mathbf{r_1}, \mathbf{r_2})\rangle \textbf{ if } v_1 = v_2 \\ \langle mergeStates(\mathbf{l_1}, \mathbf{l_2}), \perp, \\ mergeStates(\mathbf{r_1}, \mathbf{r_2})\rangle \textbf{ otherwise} \end{cases} \tag{4.15}$$

Next it is necessary to define how this might be applied to a set of states which all share the same signature and are all sorted. Utilising the notation $\mathbf{t} = \langle \mathbf{l}, v, \mathbf{r} \rangle$, it is sufficient to recursively apply the *mergeStates* function as follows:

$$mergeStatesSet(\{\mathbf{t_0}...\mathbf{t_n}\}) = \begin{cases} \mathbf{t_0} \textbf{ if } n = 0 \\ mergeStates(\mathbf{t_0}, \\ \quad mergeStatesSet(\{\mathbf{t_1}..\mathbf{t_n}\})) \textbf{ otherwise} \end{cases} \tag{4.16}$$

To extend this to sets of cache states which do not share the same signature, it is necessary to define functions which group the cache states by signature. Hence the function *splitBySignature* is defined which takes a set of cache states A, identifies each unique signature in A and produces sets containing each cache state with a given signature, sorting all cache states in these sets as required by the *mergeStatesSet* function.

$$signatures(A) = \{signature(\mathbf{t}) | \mathbf{t} \in A\} \tag{4.17}$$

$$splitBySignature(A) = \{\{sort(\mathbf{t}) | \mathbf{t} \in A, signature(\mathbf{t}) = s\} | s \in signatures(A)\} \tag{4.18}$$

Finally, the *merge* function can be defined as the union of merging each set of states which have the same cache signature, with the formal definition as follows:

$$merge(A) = \bigcup \{mergeStatesSet(B) | B \in splitBySignature(A)\} \tag{4.19}$$

4.4.5 The Complete Algorithm

To complete the algorithm, it is necessary to define a *next* function, which updates the set of abstract cache states with a memory access and then compresses the result. To do this it is necessary to extend the *evict* and *touch* functions such that they can be applied to sets of cache states. These extended functions can be defined as follows:

$$evictSet(T,m) = \bigcup\{evict(\mathbf{t},m) \big| \mathbf{t} \in T\} \tag{4.20}$$

$$touchSet(T,m) = \bigcup\{touch(\mathbf{t},m) \big| \mathbf{t} \in T\} \tag{4.21}$$

As *evict* has no effect on cache states which are a hit, and because *touch* will only be applied to cache states where m is guaranteed to be a hit due to the operation of the cache, it is simple to define the *next* function as the functional composition of *evictSet*, *touchSet* and *merge*. Hence, *next* is defined as:

$$next(T,m) = merge(touchSet(evictSet(T,m),m)) \tag{4.22}$$

Finally, it is necessary to define the application to a trace of accesses to memory blocks. First, simplify the notion of a control flow graph to a list which contains only two types of object. The first of these is an access to a memory block, and the second a branch containing a set of sub-traces. While this definition does not contain the notion of loops, or other more complicated structures expressible in a control flow graph, it is capable of expressing a sound bound on the paths through a program provided that all loop bounds are known[1]. The function *step*, which steps through a given graph G, updating hit and miss counters $\langle h, m\rangle$, (and the auxiliary functions *updateHitsMisses* and *mergeHitsMisses* which provide update and merging facilities for a counter of hits and misses respectively) can then be defined as follows:

$$updateHitsMisses(T, \langle h,m\rangle, m) = \begin{cases} \langle h+1, m\rangle \textbf{ if } \forall t \in T, m \in T \\ \langle h, m+1\rangle \textbf{ if } \forall t \in T, m \notin T \\ \langle h, m\rangle \textbf{ otherwise} \end{cases} \tag{4.23}$$

$$mergeHitsMisses(\langle h_1, m_1\rangle ... \langle h_n, m_n\rangle) = \langle (min(h_1...h_n), min(m_1...m_n))\rangle \tag{4.24}$$

[1]A more advanced analysis would be able to take into account information on feasible paths, but this is beyond the scope of PLRU analysis.

$$step(T, G, \langle h, m\rangle) = \begin{cases} \langle T, \langle h, m\rangle\rangle \textbf{ if } G = \emptyset \\ step(T', tail(G), \\ \quad updateHitsMisses(T, \langle h, m\rangle, head(G))) \textbf{ if} \\ \quad head(G) \text{is a memory block}, T' = next(T, head(G)) \\ step(\cup\{T \big| \langle T, \langle h, m\rangle\rangle \in R\}, tail(G), \\ mergeHitsMisses(\{T \big| \langle T, \langle h, m\rangle\rangle \in R\})) \textbf{ if} \\ \quad head(G) \text{is a branch}, \\ \quad R = \{step(T, g, \langle h, m\rangle) \big| g \in head(G)\} \end{cases} \tag{4.25}$$

The *step* function is recursive and broken into 3 clauses; the first clause is the base case: if the end of the program or branch has been reached, then the function should return the set of cache states under consideration as well as the current hit and miss counters. The second case handles a simple memory block, and simply updates the cache states, hit and miss counters using the appropriate functions. The final case handles branches. Here, each branch is evaluated separately with the starting point being the current cache states and hit/miss counters. The return values of these are collected, and the next value for the analysis after the branches have merged considers any cache state that could result from a branch, with the fewest guaranteed hits and misses observed forming the counters.

Finally, the analysis function can be defined as follows:

$$analysis(T, G) = \langle h, m\rangle | \langle T', \langle h, m\rangle\rangle = step(T, G, \langle 0, 0\rangle) \tag{4.26}$$

Which returns the hit and miss counters for a given set of initial states T (typically, either a singleton containing either a known initialised state or a completely unknown state) and a given graph of memory accesses G.

4.5 Proof of Soundness

In order for the approach given with to be sound, the merge, touch and evict operators cannot cause a potential concrete state to not be considered by the analysis. Much of the proof of soundness can be inferred by the correctness of the Collecting Semantics and the map between the abstract states used in Full Tree analysis given by (4.4). Hence it is necessary to ensure that the following properties hold:

- That the *merge* operator does not result in the exclusion of valid states.
- That in the presence of unknown pointers, the *evict* and *touch* operators do not exclude valid states.

In order to accomplish this, two results will be proven:

- Merging is sound: For any set of cache states A, merging does not result in an unsound approximation. That is, $\forall A$, $cSet(A) \subseteq cSet(merge(A))$.
- Updating is sound: For any set of cache states, A, and memory block m, then for each $\mathbf{a} \in A$, $touchSet(evict(\mathbf{a}, m)) \subseteq cSet(touchSet(evictSet(A, m)))$.

4.5.1 Soundness of Merging

The first step in proving the soundness of Full Tree analysis is to show that the merging operation is sound. This can be accomplished by showing that for a set of cache states A, the following is true:

$$\forall A, cSet(A) \subseteq cSet(merge(A)) \tag{4.27}$$

That is, that given a set of cache states A, any cache state that is represented in A is also represented after A has been merged. Showing this demonstrates that no cache state that is in A can be removed from consideration by the *merge* function, and hence that the merge operation is sound as it provides a set of states which bounds the states under analysis. (4.27) can be first rewritten by utilising the definitions of *cSet* (4.5) and *merge* (4.19) to rewrite these functions, yielding the following:

$$\bigcup\{c(\mathbf{a})|\mathbf{a} \in A\} \subseteq \{c(\mathbf{a})|\mathbf{a} \in \bigcup\{mergeStatesSet(B)|B \in splitBySignature(A)\}\} \tag{4.28}$$

As the left hand side of 4.28 is a union, the set of cache states A can be split up providing that all cache states are still considered. For the sake of creating symmetry between the sides of the equation, it would be desirable to use the *splitBySignature* function to do this. In order for this to be valid, it is necessary to examine the *splitBySignature* function to ensure that it behaves as desired.

The *splitBySignature* function is intended to take a set of cache states and group them into a series of sets such that the signature of each element in each set is the same. Further, the elements returned have the *sort* function applied to them, such that the signature of the cache state can be read simply. As the *sort* function only performs subtree flipping, it can be trivially deduced to be sound as subtree flipping is a sound operation. Similarly, the *splitBySignature* function must be sound, as each cache state has exactly one signature, and therefore must be in exactly one of the returned sets, therefore ensuring that no cache state can be removed by applying *splitBySignature* to a set of cache states. Therefore:

Lemma 1. *The function splitBySignature applied to a set of cache states A does not cause the loss of any cache states.*

Proof. By above,

$$\mathbf{a} \in A \Rightarrow sort(\mathbf{a}) \in \bigcup splitBySignature(A) \tag{4.29}$$

□

Observing the left hand side of Equation 4.28, it is possible to apply Lemma 1 to each $a \in A$. This yields the following:

$$\bigcup\{c(\mathbf{a})|\mathbf{a} \in A\} \Rightarrow \bigcup\{c(\mathbf{a})|sorted(\mathbf{a}) \in \bigcup splitBySignature(A)\} \tag{4.30}$$

As for each $\mathbf{a}$, $\mathbf{b} = sorted(\mathbf{a})$ will be in one of the sets $B \in splitBySignature(A)$, we can perform an additional rewrite to obtain:

$$\bigcup\{c(\mathbf{a})|\mathbf{a} \in A\} \Rightarrow \bigcup\{\cup\{c(\mathbf{b})|\mathbf{b} \in B\}|B \in splitBySignature(A)\} \tag{4.31}$$

$$p(\langle h, v, \mathbf{l}, \mathbf{r}\rangle) = \begin{cases} \langle v\rangle & \textbf{if } h = 1 \\ \langle a_0 ... a_m\rangle & \textbf{where } \langle a_0...a_n\rangle = p(\mathbf{l}) \\ & \langle a_{n+1}...a_m\rangle = p(\mathbf{r}) \textbf{ otherwise} \end{cases}$$

Figure 4.9: An example function mapping pointer values to a vector

Hence, combining (4.31) with (4.28), specifically by using (4.31) to rewrite the cache set A on the left hand side to take the same form as A on the right hand side, it is possible to obtain the following:

$$\begin{aligned} \bigcup\{\cup\{c(\mathbf{b})|\mathbf{b} \in B\}|B \in \mathit{splitBySignature}(A)\} = \\ \{c(\mathbf{b})|\mathbf{b} \in \cup\{\mathit{mergeStatesSet}(B)|B \in \mathit{splitBySignature}(A)\}\} \end{aligned} \quad (4.32)$$

The function *splitBySignature* simply splits a set of cache states into sets such that every element of each cache state in the set has the same signature and changes their form to being sorted. Observing this, (4.32) is satisfied by showing that the inner components hold for sets of cache states with a single signature which are sorted, as follows:

$$\begin{aligned} \bigcup\{c(\mathbf{b})|\mathbf{b} \in B\} \subseteq \{c(\mathbf{b})|b = \mathit{mergeStatesSet}(B)\} \\ \text{assuming} \forall \mathbf{b_1}, \mathbf{b_2} \in B, \mathit{signature}(\mathbf{b_1}) = \mathit{signature}(\mathbf{b_2}) \end{aligned} \quad (4.33)$$

In the recursive case, the function *mergeStateSet* returns $\mathit{mergeStates}(\mathbf{b_0}, \mathit{mergeStateSet}(\{\mathbf{b_1}...\mathbf{d_n}\}))$. If one uses the binary operator $\odot$ to represent $y \odot x = \mathit{mergeStates}(x, y)$, then the *mergeStates* function can be simply expressed as $\mathbf{d_n} \odot \mathbf{d_{n-1}} \odot ... \odot \mathbf{d_0}$, i.e. a sequence of applications of the *mergeStates* function. The *mergeStates* function is defined to take sorted cache states (as per the assumption given by Equation 4.33) and set any pointers which differ to be $\perp$. This can be seen trivially from its recursive definition which only modifies a pointer to be $\perp$ if the pointer in the two subtrees to merge differs, otherwise it keeps the same value.

As the *mergeStates* function only sets pointers to be $\perp$ when two pointers in the same location differ, the effect of a sequence of applications can be expressed on a per pointer basis. Let the function p be a map from cache states to a vector of pointers in that cache state, such that the notation $p(\mathbf{b})_l$ refers to the value of the pointer at location l for a given cache state b (while the exact definition of p is not necessary for this proof, provided that p maps the same locations in the cache to the same indices of the vector it produces, for completeness an example of such a function is given in Figure 4.9). The effect of a sequence of applications on each location will be that $p(mergeStateSet(B))_l = \perp$ if and only if there exist $\mathbf{b_1}, \mathbf{b_2} \in B$ such that in $p(\mathbf{b_1})_l = 0$ and $p(\mathbf{b_2})_l = 1$. Alternatively, for $\mathbf{b} \in B$, $p(mergeStateSet(B))_l = p(\mathbf{b})_l$ if and only if for all $\mathbf{b'} \in B$, $p(\mathbf{b})_l = p(\mathbf{b'})_l$

A trivial consequence of this is that for all $\mathbf{b} \in B$, for all pointer locations l, then $p(mergeStateSet(B))_l \in \{p(\mathbf{b})_l, \perp\}$. This is because either for all $\mathbf{b'} \in B$, $p(\mathbf{b})_l = p(\mathbf{b'})_l$ and hence $p(mergeStateSet(B))_l = p(\mathbf{b})_l$ or alternatively there exists $\mathbf{b'} \in B$ with $p(\mathbf{b})_l \neq p(\mathbf{b'})_l$ and hence $p(mergeStateSet(B)) = \perp$. Therefore:

Lemma 2. *Given B, a set of cache states with the same signature, and l, a pointer location in the cache tree, $\forall \mathbf{b} \in B$, $p(mergeStateSet(B))_l \in \{p(\mathbf{b})_l, \perp\}$.*

Proof. By above. □

To see the effect of this on the concretisation function c (4.4), observe that c guarantees that for each unknown pointer, both possible values are considered. As by Lemma 2, for a given cache state $\mathbf{b}$, contained in a set of cache states with the same signature B, each pointer in $mergeStateSet(B)$ is either identical to the corresponding pointer in $\mathbf{b}$ or unknown. As this excludes the possibility of a conflicting known pointer, it is trivial to conclude that $\mathbf{b} \in c(mergeStateSet(B))$.

Corollary 3. *Given B, a set of cache states with the same signature, $\forall \mathbf{b} \in B$, $\mathbf{b} \in c(mergeStateSet(B))$.*

Proof. Observing that by Lemma 2, pointers in $mergeStateSet(B)$ either match their counterpart in $\mathbf{b}$ or are unknown, and hence b must be in the set returned by $c(mergeStateSet(B))$. □

Therefore, (4.33) is satisfied with the restriction of a single signature, and hence (4.32) is satisfied with multiple signatures, and hence by Lemma 1 (4.28) and (4.27) are satisfied. Therefore, one can conclude:

Theorem 4 (Soundness of Merging)**.** *For an arbitrary set of cache states B, $cSet(B) \subseteq cSet(merge(B))$.*

Proof. Observing that by construction, (4.27) is satisfied by the proof of Corollary 3. □

4.5.2 Soundness of Updating

In order to complete the proof of soundness, it is necessary to show that the algorithm behaves appropriately in the presence of unknown data. This can be accomplished by showing the following statement is true:

$$\forall m, A, \mathbf{a} \in cSet(A) \Rightarrow touchSet(evict(\mathbf{a}, m), m) \subseteq cSet(touchSet(evictSet(A, m), m)) \tag{4.34}$$

Where A is a set of abstract states and m a memory location. Specifically, if this statement is true, then for each concrete cache state represented by an abstract state, when that cache state is updated with a memory access m, the updated state is contained within the updated abstract state.

First observe that as *cSet* is the union of c on all elements of the set A, there exists an abstract state $\mathbf{x} \in A$ such that $\mathbf{a} \in c(\mathbf{x})$; by definition, this implies that the signatures of $\mathbf{a}$ and $\mathbf{x}$ are the same. As the *evictSet* function takes the union of applying the *evict* functions respectively, for a given $\mathbf{a}$ and corresponding $\mathbf{x}$, rewriting the right hand side of (4.34) gives:

$$cSet(touchSet(evict(\mathbf{x}, m), m)) \tag{4.35}$$

Next observe that by Lemma 2, any pointer in $\mathbf{x}$ must be equal to the corresponding pointer in $\mathbf{a}$ or unknown. The function *evict* returns a set where cache lines which can be pointed at (considering the unknown pointers in the state) are evicted and replaced with the new memory location; in the case that a pointer is unknown, both possibilities are considered. As no pointer in $\mathbf{x}$ contradicts a pointer in $\mathbf{a}$ (Lemma 2), it can be inferred that there exists $\mathbf{x}' \in evict(\mathbf{x}, m)$ such that $evict(\mathbf{a}, m) \in c(\mathbf{x}')$. Using $\mathbf{x}'$, and noting that $touchSet$ and $cSet$ function take the union of applying the c and $touch$ functions respectively, (4.35) can be rewritten as:

$$cSet(touch(\mathbf{x}', m)), \forall m \tag{4.36}$$

As $evict(\mathbf{a}, m) \in c(\mathbf{x}')$, it can be inferred that $evict(\mathbf{a}, m)$ and $\mathbf{x}'$ have the same signature. Now note that the $touch$ function updates pointers solely on which subtree a memory location resides in; as $evict(\mathbf{a}, m)$ and $\mathbf{x}'$ have the same signature it follows that the $touch$ function will update the same pointers in $evict(\mathbf{a}, m)$ and $\mathbf{x}'$. Hence, it is possible to define $\mathbf{x}'' = touch(\mathbf{x}', m)$, where $touchSet(evict(\mathbf{a}, m), m) \in cSet(\mathbf{x}'')$. Using this, rewriting (4.36) gives:

$$cSet(\mathbf{x}'') = touchSet(evict(\mathbf{a}, m), m) \tag{4.37}$$

As, by definition, $touchSet(evict(\mathbf{a}, m), m) \subseteq cSet(\mathbf{x}'')$, we can therefore conclude (4.34) to be true and hence:

Theorem 5 (Soundness of Updating)**.** *Given a set of abstract cache states A, $\forall \mathbf{a} \in A$, $\forall$ memory locations m, $touchSet(evict(\mathbf{a}, m), m) \in cSet(touchSet(evictSet(A, m), m))$, and hence updating a cache state is sound regardless of the presence of unknown data.*

Proof. By above. □

4.5.3 Combining to the Final Proof

Theorem 4 showed that merging cache states is sound for a set of arbitrary cache states. Theorem 5 showed that updating a set of abstract cache states is sound in the presence of unknown data. Hence, as each step of the proposed algorithm is sound, it only remains to state the final result:

Theorem 6 (Full Tree Analysis is Sound)**.** *Proof.* Observing that by Theorems 4 and 5, and the map between states in Full Tree analysis and the Collecting Semantics (4.4) all steps of the algorithm are sound, and hence the algorithm as a whole is sound. □

4.6 Generalisation to HNMRU Caches

As introduced in Chapter 2, the HNMRU cache is a generalisation of the PLRU cache. HNMRU provides more configurability, specifically in regards to the trade-off between cache performance and silicon used on chip; hence while an HNMRU cache may exhibit lower performance than a PLRU cache, it also requires less power to operate. This section adapts Full Tree PLRU analysis to work with an HNMRU cache. This is accomplished by applying the same information theoretic techniques as used in the derivation of Full Tree PLRU analysis to the HNMRU cache where there is a substantial difference, and using this information to derive an analysis appropriate for HNMRU.

4.6.1 Modelling the Differences Between HNMRU and PLRU

As an HNMRU cache is similar to the PLRU cache in many ways, it is necessary to determine how to model the differences between the two. The only substantial difference is the presence of a random number generator (RNG) in the HNMRU cache, which presents some difficulty to model. Whilst the random numbers generated by the RNG are only used during eviction, the RNG successor state is a function of its current state. This means that compressing the RNG state is not an option, as an uncertainty in the RNG state would not only effect the next eviction (as is the case with pointers) but every subsequent eviction, as there is no prospect of returning to a certain RNG state. Whilst the alternative, not merging RNG states, may lead to state explosion, having every eviction return multiple states will certainly lead to state explosion.

Provided that the RNG is updated regardless of a cache hit or miss, on a single code path, the RNG will be constant across all considered cache states (as the number of accesses is constant on a single code path) and hence merges will be able to take place as necessary. This assumption fails when multiple code paths require consideration, as the number of accesses to memory will likely differ, and hence merging states from different paths becomes impossible. Hence, in order to analyse an HNMRU cache, it is necessary to adapt the HNMRU cache algorithm in a manner that mitigates this problem. Two such adoptions are presented below, with a brief evaluation of the caveats associated with each:

1. Remove the random elements from the HNMRU scheme entirely and use a deterministic scheme (a simple example of this being a modification of PLRU: when the most recently used path is set to the i'th branch, evict by following the $(i + 1)$'th branch). The disadvantage of this scheme is that the pathological case, where useful memory addresses are consistently evicted in minimal time, becomes easily repeatable, and hence this scheme would be expected to perform poorly.

2. Use a cyclic RNG with a relatively small cycle. Provided that the RNGs cycle is sufficiently small, and the initial value is known, this provides the possibility of being able to merge states from different paths once the RNG occupies the same state. Whilst a cyclic RNG could potentially remove the benefit of using an RNG in the first place,

namely making a pathological worst-case probabilistically impossible to encounter repeatedly, this would only be of real concern if the cycle of the RNG and the length of pathological series of memory accesses had a relatively low lowest common multiple, as otherwise it would take a large number of loops for the worst-case RNG state to reappear at the correct position in the loop. Hence it would be relatively trivial to modify a program to exclude this possibility by inserting dummy load instructions.

4.6.2 Adapting Full Tree analysis to HNMRU and Determining Expected Performance

For analysing HNMRU, the Full Tree PLRU algorithm can be generalised by two simple modifications. The first of these considers the RNG, and is not necessary if the RNG is not present, as in the first mitigation described. To enable multipath analysis with the RNG, it is necessary to store the present state of the RNG, and not merge states if the RNG states differ. The given algorithm can be generalised to the HNMRU cache with a simple modification; as the usage of data within the cache state remains the same, it still holds that the pointers are the most promising candidate to be lost due to compression. Then the only remaining issue is to adapt the mapping of physical states to logical states to take account of the generalised tree structure of HNMRU. This can be accomplished by simply changing the algorithm to perform a full sort of all elements in each subtree at every depth, keeping pointers pointed at the same element, rather than just the simple check to see if two subtrees are in the correct order performed when analysing PLRU. This is illustrated in Figure 4.10.

One interesting property of HNMRU caches, with respect to analysis is that the sizes of the compressed state spaces vary dramatically. For example, a 12-way HNMRU cache can be implemented as any of 2-6, 2-2-3, 2-3-2, 3-2-2, 3-4, 4-3, or 6-2, which vary from having 462 ways to store 12 unique elements in the case of the 6-2 cache to 155925 in the case of the 3-2-2 cache. This leads to a tradeoff: different cache configurations can take different amounts of effort to analyse, albeit with the effect of a higher degree of uncertainty being introduced when cache pointers are compressed. Hence, when deciding on an HNMRU cache for a real-time system, it is necessary to consider the configuration of the cache for analysis.

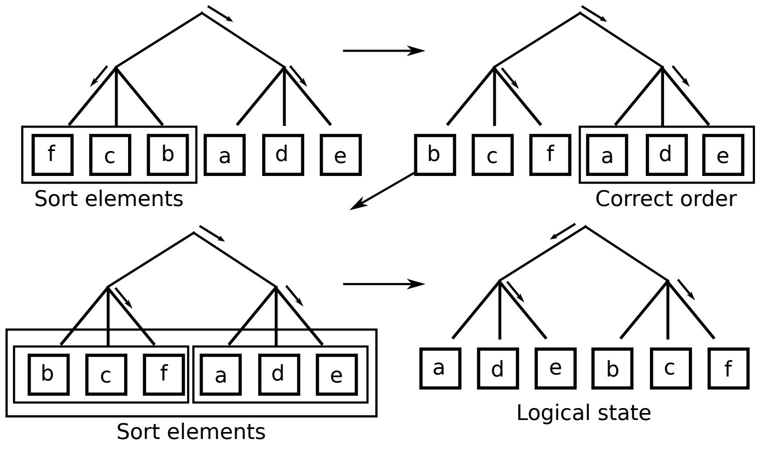


Figure 4.10: Subtree sorting an HNMRU 2-3 Cache

4.7 Evaluation

4.7.1 Comparison to Existing Methods

Evaluation was carried out on three sets of benchmarks, using three different methods. Full Tree analysis (*ft*) was compared against the Collecting Semantics (*cs*), and the current state of the art, Grund and Reineke's Potential Leading Zeros analysis (*plz*). The first set of benchmarks are synthetic benchmarks, mostly compiled by Grund and Reineke [54]. The second and third sets of benchmarks are multipath benchmarks extracted from the Mälardalen [35] and Papabench [44] benchmark suites respectively; for these, due to excessive memory usage, it proved impossible to provide a comparison against the *plz* method.

Synthetic Benchmarks

The Must analysis of the Full Tree algorithm (PLRU-ft) was compared against the current leading analysis provided by Grund and Reineke [54] based on approximating leading zeros (PLRU-plz) and the Collecting Semantics of a PLRU cache (PLRU-cs), using the benchmarks that Grund and Reineke chose to present a fair comparison. These benchmarks were evaluated on a fully associative cache, with the intention of stress testing the algorithm. The benchmarks are as follows:

- *loopK*: Subjects the analysis to a loop of the memory blocks 1 to K,

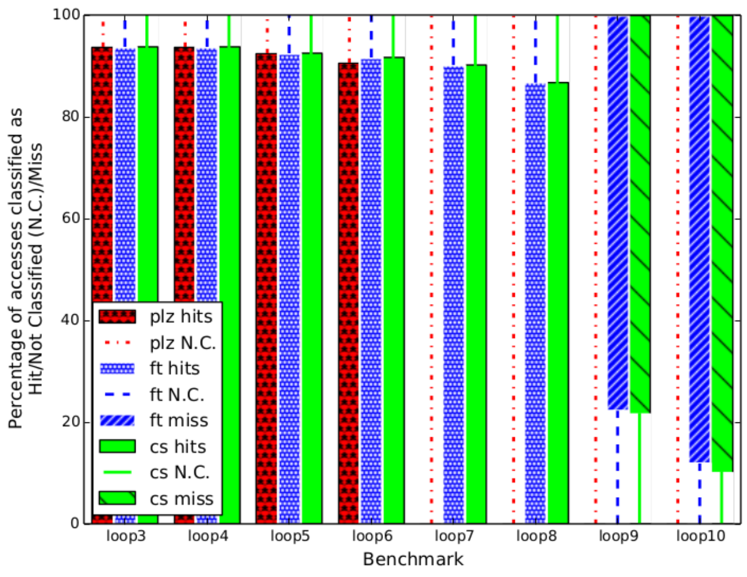


Figure 4.11: Results for the synthetic *loopK* benchmark

with the loop repeating 16 times. This test provides a stress test to show the boundaries of the analysis.

- *randomK*: Subjects the analysis to accesses to 100 memory blocks randomly chosen from range $(1, K)$. This test provides a sample of performance in typical non-looping conditions.

When the synthetic benchmarks were tested on a fully associative 4-way cache, the Must analysis of PLRU-ft behaved identically to PLRU-plz; in addition, the running times of the algorithms were approximately the same. The reason for this is that the tree structure of a 4-way cache is simple enough that PLRU-plz can represent it accurately, and hence PLRU-plz provides an accurate Must analysis. Hence the main benefit of PLRU-ft in the case of a 4-way cache is the May analysis it provides. All remaining experiments were performed on a fully associative 8-way cache.

Figure 4.11 shows the classification results for the *loopK* benchmark. This illustrates that unlike *plz*, *ft* is capable of analysing all elements in the cache. This is shown by the fact that it can predict hits for loops of size 7 and 8. Figure 4.12 shows the time that the analysis takes. Here, the Full

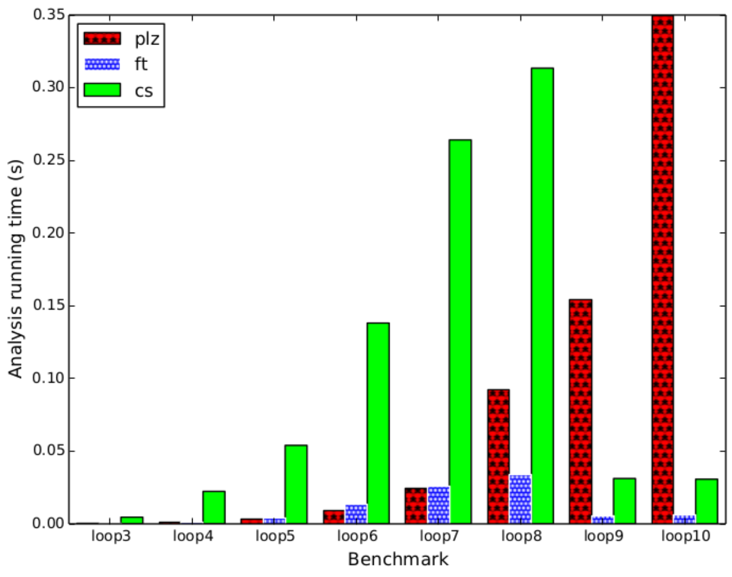


Figure 4.12: Analysis time for the synthetic *loopK* benchmark

Tree Analysis *ft* is up to 10 times faster than using the Collecting Semantics *cs*. It is also much faster than *plz* for high values of K, due to the additional pessimism of *plz* resulting in *plz* analysing a large number of states.

Similar benefits are seen in the *randomK* benchmarks. Figure 4.13 shows the classification results for the *randomK* benchmarks. Again, Full Tree Analysis *ft* outperforms Potential Leading Zeros analysis *plz* in all cases. Similarly, the time taken to reach this result is lower than that seen in *cs*, with the largest observed difference being approximately 10 times. Again, for large values of K, *ft* is much faster than *plz*.

Mälardalen and Papabench Benchmarks

To assess the effectiveness of Full Tree Analysis on more realistic benchmarks, the Mälardalen [35] and Papabench [44] benchmarks were used. The benchmarks were compiled for the MIPS architecture and the resulting binaries interrogated using the Heptane analyser [32] to find a control flow graph bounding all paths through the program. These graphs were then analysed using the *ft* and *cs* methods, assuming a fully associative 8-way PLRU cache with a line size of 32 bytes. Note a relatively small cache size

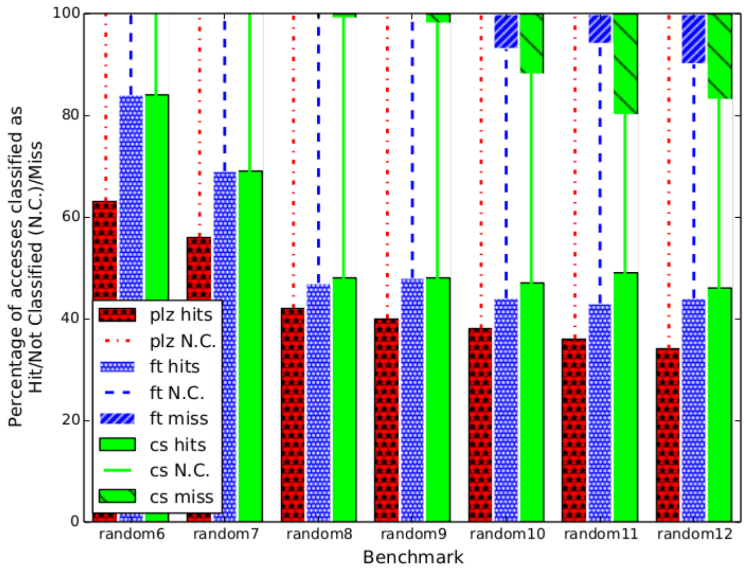


Figure 4.13: Results for the synthetic *randomK* benchmark

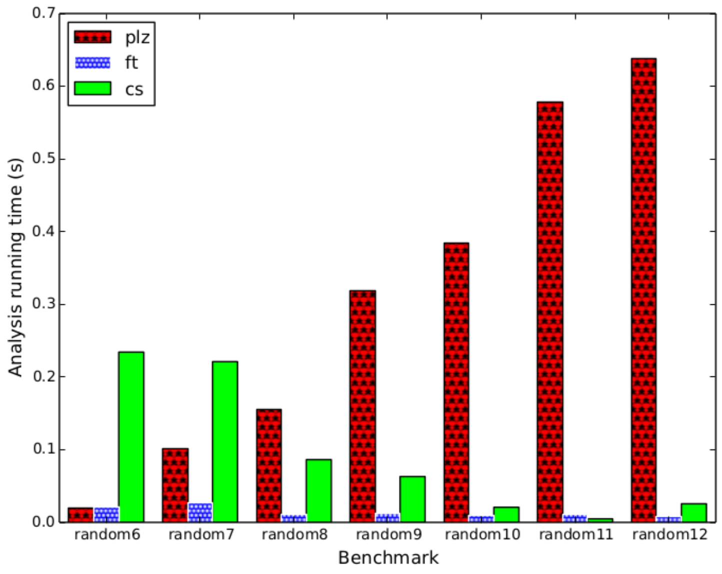


Figure 4.14: Analysis time for the synthetic *randomK* benchmark

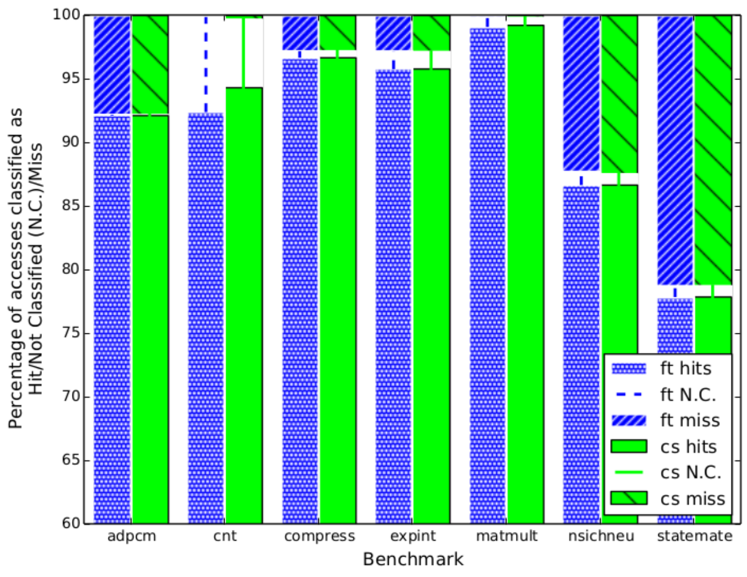


Figure 4.15: Results for selected Mälardalen benchmarks

was chosen so that the effects of low hit rates could be examined. The size of the control flow graphs analysed varied from less than 1KB (*cnt*) to more than 100MB (*compress*), demonstrating that the technique is applicable to complex programs.

Figure 4.15 shows the results for selected Mälardalen benchmarks, indicated on the x-axis. For the majority, the results from ft analysis are competitive with cs. The worst results are observed in *cnt*, which exhibits some additional pessimism due to the nature of the program branches which are merged resulting in infeasible states being considered combined with the comparatively high proportion of misses. These problems also negatively impact the analysis time of ft, as shown in Figure 4.16 which shows *cnt* taking longer to analyse using Full Tree analysis ft than Collecting Semantics cs. However, it should be noted that for all benchmarks of a substantial length, ft outperforms cs by a factor of between 2 and 5 (note the log scale on the graph). In the case of *cnt*, even though the benchmark takes longer to analyse under ft than cs, both still perform the analysis in less than a second.

The results for the Papabench benchmarks shows similar properties

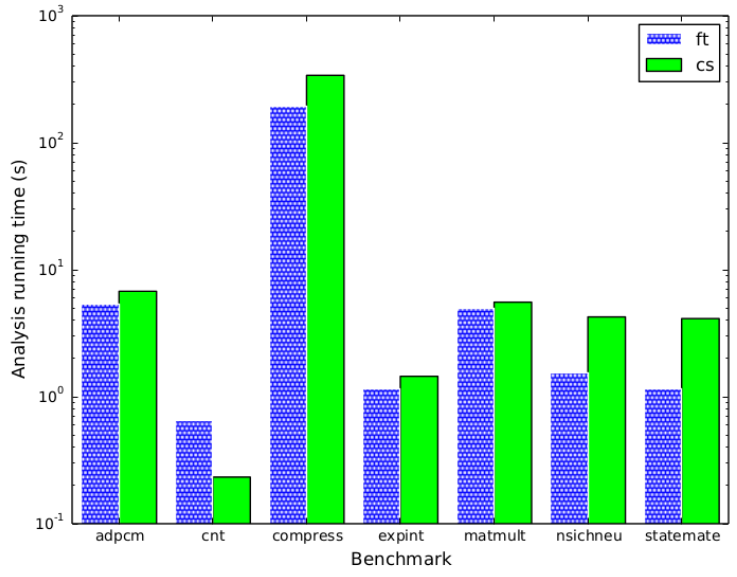


Figure 4.16: Analysis time for selected Mälardalen benchmarks

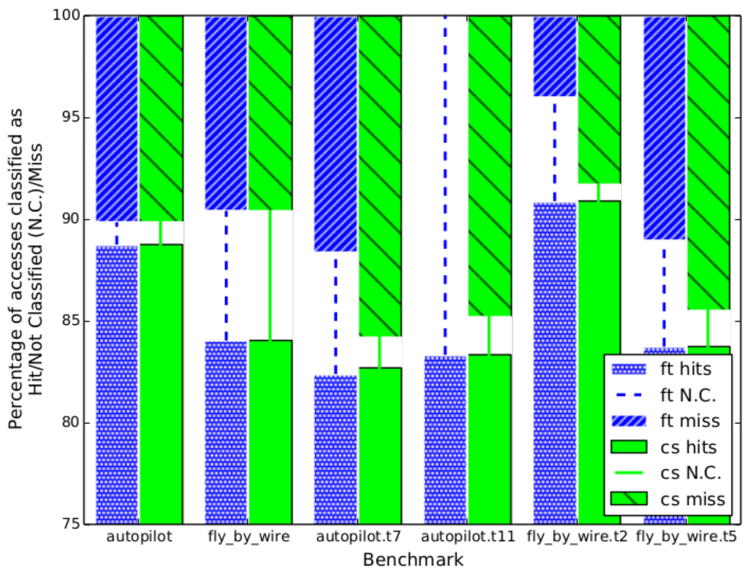


Figure 4.17: Results for the Papabench benchmarks

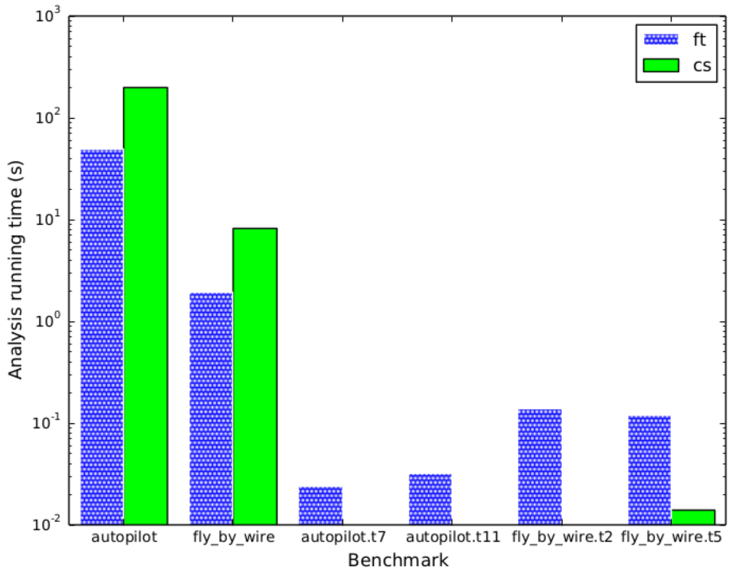


Figure 4.18: Analysis time for the Papabench benchmarks

to those for the Mälardalen benchmarks. Figure 4.17 shows that when analysing the benchmark as a whole, as is the case with the *autopilot* and $fly-by-wire$, results are competitive with the Collecting Semantics. However, when the individual tasks are analysed the effect of the introduced pessimism is proportionally higher, leading to proportionally worse results. Similarly, the time taken for analysing individual tasks is worse for ft than cs, but still less than a second. For the larger benchmarks, ft is faster than cs by a factor of 5.

4.7.2 Evaluating the Analysable Performance of HNMRU Against PLRU

Synthetic Benchmarks

Using the same methodology as comparing PLRU-ft to PLRU-plz, the analysable performance of the 8-way HNMRU caches (using the predictable eviction scheme) was evaluated, to determine if the claim of Roy [86] that HNMRU cache results in a minimal performance decrease compared to PLRU cache holds with respect to analysable performance. It should be

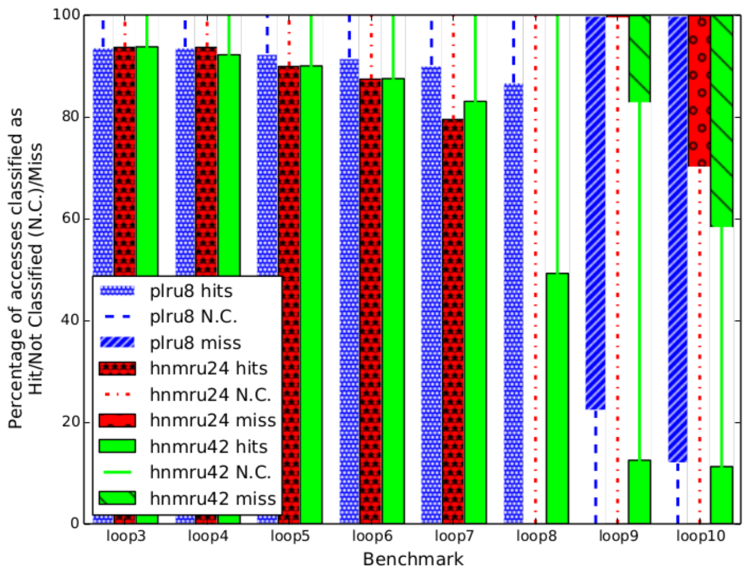


Figure 4.19: Comparison of the analysable performance of 8-way HNMRU and PLRU caches using the loop benchmark

noted that as a predictable eviction scheme has been used in place of the pseudo-random scheme used by Roy, it would be expected that the HNMRU cache could encounter pathological cases, which could degrade analysable performance significantly.

As seen in Figures 4.19 and 4.20, the HNMRU cache exhibits poorer analysable performance than the PLRU cache. The most obvious disadvantage of the HNMRU cache is that the May analysis yields much less accurate information, and as a result the analysis determines far fewer misses than for PLRU. Also, in most cases, the Must analysis of the HNMRU cache produces between 10 and 20% fewer hits than would have been found if a PLRU cache had been used instead. The exceptions to this occur either when the cache is being accessed by relatively small number of elements, in which case HNMRU 4-2 provides results very close to that of PLRU. Also, due to the fact that HNMRU can exhibit very different behaviour to PLRU, the corner case of the *loop*(9) benchmark the HNMRU 4-2 cache outperforms the PLRU cache, demonstrating that different cache policies can have specific corner cases where they are more applicable.

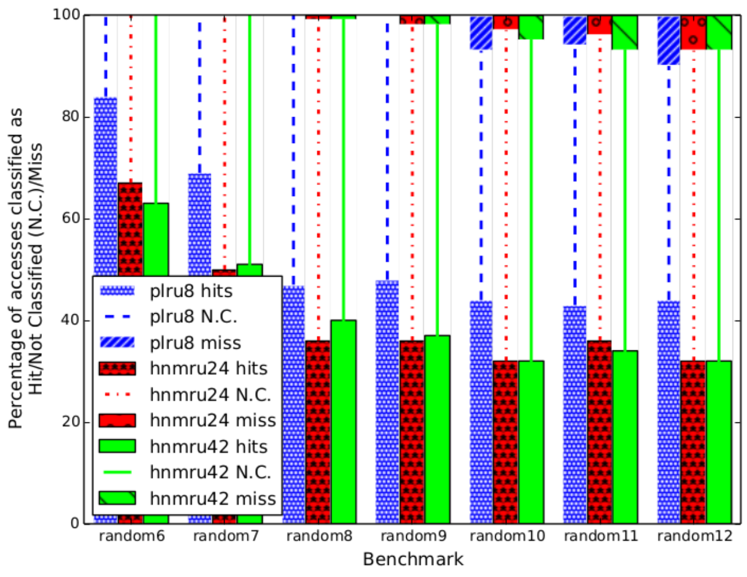


Figure 4.20: Comparison of the analysable performance of 8-way HNMRU and PLRU caches using the random benchmark

Mälardalen and Papabench Benchmarks

As with PLRU, real world and multipath code was evaluated by testing the performance on data extracted from the Mälardalen and Papabench benchmark suites.

In comparison with the synthetic benchmarks, Figure 4.21 shows that the HNMRU caches appear to have much better behaviour on the Mälardalen benchmarks. In particular, the HNMRU 4-2 cache rarely trails the PLRU cache by more than 5%. This is in part because of the scale of the Mälardalen benchmarks means that the uncertainty from the initial state persists for less of the total runtime than in the synthetic benchmarks, and partly because the Mälardalen benchmarks do not appear to place sufficient stress on the cache for the disadvantages of the HNMRU cache to be apparent. Assuming that the Mälardalen benchmarks are representative of “real life” code, these results would appear to indicate that even without the random selection of the eviction location, the analysable performance of HNMRU is sufficiently high for the cache to be useful in real applications.

Surprisingly, as seen in Figure 4.22, the HNMRU cache is favoured even

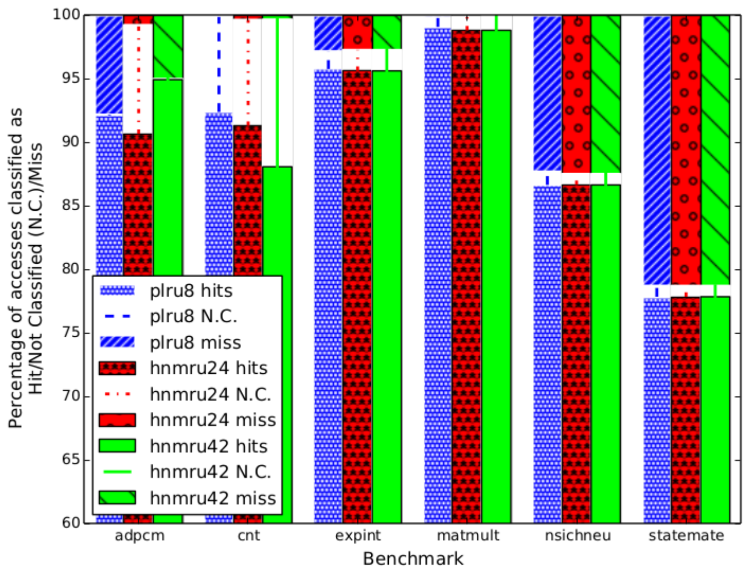


Figure 4.21: Comparison of the analysable performance of 8-way HNMRU and PLRU caches using the Mälardalen benchmarks

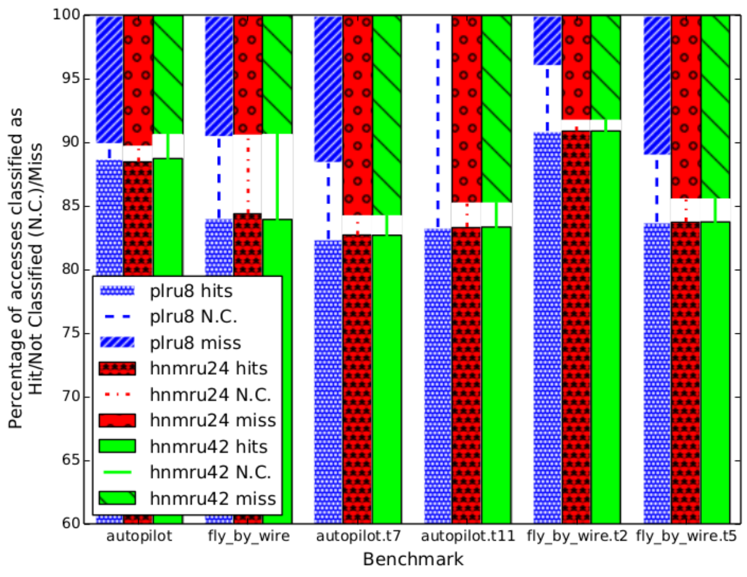


Figure 4.22: Comparison of the analysable performance of 8-way HNMRU and PLRU caches using the Papabench benchmarks

more by the Papabench benchmarks, yielding higher analysable performance on some benchmarks. This is expected to be due to the fact that fewer unknown pointers may be encountered, due to the shortened tree height of the HNMRU caches considered. As has been outlined previously, unknown pointers are directly responsible for pessimism, and therefore the reduction in the number of unknown pointers due to lower tree height could be responsible for the increased accuracy.

The difference in performance of HNMRU observed when comparing the synthetic and Mälardalen/Papabench benchmarks may also be due, in part, to pathological cases introduced by the removal of the pseudo-random element of HNMRU algorithm. It is necessary to at the very least degrade the quality of the RNG that HNMRU uses for the sake of analysability, as a true RNG would leave no method for merging multiple program paths. However, the fact that the performance penalty decreases substantially when analysing "real life" code would appear to indicate that any pathological case introduced is not a common occurrence and can be treated as the pathological cases for other cache architectures are, namely that when it is found that a program may trigger a pathological case, the program is modified to avoid the pathological case.

4.8 Summary

The main contribution presented in this chapter has been to apply the principles of lossy compression to the problem of deriving a simplification of a PLRU cache state, for use in abstract interpretation. This has resulted in the new Full Tree cache analysis algorithm, PLRU-ft. By preserving the tree structure and cache lines of all cache states provides a better estimate of the cache state than previous work, enabling a full Must/May analysis which was not previously possible. When compared to the previous Must analysis, PLRU-plz, PLRU-ft yields a tighter bound than PLRU-plz on every test conducted, while taking a similar or shorter amount of analysis time to do so.

A proof was presented of the soundness of PLRU-ft, which enables the use of the algorithm on critical real-time systems. An evaluation of PLRU-ft was carried out, which showed its accurate to be highly competitive with the Collecting Semantics of a PLRU cache, but using fewer resources. The evaluation also gave an indication of the analysable cache performance of PLRU on the Mälardalen benchmark suite.

The PLRU-ft algorithm was then generalised to a simplified form of the HNMRU cache. Roy [86] claimed that the HNMRU cache had similar performance to a PLRU cache, suffering from a 3-5% handicap depending on configuration. This claim was tested with respect to the analysable performance of an HNMRU cache, rather than the performance observed during testing. Due to concerns about the analysability of multipath code and knowledge required to effectively analyse an HNMRU cache, the HNMRU algorithm was simplified by removing the random element used, which may have introduced the possibility of pathological cases. Whilst HNMRU cache scored poorly for analysable performance on the highly synthetic benchmarks used for initial evaluation, on the Mälardalen benchmarks the analysable performance exhibited a handicap similar to the claims of Roy. On the Papabench benchmarks, this was largely repeated, although for some benchmarks HNMRU outperformed PLRU. This suggests that the pathological cases avoided by the use of a RNG in the HNMRU algorithm are not common, and may be avoidable by other means.

Chapter 5

Lossy Compression for Loop Bound Analysis

As detailed in Chapter 2, Loop Bound Analysis techniques typically require the use of annotations. Annotations are used to supply additional information to the analysis, which supplements the information the analysis can infer from the program. However, annotations rely on human input which may be incorrect [61]. As such, it would be desirable to find a way of automatically inferring loop bounds without the need for annotations.

Initially, this seems impossible: annotations are typically justified by the halting problem: As the halting problem is not solvable, annotations are used to supply additional information such that it is possible to place a bound on a program. Fortunately, the halting problem does not necessarily apply in this circumstance. Loop bound analysis is used to find a worst case - thus assuming that a worst case exists. Given the assumption of termination, computability theory [39] states that it is computable to find the maximum length of time that the program will execute for.

Hence if it is assumed that a program has a worst case, then finding the worst case is a computable property. Hence the problem is to find the worst case efficiently. As with the PLRU Cache in Chapter 4, Lossy Compression can again help. In this case, the problem is to identify information which can be discarded without a significant loss of accuracy, a representation which enables this compression, and when to perform the compression.

This chapter examines how such a scheme may be created, by explicitly using Information Theory to construct an appropriate compression scheme.

Section 5.1 examines the different types of information necessary for loop bound analysis and how they may be represented and merged. Section 5.3 expands upon this, given a more detailed overview of the strategies employed to implement the compression. This is continued in Section 5.4, which gives a detailed overview of the proposed algorithm. Section 5.6 examines the performance of the implementation on various benchmarks, and finally Section 5.7 gives a summary of the chapter.

5.1 Information Theory for Loop Bound Analysis

As with the PLRU cache, it is necessary to examine the information used in Loop Bound Analysis. Loops in programs fall into two categories. The first iterates a fixed number of times, dictated by some constant in the program. For these, loop bound analysis is trivial. The second kind of loop iterates a variable number of times, according to some variable. Obviously, as the bound cannot simply be read from the program source code, analysing these loops is substantially harder.

As loop bounds depend on variables, in order to perform loop bound analysis it is necessary to perform value analysis on variables. As value analysis is reducible to the halting problem, this part of the analysis is where the difficulty lies. Hence the major problem to solve is how to model the values of variables through the program.

Abstract interpretation provides a basis for starting value analysis [36]. Abstract interpretation provides a sound approximation of values by simplification of the problem. However, the main problem with abstract interpretation is that it may fail to prove the termination of a bounded program - a false negative - or require information in annotations. This is due to the simplifications involved; in order not to encounter the halting problem, the abstractions used typically create false negatives that need correcting via additional information.

In the field of real-time systems, the goal is not to prove that an arbitrary program does not terminate. Instead, the goal is to prove a bound on the termination of a program that is believed to terminate. Assuming that the program does terminate one does not have to be concerned with the halting problem. Hence different simplifications can be used such that useful information is propagated from the values of the inputs throughout the program. Support for this assumption could be given in the form of restricted programming rules [61] where termination is guaranteed, or could instead be as simple as an engineer judging that the program should terminate for all inputs.

However, the caveat to assuming that the program does terminate is that the technique becomes inapplicable should it be untrue. Specifically, if the program does not terminate for some input value, then the analysis will never be able to reach a conclusion. Fortunately, this is not an entirely dissimilar problem to any other excessive resource utilisation problem during analysis - at some point, for any analysis technique, the resources expended on analysis outweigh the value of the result, and the analysis is deemed a failure. Hence the net effect of assuming termination is to place an emphasis on the user to try and provide a terminating input and also give a bound on the amount of effort they wish to expend on analysis.

5.1.1 Types of Information

The first step in finding a compression algorithm is to identify the types of information. As stated, loop bound analysis can be broken into two portions: firstly the value analysis to determine the potential values of variables, and secondly how these interact to give loop bounds. This split gives an insight into the types of information in the problem.

The first type of information relates to the first problem: Value analysis. Specifically, the potential values of variables at a given point in the programs execution must be modelled in order to accomplish value analysis. The requirements and desired properties for a representation of values are examined in Section 5.1.2, including a strategy for merging.

The remaining types of information relate to program flow. When evaluating a function, the result of the function may have provable properties with regards to its input, that manifests in the form of additional assumptions that can be made. These assumptions are examined in Section 5.1.3. Further, when functions are used as part of the control flow of a program and value analysis indicates a range of possible values, multiple successor states may be considered. This results in additional assumptions being added to the analysis, to ensure that the assumptions made when investigating a specific path remain consistent. Further, if possible it would be desirable to merge the multiple paths from a branch when possible: the earlier this is done, the less effort needs to be expended on the analysis. The additional assumptions, as well as merge strategies, are discussed in 5.1.4.

5.1.2 Variables

Given that the main problem identified is that of variables and their impact on the control flow of a program, it is necessary to examine variables, how they are manipulated, and how they effect the control of a program.

At a very basic level, there are two types of variable within a computer. The first type of variables are *floating points*. Floating points approximate the real numbers, $\mathbf{R}$, and despite some notable shortcomings, are relatively successful at this. Whilst special values may exist for floating points (e.g. NaN as the result of 1 divided by 0), these values signal errors and should not be encountered in successful operation, and hence should not be encountered as the correctness of the program is assumed. The second type is *modular integers*. Generally, if a data type is not analogous to floating points, it is analogous to $\mathbf{Z}_{2^n}$, where n is the size of the data type in bits.

However, distinct from the actual representation of these variable types are the usage of such variables. For example, a constant floating point value shares more properties with modular integers than real numbers, given that it may only have a single value. Similarly, a high precision fixed point number, despite being analogous to modular integers may be used as an approximation of a real number. As abstract interpretation is concerned with the behaviour of a system rather than its physical state, it is important to attempt to determine how variables are used rather than what the variable actually is.

Once usage is considered, there is a third type of variable: irrelevant variables which do not effect the control of the program. A typical example of these could be the outputs of a function; once a function has an output, the variables used inside the function fall out of scope, and hence no longer effect the control flow of the program. Similarly, compilers may find variables which have outlived their usefulness as part of optimisations, and can be marked as irrelevant at any point. As compiler optimisations are assumed to be correct, information stored on any variable marked as irrelevant by the compiler can be dropped from analysis without consequence.

Having identified that variables are either analogous to integers, reals or irrelevant, it is necessary to examine the usage of variables within a program to determine how multiple values might exist in a single abstract state. Due to its usage, a special case is the program counter. As the program counter specifies the next instruction to execute, it is inherently intrinsic to the behaviour of the program. Hence, whilst it may be possible to have uncertainty in variables in general, the program counter must be represented accurately at all times.

Other variables do not have this special status, and therefore could be modelled as containing multiple values. Typically, a function is specified to accept a range of input variables of a given type, and this provides a useful starting point for a suitable representation. Functions may also exclude values from the range of valid inputs; for example, division by zero is invalid and should not occur in a correct program. Ranges of values with exclusion provide a benefit to compression in that they are easy to merge into a single state. Hence the representation chosen is composed of three components:

- Range: An open or closed range encompassing all values which the variable could contain.
- Exclusions: A list of open or closed ranges which encompass values which the variable cannot contain.
- Usage: A switch determining the usage of the data: Integer, Real, or Irrelevant.

This representation gives a simple way to specify all values that may be permissible for a variable. Further, it is trivially possible to losslessly merge two possible sets of values for a variable, by picking a new main Range encompassing both old ranges, using both sets of exclusions and adding a new exclusion if required. Unfortunately, only using lossless compression will increase the size of the representation, as the number of sets of excluded values will increase. Hence it is necessary to use lossy compression and discard information. A sound way to discard information is to simply discard elements of the list of exclusions; the value of ranges will still contain all possible values, but will include additional invalid values.

Hence this representation for variables efficiently represents possible values, the usage of the variable and has the ability for efficient but sound loss of information to aid tractability. Next it is necessary to determine how these variables are modified and used.

5.1.3 Functions and Structure

Once the basic usage and representation of variables has been stated, it is necessary to examine how variables are modified. Stating the obvious, computer systems provide a basic set of operations which are used to manipulate variables. These operations are defined as mathematical operators; hence examining the properties of the mathematical operators yields information on how modifications should occur.

Computer programs, regardless of programming language or any other implementation method, must be expressible in terms of a sequence of mathematical operators. If they are not, then the program cannot run on the given hardware, and hence the program is not useful. Hence, if the effects of these operators can be represented sufficiently accurately it is possible to analyse any program. Therefore, this low level form of analysis gives a common base across all programs, and further provides a relatively small set of operations which need analysis. Further, a low level analysis can take into account any optimisations which a compiler may make.

Using low level information differs from current approaches which use high level information [100]. Approaches utilising high level representations state advantages such as being able to easily extract information such as constant loop bounds by identifying common ways to implement such bounds and finding these in the code. This in turn aids in proving the existence of loop bounds on more complicated structures. However, as already stated, it should be possible to assume that loop bounds exist, and hence there is no explicit requirement to prove the existence of loop bounds - only the values of loop bounds that exist. In turn, this makes many of the reasons for using high level information redundant; whilst it may be easier to prove the existence of a loop bound from high level information, the number of times any loop can execute given a range of inputs is a constant regardless of representation.

One side effect of discarding high level information is that this also discards information on the structure of the program. From the point of view of a loop bound analysis, the most problematic piece of structure discarding is the loops themselves. Once a low level approach is used, code is represented as basic blocks, and loops may not be obvious. Fortunately, if one is able to translate between the low and high level representations, a loop bound can be constructed from bounds on the number of times that the basic blocks inside that loop may execute.

Many of the mathematical operators available are expressible in terms of addition and multiplication, as applied to the real numbers. In both cases, these operations perform a well defined mathematical group with the real numbers. A mathematical group gives a guarantee on the following properties, for a binary operations $\cdot$:

1. **Closure**: For all x, y in the group, $x \cdot y$ is also in the group
2. **Associativity**: For all x, y, z in the group, $x \cdot (y \cdot z) = (x \cdot y) \cdot z$
3. **Identity Element**: There exists an identity element i such that $x \cdot i = x$
4. **Inverses**: For each element x, there exists x^{-1} such that $x \cdot x^{-1} = i$

In their usage in computer systems, all of these properties are preserved by the approximation of reals as floating point numbers when using addition and multiplication. However, in the case of the approximation of integers as modular integers these properties are not preserved. This is because in computer systems an integer overflow is treated as an error condition. For addition, this causes the property of closure to be invalid. Fortunately, in practice this should not be a problem, as overflow is normally treated as an error in the program.

In addition, multiplication also loses the group property when applied to modular integers, as $\mathbf{Z}_x$ only forms a group under multiplication if x is a prime number, because otherwise not all elements will have inverses. This is almost never the case, because as stated before the modular integers in a computer system are mapped to $\mathbf{Z}_{2^n}$, the integers modulo 2^n (typically with $n = 32$ or 64); hence under multiplication, these integers only form a group under multiplication for $n = 1$. Therefore, for any modular integer variable that can hold more than two values, the property of Inverses can not be assumed, and hence any analysis must take this into account.

In practice, the loss of inverses results in potential loss of accuracy when applying multiplication to ranges of integer values. This can be seen when multiplying two ranges of numbers together; if $x \in [1, 2]$ and $y \in [3, 4]$, $xy \in [3, 8]$. If x and y are real numbers, than xy can be any number in the range $[3, 8]$. This is not true in the case that x and y are integers, as in this case xy cannot be 5 or 7. There are three views to dealing with this problem: the first is that as multiplication must introduce some additional constraints with regards to divisibility and that these constraints must be represented. The downside to this option is a significant increases in the complexity of the representation and additional complexity when determining what values a variable can hold. The second view is that a program is unlikely to undo work that it has already done. Taking the second view, these constraints do not have to be represented, as the divisibility of a number obtained by multiplication is unlikely to be checked. However, without these additional constraints invalid results may be introduced. For example, if a memory address were computed by multiplying $x \in [1, 4]$, a range of values with a single value $y = 4$, without the constraint of divisibility by the y any value in the range $[4, 16]$ could be considered. However, as a memory address is likely to refer to some form of memory structure, memory addresses not divisible by 4 are likely to be invalid and even nonsensical should they be evaluated. Finally, a third option would be a compromise, which only stored a limited amount of information.

For both addition and multiplication, it is possible to add properties on the value of the output. For example, adding positive numbers results in a number strictly greater than either input. Given that such properties can inferred as mathematical fact and give useful information about the value of the output, this information should be kept. In particular, these additional properties may not be obtainable by other means; for example, adding the ranges $[1, 2]$ and $[2, 4]$ gives a value in the range $[3, 6]$; without exploiting the properties of addition, it would be impossible to prove that the result is greater than the second operand, as the ranges intersect.

Other functions may not provide as many properties for free. For example, the modulo operator does not preserve the properties of groups. As the modulo function gives the remainder of division, an inverse for the modulo operator can no longer be defined. Hence information known about the inputs may not give as great a return in information on the output as with addition and multiplication. Similar problems can be observed in other surjective functions (e.g. trigonometric sin and cos). However, surjective functions do have the benefit that the output of the function is more restricted than the input (e.g. the function "$\%y$" is guaranteed to give an output less than y).

In a similar vein, bitwise logical operators lack the piecewise continuous nature of other functions described so far. Whilst bitwise operators are surjective, as is modulo, bitwise operators give additional information about not only their output but also their input. As bitwise operators are only defined for integers, one can infer that not only is the output of the function an integer, the input of the function is also an integer.

Hence to summarise, functions provide two forms of information. The first form of information relates to the outcome of the function; depending on the function and inputs, properties can be inferred on the output. The second form of information is on the input to the function: given that functions may not make sense for all input values, these restrictions can be reasonably placed upon the input values. The functions discussed in this section have been primarily concerned with the values of variables and finding properties on the outputs; the next section discusses functions used to control program flow, and how the evaluation of these functions may result in assumptions being placed on inputs.

5.1.4 Control and Compression

In addition to pure mathematical operators, many functions will be used to express the control flow of the program; examples of these are the less than function, or simply casting a value to a boolean. This class of functions return a boolean result, and this result is used to determine which path to take through the program.

In the event that it is possible to prove that the output of a comparison is fixed, regardless of input, then the analysis of these functions is trivial. Otherwise, it is necessary to consider each possibility. In this case the main problem to overcome is ensuring that the inferred properties of the relative values of variables remain consistent for the remainder of that paths analysis. This results in multiple paths to consider, each with differing inferred properties of variables, and hence the main problem when performing code analysis: state explosion.

To counter state explosion, it is necessary to implement a method of compression which can merge multiple paths back together. As stated before, variables which have been marked as irrelevant have no impact on the control flow of a program, because they are not used again. Similarly, properties inferred by an analysis may not be used again, in which case these proper-

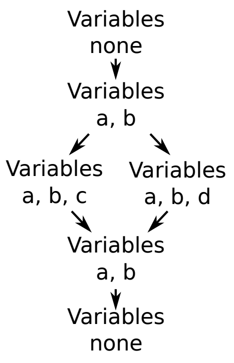


Figure 5.1: A simple CFG, illustrating that all variables, no matter their age, will eventually be discarded

ties can be removed as well. Hence, the value to the analysis of properties inferred on variables which have been marked irrelevant is zero, and these can be discarded without penalty. Similarly, variables which have fallen out of scope again have no impact on the control flow of a program and hence data on any the properties of these variables can also be discarded.

Following on from this, one observation of the effect of irrelevant variables is that as all variables will eventually fall out of scope, and therefore become irrelevant. Hence continuing to execute a particular path of the program will eventually lead to fewer relevant variables, and therefore less information will be discarded when lossy compression is employed. Even if new variables are introduced on the path, these new variables will eventually be discarded as illustrated in Figure 5.1. Unfortunately, continuing to execute a particular path of the program without performing merges will lead to further state explosion. A balance can be achieved by evaluating the path of the program which has the most information attached to it until this is no longer the case, and then checking for relevant merges. After any applicable merges have been applied, evaluation can resume on the next path, as determined by the amount of information known about that path. The amount of information known about a path under evaluation is trivially known by counting the number of properties that have been inferred on variables on that path.

A similar effect also to variables being marked as irrelevant also occurs on assignment: inferred properties can be lost. This will most typically occur on reassigning a variable, although not all properties need be lost when this happens. For example, when subtracting a known positive number y from

Property 1	Property 2	Merged Property
None	Any	None
$=$	$\neq$	None
$=$	$<, \leq$	$\leq$
$=$	$>, \geq$	$\geq$
$<$	$>$	$\neq$
$<$	$\geq$	None
$>$	$\leq$	None
$\leq$	$>, \geq$	None
$\geq$	$<, \leq$	None

Table 5.1: A commutative operator table for Merging properties

a variable x, properties stating a third variable z is equal or the less than x are no longer valid for $x - y$ and would have to be rechecked from the values that $x - y$ can take. However, variables which are greater than x will remain greater, and hence the property of being greater than $x - y$ remains valid. This can be achieved by using the transitive nature of the properties in question, as $x - y < x < z$, and hence $x - y < z$.

A final observation is that even with compression, the same code may be evaluated multiple times with different properties of variables. In this case it is possible to have fore-knowledge of the properties that may be tested by the code, as these will have been revealed by the previous evaluation. Hence any property that is not tested may be discarded early.

As already stated, the representation of values of variables picked has a simple merging operation. Hence the main task is in merging states is to be able to merge properties correctly. Fortunately, this is relatively trivial: properties inferred by operators governing the path through the control flow diagram will typically be using the operators $=, \neq, <, \leq, >, \geq$. As can be seen in Table 5.1, it is possible to merge these properties without losing information.

Given that it is possible to merge properties without losing information, it remains to see what information is lost when merging sets of potential values of variables and properties. The answer is combinations: when merging the potential values of multiple variables, combinations of variables that could not normally occur can be considered in the analysis. To an extent this may be possible to counteract given that the properties describing the

Information Type	Value	Merge Strategy
Variables		
Possible Values	High - Unsound if possible values not considered	Merge by encompassing both ranges
Excluded Values	Medium - Prunes search space	Merge by extending list, discarding if list becomes too long
Usage	Low - May be used to infer additional properties	Merge by assuming worst case
Program Control		
Program Counter	Critical - Required to be accurate	Only merge when identical
Properties	Medium - Prunes search space	Merge by using lossless merge operator
Discarded Variables	Medium - Prunes search space	Delete variables once no longer used
Combinations	Medium - Prunes search space	Merge by merging variables and properties in program states with the same program counter

Table 5.2: The types of information involved in Loop Bound Analysis

relationships between variables remain, but in general this should lead to a more pessimistic analysis.

5.1.5 Summary

The previous sections have discussed the value and overall merge strategies for the different types of information involved in loop bound analysis. These are summarised in the Table 5.2. Having given an overall outline, the next section will give examples of how this approach may be applied to actual programs.

5.2 Strategy Example

To aid in understanding the overall strategy of merging when information loss will be minimised due to variables falling out of scope, two small examples are presented here. Throughout this section, a branch of the analysis

```
1 def gt5(x)
2   if x > 5:
3     result = 1
4   else:
5     result = 0
6   del x
7   return result
8
9 analyse(gt5(Range(0, 10)))
```

Figure 5.2: A Trivial Example

will be referred to by the notation $\{pc, variables\}$ where pc denotes the current program counter and $variables$ denotes a fixed domain function from variable names to the range of values the variable can take, and the inferred properties made on the variables relation with other variables.

5.2.1 Trivial Example

Figure 5.2 details a trivial example. The function `gt5` simply checks if a value is greater than 5. The example requests an analysis of the `gt5` function with the input range 0 to 10. Hence, the analysis starts at line 1 with the state $\{1, (x = (0, 10))\}$. Obviously, as soon as the conditional on Line 2 is evaluated, the analysis must consider both possibilities. This splits the analysis so that there are two paths being considered, $\{2, (x = (5, 10))\}$, and $\{2, (x = (0, 5])\}$. As both of these contain the same number of variables and have evaluated the same number of instructions, the order in which they are evaluated is irrelevant.

Hence the analysis picks $\{2, (x = (5, 10))\}$ and continues evaluation. The variable *result* is created and assigned the value 1, before Line 6 is reached and the variable x is deleted. This causes the analyser to re-evaluate which path should be analysed. The states are now $\{2, (x = (0, 5))\}$ and $\{7, (result = 1)\}$. Whilst both contain the same number of variable, this time $\{2, (x = (0, 5))\}$ has the lower program counter.

Looking for a chance to re-merge the forked states, the analysis changes the state it is evaluating and proceeds from $\{2, (x = (0, 5))\}$. In this state, *result* is created with the value 0, before the analysis again arrives at Line 6 and deletes x, again causing the analyser to reevaluate the paths. In this

```
 1  def f(y):
 2    while y < 10:
 3      y *= y
 4    return y
 5
 6  def g(y):
 7    z = 0
 8    while z < y:
 9      z += 50
10    return z
11
12  def m(x):
13    x = f(x)
14    x = g(x)
15    return x
16
17  analyse(m(Range(2, 15)))
```

Figure 5.3: A Contrived, More Complicated Example

case, however, it sees that both states have the same program counter, and are therefore suitable for merging. Hence, the analysis merges both states to get $\{7, (result = [0, 1])\}$, and then continues. The function $gt5$ terminates immediately afterwards, with the analyser having explored all possible paths through the program, and finding that at most 5 lines of the program will execute.

5.2.2 While Loop Example

Figure 5.3 illustrates a more contrived example with a non-trivial program flow. The analyser is requested to analyse the function m with the range of values $(2, 15)$. Hence the analysis starts at Line 12. The first instruction calls function f. Importantly, the function initialises a new variable, and hence the state on entering f is $\{1, (x = (2, 15), y = (2, 15))\}$.

Upon encountering the loop on Line 2, the analysis must consider two paths, and hence splits the analysis to $\{2, (x = (2, 15), y = [10, 15))\}$ and $\{2, (x = (2, 15), y = (2, 10))\}$, where x is as before and the new variable is y defined inside the scope of the function f. As these contain the same number of variables, the order they are analysed is irrelevant. Picking $\{2, (x =$

$(2, 15), y = [10, 15))\}$ first, the analysis simply skips the loop and returns the value. On return, the variable y drops out of scope, and hence after reaching the state $\{14, x = [10, 15)\}$ the analysis returns to evaluate the state $\{2, (x = (2, 15), y = (2, 10))\}$, as it has more variables.

After executing the body of the while loop, the analysis again evaluates the while loop condition with the state $\{2, (x = (2, 15), y = (4, 100))\}$. This again results in a split to two states, with $\{2, (x = (2, 15), y = [10, 100))\}$ and $\{2, (x = (2, 15), y = (4, 10))\}$ being considered. As in the first case, the false branch $\{2, (x = (2, 15), y = [10, 100))\}$ is picked first and simply evaluated until y falls out of scope, at $\{14, x = [10, 100)\}$. As two states now exist with the same program counter, they are merged, resulting in $\{14, x = [10, 100)\}$, and the analysis goes back to consider the remaining state.

With the state $\{2, (x = (2, 15), y = (4, 10))\}$, the body of the while loop must be executed a second time, resulting in $\{2, (x = (2, 15), y = (16, 100))\}$. In this case, the condition of the while loop must succeed, and hence splitting the evaluation for a third time is unnecessary. Hence this state is evaluated until y falls out of scope, resulting in $\{14, x = (16, 100)\}$. As this state now has the same program counter as the previous state, they are merged, and once again the result of this is $\{14, x = [10, 100)\}$.

Continuing with the state $\{14, x = [10, 100)\}$, Function g is called, which creates two new variables in its scope, y and z. Trivially, the first encounter with the while loop on Line 8 will cause the loops body to be executed. Hence the state at the second encounter is $\{8, x = [10, 100), y = [10, 100), z = 50\}$. This results in a split, with $\{8, x = [10, 100), y = [10, 50), z = 50\}$ and $\{8, x = [10, 100), y = [50, 100), z = 50\}$ being considered. Picking the first path, this simply continues to evaluate until y and z fall out of scope at the return statement on Line 10, resulting in the state $\{15, (x = 50)\}$.

Returning to the previous state $\{8, x = [10, 100), y = [50, 100), z = 50\}$, the body of the loop is executed a second time, advancing the state to $\{8, x = [10, 100), y = [50, 100), z = 100\}$. This time, due to y being a half-open interval, the condition $z < y$ fails and the loop body is not executed. Hence the evaluation continues until the y and z fall out of scope on return, giving the state $\{15, (x = 100)\}$. The two states are then merged, giving $\{15, (x = 50 \text{ or } x = 100)\}$, and the evaluation terminates on the return of m.

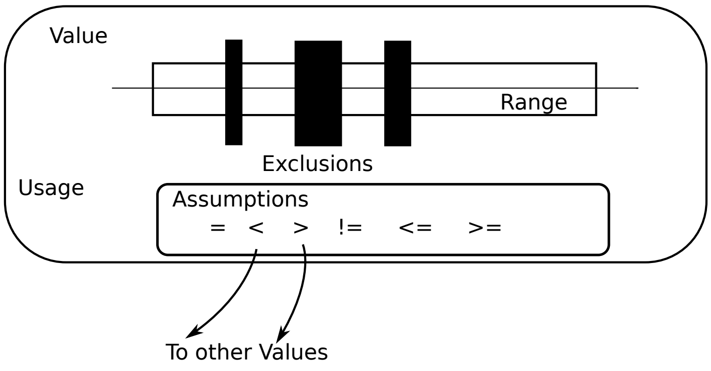


Figure 5.4: The information contained within a single abstract variable

Hence the analysis concludes that the possible return values of m are 50 or 100, that the loop inside function f executes at most 3 times, and the loop inside function g executes at most 2 times.

Having demonstrated the overall approach of the algorithm, it follows to give a detailed implementation.

5.3 Requirements for Implementation

Three types of information have been identified. These are as follows:

1. **Values**: The concrete values that variables can take, represented as ranges with exclusions and a usage.
2. **Properties**: Inferred properties made on the relative values of variables, needed to enforce constraints from the control flow of the program; represented as relations between variables.
3. **Combinations**: Which combinations of values and properties are valid at any one time; represented as the instances of variables belonging to separate paths.

An illustration of the components that represent the information needed to accurately represent an instance of a single variable is given in Figure 5.4.

In order for the analysis to be sound, the implementation must at least consider all feasible paths through the program. As detailed previously, Values have a sound lossless merge operator and an overly complicated value can have information discarded soundly. In a similar note, properties can be merged losslessly, but do not become more complicated with each merge. However, a large number of properties may cause a slowdown in evaluation speed; discarding properties for a speed up is sound, but may come with a corresponding loss in accuracy. Finally, combinations can be merged simply by merging the values and properties of each variable in the program paths to be merged, provided that these program paths correspond to the same point in the program as the program counter cannot be changed due to its criticality in program execution.

In order to counter state explosion whilst minimising loss of accuracy, the order of evaluation of different paths has to be picked carefully. As detailed previously, when variables are marked irrelevant the information contained in those variables has no impact on the code; this decreases the potential to lose information, as fewer variables have to be merged. Hence the order of evaluation is picked as evaluating the path of code which has the most information known until this is no longer the case. At this point any applicable merges are performed amongst paths which share the same program counter, and overly complicated state is soundly simplified and the process is repeated. In the case of ties with regard to amount of available information, earlier points in the program's execution are favoured to enable merging into later points.

As opposed to PLRU-ft, this approach is parametrisable in that the definition of an overly complicated state (where either the list of exclusion values or properties is long) can be defined by the user. Overly simplifying the states may result in an analysis that never terminates due to critical information being lost. Not performing sufficient compression on a complicated program carries the risk of the analysis taking too many resources to complete.

In terms of user interaction, the only annotations required are for the valid input values of the program. These should be well defined in any program, and hence can be assumed correct with high confidence. As each operation is implemented in the analysis to be sound and accurate on abstract values used, the information on input values is propagated through the program and no further annotations are required. However, care must be taken in that all combinations of values specified must also be valid; failure to observe this may result in an analysis which does not terminate.

Having specified the requirements, the next step is to construct an implementation. This implementation is given in the next section.

5.4 Implementation

One requirement which may be somewhat difficult to fulfil is knowledge about when variables drop out of scope. Whilst it would be possible to use scope information from a programming language, this information may not be as accurate as possible; variables may not be used, but still remain in scope. Whilst inaccurate information does not give unsound results, it will cause degraded performance as merges in the analysis will be postponed. As previously stated compilers generate information on when variables can be removed from scope in order to apply optimisations. Hence, more accurate information can be obtained by examining the intermediate code that a compiler generates, which should ideally represent all information that the compiler has obtained. As argued earlier, the drawbacks of losing high level information, such as easily obtainable locations for the beginning and end of complex loops, can be mitigated. Hence, this analysis targets intermediate code.

In particular, the implementation accepts the GNU InterMediate Programming Language (GIMPL), employed by the GCC compiler suite [34].GIMPL gives an easy representation of the basic blocks within a program, which can then be matched to the appropriate high level loops with minimal difficulty. Further, GIMPL clearly labels variables which can be dropped from scope before the end of a function call, and hence gives the information which is required for efficient analysis.

The analysis proposed is a form of symbolic execution [22]. Whilst symbolic execution is not novel, the analysis uses an efficient representation of variables combined with a non-standard evaluation order which guarantee as little valuable information as possible is lost. This in turn means that the only variables that require annotations are input variables, with other variables being inferred by propagated information. As the valid values for input variables are presumably known for a given system, this significantly reduces the chance of human error affected the validity of the analysis.

5.4.1 Representation

As discussed in Section 5.1.2, at least 3 components are required for the representation of a value: a range of possible values, exclusions, and a flag denoting the usage of the variable. This implementation implements exclusions in two ways; in addition to lists of ranges which a value cannot be, a "divisible by" value is used. This is a specific optimisation to deal with one common use case: calculating a memory address using pointer arithmetic. In the case that one operand for a multiplication operator is a constant, single value, the "divisible by" value can be used. It is a reasonable assumption that the size of memory structures is known and fixed in cases where pointer arithmetic is appropriate. This knowledge enables the "divisible by" value to be used and exclude any values which may be in the prescribed range of a given pointer, but would be impossible to encounter. In the example of pointer arithmetic, this would exclude values that are invalid, and may give inputs which are not in the specification of the system. An example of this is an integer array containing (x, y) coordinates; the index of any valid (x, y) coordinate must be divisible by 2; hence storing this constraint gives additional accuracy by excluding index that result in a (y_n, x_{n+1}) pair.

Hence, the representation used is formally given as a list of four items:

- **r** Range: The minimal range in which all possible values lie
- **e** Exclusions: A list of ranges which are excluded
- **d** Divisibility: A single value which any concrete value must be divisible by
- **u** Usage: If the value is used as a real or discrete value

Of note is that this representation does not track all possible information about concrete values. For example, the sign and absolute value of a variable is not explicitly tracked. This means that some calculations may not be accurately represented; for example, multiplying the range (-1, 1) by -1 gives the range (-1, 1). This may give the impression of a no-op, but this is only the case for the value 0; any other value in the range will be inverted. Hence, for all other values, information has been lost, although as all values are still considered, such information is not particularly useful.

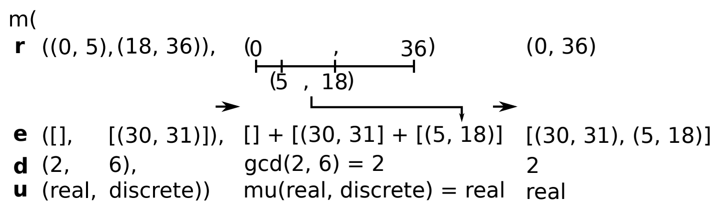


Figure 5.5: An example of merging two states

Merging the representation of such variables is accomplished by the following operation:

$$mu(u_0, u_1) = \begin{array}{l} \text{discrete if } u_0 = u_1 = \text{discrete} \\ \text{real otherwise} \end{array} \tag{5.1}$$

$$\begin{aligned} m((r_0, e_0, d_0, u_0),& \\ (r_1, e_1, d_1, u_1)) &= (r_0 \cup r_1, e_0 + e_1 + [((r_0 \cup r_1) - r_0) - r_1], \\ &\quad gcd(d_0, d_1), mu(u_0, u_1)) \end{aligned} \tag{5.2}$$

Where the union of two ranges is defined as the smallest contiguous range encompassing both ranges. Similarly, subtracting ranges is defined by excluding a smaller range from a larger range; hence the expression $((r_0 \cup r_1) - r_0) - r_1$ refers to the range of values between the ranges r_0 and r_1. Finally, the function *gcd* refers to the greatest common denominator function, which simply finds the greatest value which divides all operands; obviously, this is the best value possible for merging the divisibility constant of a number. A worked example of the merge operator is given in Figure 5.5.

Inferred properties are given as relations between values. In order to minimise the amount of information stored, the only properties stored are equality, less than, and their negations. As stated in Chapter 3.1, it is advisable to perform lossless compression before any lossy compression is attempted. This is accomplished here by observing that a property such as $x > y$ can be equivalently represented as $y < x$, and hence it is unnecessary to store information on both less than and greater than properties.

Having defined a representation, it is necessary to define how the basic functions which utilise this representation work.

5.4.2 Functions

Binary comparisons are the first class of function that will be examined. These functions are typically used for the control flow of a program, and hence are important to be accurately modelled. The binary comparisons all operate in a similar manner. First, they check the properties on the variables being compared. If properties have been inferred which specify that the binary comparison should succeed or fail, this result is used. If not, the variables are compared based on their possible values, and every combination possible is considered individually. This results in at most 4 separate values: 2 represent the intersection of ranges (where the comparison may or may not be true) and 2 represent the mutually exclusive area of the ranges (where the comparison will or will not be true). However, in the case of exclusive ranges, this will result in only a single value, as expected. Importantly, each of the possible results will place additional properties on input values whilst the results of that comparison remain valid.

The standard mathematical operators $+, -, \times, \div, \%$ are implemented on ranges in the obvious manner; all values are adjusted by the appropriate amount and operator. As is the case with merging two states, this can cause the number of excluded ranges to grow, and this effectively causes lossy compression: if the number of excluded ranges exceeds a user supplied constant, then excluded ranges are discarded in order of size, from smallest range to largest. This is because as the smaller ranges represent fewer values, discarding these ranges discards less information on infeasible values than discarding a larger range which represents more values.

To handle the divisibility component of a variable, which represents the greatest number that all valid values of the variable must be divisible by, then multiplication and division by a single value operators modify this as expected (by multiplication/division as appropriate). In the case of a range of values for multiplication/division, as well as addition/subtraction, the divisibility component of the result is the greatest common divisor of the divisibility components of the two input variables, as this is the largest single value that can be tractably found which is guaranteed to divide the result. In the case of modulo, it is not possible to infer a divisibility relationship with the output, and hence information on divisibility is lost.

Finally, the usage component simply tracks if both operators are discrete; if they are then the result is also discrete. Otherwise, it is treated as a real value, as real values make fewer assumptions than discrete values.

Where possible, additional properties on the results in relation to the input values and operator are inferred. Two examples of this are as follows: multiplying two values strictly greater than one or adding two positive values produces an answer greater than either input, which results in an additional property that can be enforced. These properties may not be obvious if not explicitly enforced at this stage; adding the ranges $(1, 3)$ and $(1, 4)$ gives the result $(2, 7)$, which would otherwise appear to overlap both input values.

Some bitwise operators present more of a challenge. Trivially, bitwise operations are only valid on discrete values, as it makes little sense to use bitwise operators on floating point values[1]. Also trivial are the bitwise shift operators, which can be represented as multiplication and integer division operations respectively. Finally, the **not** operator is also trivial as it can be represented as $!= 0$, which has already been defined.

Other bitwise operations, specifically **and**, **or**, and **xor** are more complicated. It is necessary to check whether each bit could possibly be set in the range of the variables, and apply this to the result. However, some optimisations can be applied in the common cases where one of the inputs is known. For example $x\&0xff$ is equivalent to $min(0xff, x)$ for non-negative x and $max(0, 0xff - x)$ otherwise. These optimisations, where applicable, significantly reduce the computational time of these functions. As with the more standard mathematical operations, it is possible to apply properties on the outputs relative to inputs. Again using **and** as an example, $x\&y$ is less than or equal to both x and y.

Finally, having defined the mechanisms by which functions work, it is necessary to define how the program is evaluated.

5.4.3 Evaluation Order

Initially, the program starts from a single known state, with ranges for input values supplied by the user. This changes once a binary comparison used for the control of the program indicates multiple possible values, which causes it to become necessary to consider multiple paths.

As stated in Section 5.1.4, properties can be merged without losing infor-

[1]Barring the possibility of conversion to a different floating point format, which this analysis has implicitly guarded against by assuming all floating point values are analogues of real numbers.

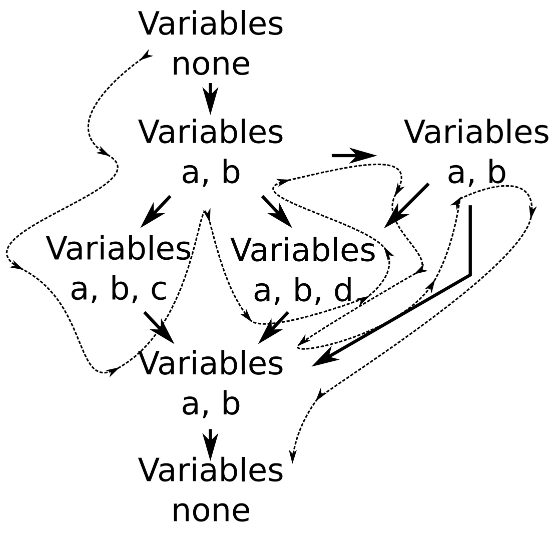


Figure 5.6: A non-trivial visiting order. The dotted line indicates the order in which nodes of the CFG are evaluated

mation. Hence the only consideration is when information is lost from the possible values of variables. As stated, once variables have been marked as irrelevant, there is less information to compress, and hence less information to lose. Hence the evaluation strategy is to execute the path of the program for which the greatest number of variables are currently relevant; once this stops being the case, conduct any applicable merges and resume evaluation with the new path which has the greatest number of variables valid.

In practice, this can be likened to a breadth-first search, as far as such a notion can be defined on a CFG. As illustrated in Figure 5.1, for simple programs this can be easy to see. However, Figure 5.6 illustrates that on more complicated programs the order of evaluation is non-trivial.

Having defined the properties of the algorithm, the next step is to test its performance.

5.5 Overview of Algorithm

To provide a pseudocode overview of the algorithm, the first functions to define deal with how to merge states under consideration. The resulting state must be an upper bound. This is handled by the Functions MergeCFG and Merge.

```
Function MergeCFG($cfg_1$, $cfg_2$)
    $cfg' \Leftarrow cfg_1$
    for edge $e' \in cfg'$ do
        $e_2 \Leftarrow$ the corresponding edge of $e'$ in $cfg_2$
        if $label(e_2) > label(e')$ then
            $label(e') \leftarrow label(e_2)$
    end
    return $cfg'$
end
```

Function MergeCFG(cfg_1, cfg_2), which returns an upper bound of the annotations of cfg_1 and cfg_2

```
Function MergeCFG($cfg_1$, $cfg_2$)
    $cfg_1, pc_1, \mathbf{variables_1} \Leftarrow \mathbf{state_1}$
    $cfg_2, pc_2, \mathbf{variables_2} \Leftarrow \mathbf{state_2}$
    $cfg' \Leftarrow MergeCFG(cfg_1, cfg_2)$
    $\mathbf{variables'} \Leftarrow \{\}$
    for $variable_1$ in $\mathbf{variables_1}$ do
        $variable_2 \Leftarrow$ the matching variable to $variable_1$ in $\mathbf{variables_2}$
        $variable' \Leftarrow m(variable_1, variable_2)$ where $m$ is as defined in
        (5.2)
        $\mathbf{variables'} \Leftarrow \mathbf{variables'} \cup \{variable'\}$
    end
    return $\langle cfg', pc_1, variable' \rangle$
end
```

Function Merge($\mathbf{state_1}$, $\mathbf{state_2}$), a function which provides an upper bound for two abstract states which have the same program counter

Having defined the merging functions, it follows to give an overview of the entire algorithm. This is given in Function BoundCFG, which takes the control flow graph of the program and annotated input variables, and provides an annotated control flow graph as output. This is accomplished by evaluating the statements of the program with respect to the input variables, and merging as appropriate.

```
1  Function MergeCFG($cfg_1$, $cfg_2$)
2      $cfg_{annot} \Leftarrow cfg_{unannot}$
3      for edge $e' \in cfg_{annot}$ do
4          $label(e') \Leftarrow 0$
5      end
6      $states \Leftarrow \{(cfg_{annot}, 0, \mathbf{in})\}$
7      while $states \neq \emptyset$ do
8          currentState $\Leftarrow$ x where x is picked from states such that no
           other state in states contains more information than x
9          remove currentState from states
           $cfg, pc,$ variables $\Leftarrow$ currentState
10         $pc_{old} \Leftarrow pc$
11         evaluate basic block $pc$ with variables, returning a set of
           successor states representing all possible outcomes of basic
           block $pc$ with variables
12         for each outcome $pc', variables'$ do
13             $cfg' \Leftarrow$ a copy of $cfg$
14             increment the label on the edge between basic blocks $pc_{old}$
               and $pc'$ by on $cfg'$ by 1
15             add $(cfg', pc', variables')$ to states
16         end
17         for each pair of states, $\mathbf{state_1}, \mathbf{state_2}$ with the same pc in
           states do
18             remove $\mathbf{state_1}, \mathbf{state_2}$ from states
19             add $merge(\mathbf{state_1}, \mathbf{state_2})$ to states
20         end
21     end
22 end
```

Function BoundCFG($cfg_{unannot}$, **in**), which takes an unannotated cfg and annotated input variables, and produces a cfg annotated with bounds on the maximum number of times each edge can be executed.

Benchmark	No of Input States	States Explored	WCET by analyser	WCET by exhaustive exploration	Pessimism
`fibcall.c`	30	7946	253	253	1.0
`insertsort.c`	10!	214864036	749	749	1.0
`prime.c`	$\approx 10^{13}$	2732670	5054	5054	1.0
`matmult.c`	$O(2^{20^4})$	188879	188879	188879	1.0
`janne_complex.c`	25	Did not terminate			
`recursion.c`	25	Did not terminate			

Table 5.3: Selected results from the Mälardalen Benchmarks; Full Results are in Appendix B

5.6 Evaluation

For evaluation, the Mälardalen benchmarks [35] were used. Initially, they were used unmodified, in single path form. However, in this form all benchmarks that could be analysed returned correct loop bounds, regardless of the programs structure or lack thereof. Hence, the evaluation considered a modified form of the Mälardalen benchmarks, where input parameters were changed to ranges. This allowed the analyser to be tested with regards to handling variable input. A worst case was then computed by hand and the results compared to compute the ratio between the hypothetical worst case the analyser found and the actual worst case, creating a pessimism score. A pessimism score of 1 is a perfect answer, whereas 2 would present a 100% pessimism penalty, and so on. To evaluate performance, the number of instructions evaluated was compared with both the worst case alone and the number of instructions evaluated should no merges take place.

Of note however, is that a number of Mälardalen benchmarks failed to run under the analyser (Full results provided in Appendix B). The reasons for these failures ranged from undocumented GIMPL features which the tools were unable to interpret to programs failing the strict assumptions of correctness. For example, the `qsort-exam` benchmark makes the assumption that the compiler allocates space before an array, such that an element loaded with the index -1 is zero. This fails the strict requirement of correctness required by this approach.

Selected results are given in Table 5.3, with full results given in Appendix B. As can be seen, the results fall into two categories. Where the compression is appropriate and does not discard important information, a perfect score is achieved. If the compression is inappropriate, the analysis fails to terminate hence does not produce any answer at all. The reason for this disparity could be caused by one of two things. In the case of the `janne_complex.c` benchmark, memory usage becomes stable. This suggests that for the input ranges given, one combination of input values gives an infinite loop. In the case of `recursion.c` however, memory usage is not stable. This suggests important information has been discarded due to the approximation of real and discrete numbers. Given the lack of user annotations, the analysis has no way of recovering the information lost to this approximation and hence has insufficient information to prove any bound. If user annotations were supplied, this information could be recovered, albeit at the expense of trusting that the user annotations supplied were indeed accurate.

For the results that do terminate, the complexity of the analysis does vary. This is mainly due to the varying structure of the programs, and how the input values impact the paths the program can take. For example, whilst the input state space for the `matmult.c` benchmark is huge, the input of a 20x20 matrix does not effect the path taken through the program, and hence the number of evaluated instructions is comparatively low. This is not the case in the `insertsort.c` benchmark on 10 elements; insertion sort is highly dependent on the ordering of the input variables, and hence a comparatively high number of states must be explored in relation to the size of the input state space.

Examining the results in further detail produces an annotated control flow graph for the program, with each annotation indicating the number of times that path can be taken within a single function call, at the basic block level. An example of this output is given in Figure 5.7. From such a CFG it is trivial to extract loop bounds.

An obvious improvement on the analysis would be to attempt to determine if the analysis will not terminate. Whilst this cannot be accomplished in every case due to the halting problem, some approaches may be able to detect some common cases. In the case of the `janne_complex.c` benchmark previously mentioned, it would be sufficient to check that the analysis did not become trapped in a loop of previously visited states. Note that this improvement does not improve the quality of results other than attempting to ensure that the results actually exist, rather than assuming the absence of a result after a sufficient amount of computation means failure.

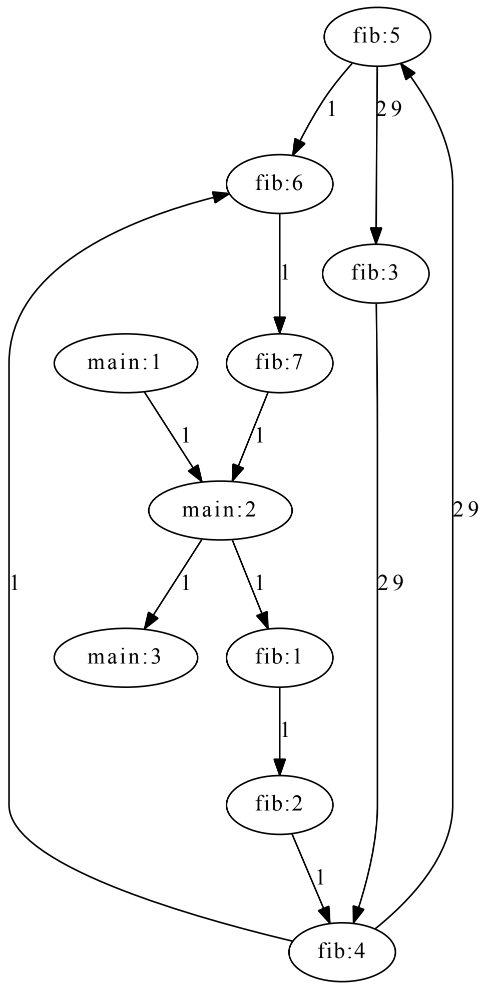


Figure 5.7: An annotated CFG for the `fibcall.c` benchmark. Labels on nodes are of the form "*function name*:*basic block number*", and labels on edges denote the maximum number of times that edge may be taken.

Comparison with other approaches is difficult, as other approaches [100, 112] make use of annotations. The lack of annotations in this approach means that the problem is fundamentally different: whereas other approaches require additional input to find accurate loop bounds, this approach is primarily focused on preserving information that can be logically inferred. However, the primary benefit of the approach outlined is the fact that the information required is much less susceptible to human error. The main downside is in the execution of the algorithm, in that it is not guaranteed to terminate, whereas other approaches are.

5.7 Summary

This chapter has outlined a different approach to loop bound analysis, which almost entirely removes the need for annotations to code. The advantage of this is that annotations are potentially susceptible to human error, a situation which is not easily improved upon [81]; with no annotations, the chance for error is minimised. However, this approach is not without its difficulties. Specifically, the compression scheme outlined here is not suitable for all programs; some properties are not tracked, and if a program's control flow relies on one of these properties then the analysis will fail to terminate.

On the other hand, the compression scheme could be varied. There is no reason why the tracked properties could not be changed. If done correctly, this would change the programs that can be analysed without significantly increasing the time taken to perform the analysis. As this would still preserve the property that annotations are not required, varying the compression used may be a more suitable method for analysing high integrity systems than relying on human annotations.

Chapter 6

Conclusions

This Chapter gives an overview of the thesis. Section 6.1 draws together the themes of the thesis, and presents an argument that the overall techniques used are an extension of the current state of the art which is broadly applicable to other problems where Abstract Interpretation or Symbolic Model Checking are used in Worst Case Execution Time Estimation. Section 6.2 presents some of the limitations of the work in this thesis, with ideas for further research. Finally, Section 6.3 revisits the original research question and goals, to determine the success or otherwise of each and presents some concluding remarks.

6.1 Summary of Contribution

This thesis began by examining current techniques in Chapter 2. Current techniques all approach the intractable problem of Worst-Case Execution Time estimation by creating a model of some kind, be it based on measurements as from Measurement-Based Techniques (Section 2.2.2) or approximation in Static Techniques (Section 2.2.3). The creation of a model is necessary as the systems involved are complicated, and cannot be analysed by exhaustive techniques in any reasonable time. Hence the creation of a simpler model is necessary which is constructed to provide a sound bound on the execution time of a program, typically by means of overapproximation.

Focusing on Static techniques, the main approaches used are based on Abstract Interpretation [36] or Symbolic Model Checking [22]. Both of these techniques seek to use some form of approximation in order to reduce the

1. *Types of Information*: Determine the alphabets used to compactly represent information.
2. *Value of Information*: Using logical argument and/or experimentation, determine the value of each type of information.
3. *Overall Strategy*: Decide which information should be discarded during the merging of states.
4. *Representation*: Pick an appropriate representation to enable the desired information to be discarded.
5. *Merge Operator*: Formally define a lossy merge operator.
6. *Recovery Strategy*: If necessary, determine how to soundly resolve information which has been discarded.

Figure 6.1: A 6-step process to create a custom compression method

number of states that must be considered during analysis. However, neither approach details how such an approximation can be found, instead leaving it to researchers to identify useful approximations. When a useful approximation is found, for example the approximation of LRU caches by two lists [98], these techniques are immensely powerful. Unfortunately, such approximations are hard to find, especially given the fact that there is no method proposed for examining or quantifying the information that is approximated. This results in approximations which have limited scope being proposed, for example the Potential-Leading-Zero (PLZ) approximation of a PLRU cache; while adequate for the Must analysis of a 4-way cache, the PLZ approach cannot fully analyse an 8-way cache.

To propose a solution to this, the thesis introduced the ideas of Lossy Compression in Chapter 3. At the heart of the approximations used by Abstract Interpretation and Symbolic Model Checking, the desire is to discard data from a representation which is of low value to the end result. In doing this, the first problem encounter would be a lack of appropriate language to discuss the quantity, value and storage of information in analysis; Information Theory (Chapter 3) provides this language. This is enhanced by lossless compression algorithms (Section 3.1), which provide techniques to achieve highly efficient representations of data which do not lose any information.

Unfortunately, while Lossless Compression can provide an aid, the normal methods of Abstract Interpretation and Symbolic Model Checking are used precisely because without using approximations the problems of WCET estimation are intractable. Hence it is necessary to look at Lossy Compression methods (Section 3.2). Lossy Compression utilises the language and ideas of Information Theory in combination with scientific methods to determine information that is probably of low value. For example, audio codecs [18, 16] rank the frequencies they encode by how likely it is for the human ear to be able to perceive them. After this ranking is complete, information on frequencies that would not, in that instant, be perceptible is discarded first. This approach is precisely what the use of Abstract Interpretation or Symbolic Model Checking in the WCET is meant to accomplish , but rather than vague guidance provides a concrete method.

Having identified the main problems encountered in creating a lossy compression technique, the main contribution of Chapter 3 is a generic set of instructions for creating a custom lossy compression method for state based problems. When applied to such a problem, the resulting compression method can be used as the basis for either abstract interpretation or symbolic model checking, depending on which technique is more appropriate. This method is summarised in Figure 6.1 for easy reference.

To demonstrate that this method is broadly applicable to problems in WCET estimation, it was applied to two unrelated problems: PLRU/HN-MRU Cache Must/May Analysis in Chapter 4 and Loop Bound Analysis in Chapter 5. These problems are typical of WCET Estimation problems as the main challenge to overcome in each is the vast number of possible states to explore. A summary of the application of this method is presented in Table 6.1.

As can be seen in Table 6.1, the same process produces results on both problems, despite the problems not sharing many common features. In fact, the main common features to both problems are the size of the state space to be explored necessitating some form of sound approximation and that not all information is equally valuable. As previously stated, the goal of the generic approach to deriving a custom lossy compression is to be able to reduce the size of the state space while minimising the amount of valuable information discarded. In this regard, the approach is successful in both cases.

Hence, as the common feature of both problems are simply that the state

Step	PLRU-ft	Loop Bound Analysis
Types of Information	Identify pointers, tree structure, and cache lines	Identify value ranges, excluded values and assumptions on variables
Value of Information	Find that pointers are least valuable type of data	Find assumptions and excluded values are least valuable
Overall Strategy	Discard conflicting information on pointers	Discard data on excluded values / assumptions when too much data present
Representation	Store cache state such that pointers do not affect logical positions of cache lines	Store Values Ranges, Exclude values and assumptions separately
Merge Operator	Merge states which have the same cache lines and tree structure, destroying conflicting pointers	Discard Exclude Ranges / Assumptions if too many are present
Recovery Strategy	Consider all possible states	Consider all possible paths

Table 6.1: Comparing the application of using the generic process given in Figure 6.1 on PLRU and Loop Bound Analysis

space is too large for an exhaustive approach and that there exists some type of information which is less valuable than other types of information to the stated goal, it can be inferred that when these two conditions hold the use of lossy compression is appropriate. Specifically, the more information that is of sufficiently low value to the analysis there is, the greater the effect of lossy compression on improving the tractability of the analysis. This is a more specific version of the premise used in Abstract Interpretation and Symbolic Model Checking, where an appropriate approximation must be found. Therefore one can use the generic lossy compression approach as a more guided method for finding such an approximation in problems suspected to be amenable to Abstract Interpretation or Symbolic Model Checking.

6.2 Limitations and Further Work

This section details some of the limitations of the present work.

6.2.1 Cache Analysis: 16-Way Caches

Chapter 4 made passing reference to 16-way PLRU caches. However, due to the size of the state space involved; this is because, even though a 16-way PLRU cache is the next step from an 8-way PLRU cache, the state space is much greater. The scale of the problem becomes apparent in that analysing a simple loop of 16 memory accesses using PLRU-ft compression requires more than four gigabytes of memory with the current implementation. Hence it would be expected that in order to analyse a 16-way cache, either additional information on the starting state would have to be known, or alternatively additional compression must be used. If a known starting state is supplied, for example by flushing the cache before the invocation of each task resulting in an empty cache state, complexity is bounded by the amount of branching in the program and hence analysis is possible provided that branching is not excessively frequent.

Additional compression presents a challenge, in that the PLRU-ft compression already discards the least valuable information. Any further information discarded may result in state explosion if critical information is discarded. Hence the more relevant path of research may be to determine how much additional information is needed on the starting state for the problem to become tractable.

6.2.2 Cache Analysis: Sequential-Fill Caches

Similarly, Chapter 4 does not deal with Sequential-Fill PLRU Caches. As previously stated, a sequential fill cache has the additional rule that invalid cache lines are always selected for filling in a sequential manner before any evictions can take place. The complication that this introduces is that left and right subtrees of a PLRU cache can no longer be treated as equal, and hence the operation to order a cache state into a logical representation becomes problematic.

This problem could potentially be overcome by introducing a new type of information: the original position of each cache line in the cache. As this information is only used when inserting a new element into the cache when the cache contains an invalid line, it can be argued it is of low value. Hence, it becomes a candidate for compression. This would be implemented by a merge operator which produces cache states that list the possible original locations of each cache line. In the event of this information being used, all possibilities would have to be considered.

However, the problem with this approach is that it would inevitably increase the size of the state space. Hence it would be necessary to evaluate this approach to see if it remains tractable.

6.2.3 Loop Bound Analysis: Assumption of Program Correctness

The Loop Bound Analysis approach in Chapter 5 makes a very large assumption: that the program given is correct and will terminate for all inputs. Obviously, this is a desirable property for any real-time system. However, it may not be a property that can be guaranteed. In the event that a program does not terminate on all inputs, then the approach given cannot work, as it will fall into an infinite loop.

An obvious improvement to the approach would be to integrate some counter-measures against this. Simple counter-measures would be trivial to implement: specifically, it is assumed that the deadlines for a task are known in advance. With this knowledge, the problem can be rephrased to "will the input program halt in a specific time", and by integrating the loop bound analysis with timing analysis the problem once again becomes computable. However, the additional complexity may render it intractable. Other trivial approaches will not be able to find all faults, as this would violate the halting problem. A non-trivial improvement would be to present a real-time visualisation of the analysis. An appropriate visualisation would enable supervision, such that potential bugs could be found during the analysis, rather than the analysis not terminating.

6.2.4 "One-Size Fits All" Compression

One assumption made throughout this thesis is that "One-Size Fits All" Compression is appropriate. By this, it is meant that information deemed least useful by the analysis would always be found to be least useful, regardless of context. In PLRU cache analysis (Chapter 4), this appears to be the case. However, as seen in Chapter 5, it does not appear to be true for Loop Bound Analysis, where the given compression can result in important information being discarded.

This leads to the questions about using multiple compression techniques, and picking those which are appropriate to the situation. While Chapter 5 performed Loop Bound Analysis without any annotations, a non-standard type of annotation may be useful in this regard. Similar in how an automated theorem prover works, where rules are selected by a user, it may be useful to use annotations which pick appropriate compression methods at different points in the analysis. This supplies data on how the user believes the analysis should be carried out, as opposed to regular annotations which simply supply data to the analysis. This would result in a user guiding the application of techniques which are known to give sound results. In the case of failure on the part of the user, the analysis would fail to give a result or give a poor quality result; however any result obtained would be guaranteed to be sound. This can be argued to be preferable to the current situation of a user giving information that must be assumed to be correct.

6.3 Revisiting the Goals and Hypothesis

This section revisits the goals and hypothesis presented in Chapter 1, and presents closing remarks on each.

• Introduce Information Theory and Lossy Compression in the context of Abstract Interpretation for WCET estimation.

Chapter 3 introduced Information Theory and Lossy Compression in their respective normal domains. It then furthered this by examining specific examples of Abstract Interpretation for WCET estimation and arguing how these could be seen as a highly specialised form of lossy compression.

• Devise a general approach which applies lossy compression to devise a suitable model of a system.

Drawing from Information Theory, Lossy Compression and the specific examples of approaches already used, a general approach was presented in Chapter 3.6. This approach attempted to encapsulate all that was necessary to devise a specialised compression scheme for a given problem.

• Apply this approach to a problem for which there is presently no satisfactory solution: a complete PLRU cache analysis.

The approach was successfully applied to an tree-fill PLRU Caches of up to 8-ways. In addition, it was also adapted to the newer HNMRU cache variant, for caches with up to 12-ways. However, for PLRU/HNMRU caches with a higher number of cache ways, the algorithm produced is not tractable. Further, the case of sequential-fill caches was not addressed. However, for the caches for which the algorithm is appropriate, the results are comparable to using the collecting semantics [37] of a cache, but the approach has been shown to be up to 10 times faster.

• Apply the same techniques to the unrelated problem of Loop Bound Analysis, in order to reduce the amount of additional information required.

The approach was applied to Loop Bound Analysis, resulting in an algorithm which did not require user annotations on any variables apart from input variables. When this approach was successful, in the tests conducted no pessimism was observed. However, the approach was not always successful: in some cases, it was unable to find any answer. This was due to the fact that important information was lost in the compression. This results in two possible interpretations: the first being that in some cases, it may not be possible to find a universal compression method which always discards the least useful information. The second interpretation being that multiple, user-guided, compression methods may be necessary in some problems, such that the user selects between compression methods which are guaranteed to be sound rather than supplying data which may be unsound.

• Evaluate how the approach was used in both problems to demonstrate the broad applicability of this technique.

While the results of the specific compression method divided by the approach for Loop Bound Analysis were only partially successful, the method itself was applied successfully to both problems, despite their lack of similarity. Further, the problem faced in Loop Bound Analysis is different in that while some information is clearly critical, the value of the remainder is context specific: making a different choice on the value of types of information would have resulted in a different technique. Hence the overall approach appears to be sound, although the exact compression needed may depend on the problem under consideration.

6.3.1 Hypothesis

Problems encountered in finding appropriate models used in real-time systems problems, such as Worst Case Execution Time (WCET) Estimation can be thought of as a highly specific form of lossy compression. Hence, the approach used to design lossy compression algorithms can also be used to design an appropriate and effective model for use in WCET estimation.

This thesis examined current approaches to building models, and identified similarities with lossy compression methods.

The main contribution of this thesis was a generic approach to designing a lossy compression algorithm which can be used in conjunction with more traditional techniques such as Abstract Interpretation to produce an appropriate model. This has been demonstrated on two distinct problems, for which traditional techniques applied in isolation failed to produce adequate models, namely 8-way PLRU cache analysis (which was subsequently presented at [52]) and Annotation-Free Loop Bound Analysis.

The main problem encountered was the definition of value of information, as used in lossy compression. While in PLRU cache analysis it is relatively straightforward to find a least valuable type of information, in Loop Bound

Analysis the value of information can be context specific. However, the principal still applies that the least valuable information should be discarded.

Given the successful application of the generic lossy compression approach to building models, it seems justified to conclude that the problems examined were amenable to lossy compression. Further, due the lack of similarity between the two problems, other than common features found in many of the modelling problems faced in modelling real-time systems, it seems justified to conclude that the technique has wider applicability. While not in the scope of this thesis, further evidence for this has been demonstrated by subsequent research which applied the principles of lossy compression to the random replacement cache architecture [53].

Appendix A

Results of Full Tree Analysis on Benchmarks

Mälardalen Benchmarks

	PLRU-cs			PLRU-ft			HNMRU(2,4)-ft			HNMRU(4,2)-ft		
	Must	May	Miss	Must	May	Miss	Must	May	Miss	Must	May	Miss
adpcm	194975	121	16623	194975	134	16610	191920	18129	1670	200946	126	10647
bs	252	7	12	225	46	0	243	15	13	248	11	12
bsort100	641570	49398	49021	641373	69213	29403	641279	54537	44173	641280	87262	11447
cnt	8477	482	32	8305	672	14	8207	750	34	7917	1042	32
compress	12742818	67422	375479	12742815	67442	375462	-	-	-	-	-	-
crc	145544	498	7206	145542	532	7174	145300	1248	6700	145295	1255	6698
edn	450031	1092	48392	450031	1115	48369	449977	1135	48403	450028	1147	48340
expint	63554	897	1921	63554	927	1891	63455	1096	1821	63456	1092	1824
fdct	5710	2	838	5710	23	817	5710	0	840	5710	2	838
fft	108945	711	17541	108945	753	17499	108938	707	17552	108941	773	17483
fibcall	777	0	8	775	10	0	775	0	10	776	1	8
fir	6021	12	169	6021	26	155	6012	13	177	6021	14	167
insertsort	5893	47	34	5893	69	12	5885	54	35	5893	38	43
jfdctint	7008	1	738	7007	23	717	7007	2	738	7008	2	737
lcdnum	1161	94	189	1161	173	110	1161	103	180	1159	129	156
ludcmp	39066	395	3724	39066	473	3646	38943	497	3745	38930	651	3604
matmult	422628	2590	868	422088	3192	806	420919	4697	470	420955	5098	33
minmax	165	9	33	163	44	0	165	9	33	165	9	33
minver	4539	80	249	4536	100	232	4509	115	244	4490	128	250
ndes	121658	2631	17022	121658	2653	17000	121447	2988	16876	121560	2454	17297
nsichneu	19066	190	2753	19066	232	2711	19066	190	2753	19066	190	2753
prime	28063	11	745	28058	45	716	28057	14	748	28064	13	742
qurt	8456	235	1276	8456	253	1258	8456	235	1276	8456	240	1271
select	11134	189	843	11134	343	689	11134	232	800	11134	189	843
sqrt	2399	41	370	2399	164	247	2399	39	372	2399	41	370
st	410648	3387	38146	410647	3402	38132	414994	4028	33159	410157	4123	37901
statemate	297999	3291	81582	297998	3311	81563	297998	3291	81583	297999	3291	81582
ud	19367	152	1108	19367	287	973	19264	318	1045	19317	388	922

Papabench Benchmarks

	PLRU-cs			PLRU-ft			HNMRU(2,4)-ft			HNMRU(4,2)-ft		
	Must	May	Miss	Must	May	Miss	Must	May	Miss	Must	May	Miss
autopilot	8207384	101539	939134	8207383	101552	939122	8179708	110513	957836	8204398	170930	872729
autopilot.t10	8194685	101410	937423	8194682	101445	937391	-	-	-	-	-	-

	PLRU-cs			PLRU-ft			HNMRU(2,4)-ft			HNMRU(4,2)-ft		
	Must	May	Miss	Must	May	Miss	Must	May	Miss	Must	May	Miss
autopilot.t11	90	2	16	90	18	0	90	2	16	90	2	16
autopilot.t12	421	15	80	421	67	28	421	15	80	421	15	80
autopilot.t13	4658	38	697	4658	98	637	4658	38	697	4658	38	697
autopilot.t6	2704	33	434	2703	64	404	2701	36	434	2704	33	434
autopilot.t7	277	5	53	276	20	39	277	5	53	277	5	53
autopilot.t8	53	2	10	53	12	0	53	2	10	53	2	10
autopilot.t9	2945	18	334	2945	53	299	2945	18	334	2945	19	333
fly_by_wire	127901	9673	14628	127900	9692	14610	128433	9330	14439	127741	10110	14351
fly_by_wire.t1	888	17	152	888	36	133	888	17	152	888	17	152
fly_by_wire.t2	339	3	31	339	19	15	339	3	31	339	3	31
fly_by_wire.t3	369	7	65	369	24	48	369	7	65	369	7	65
fly_by_wire.t4	1203	224	26	1193	260	0	1218	213	22	1201	234	18
fly_by_wire.t5	1045	22	181	1045	65	138	1045	22	181	1045	22	181

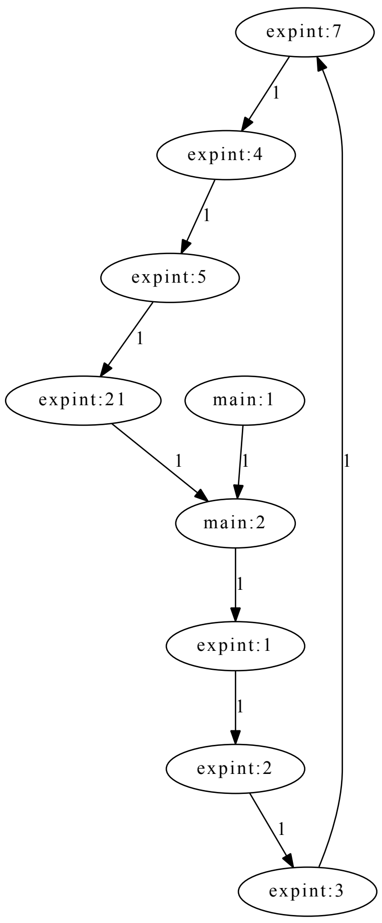
Appendix B

Results of Loop Bound Analysis on the Mälardalen Benchmarks

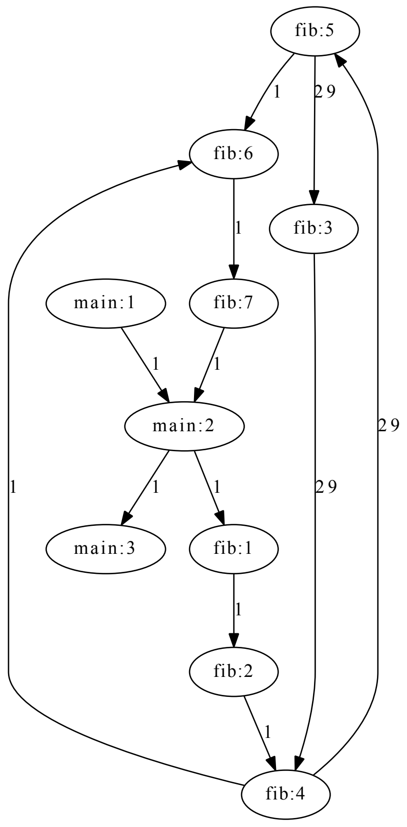
The following benchmarks were successfully analysed. The remaining Mälardalen benchmarks were unanalysable for technical reasons.

1. expint.c
2. fibcall.c
3. insertsort.c
4. janne_complex.c
5. matmult.c
6. prime.c
7. recursion.c

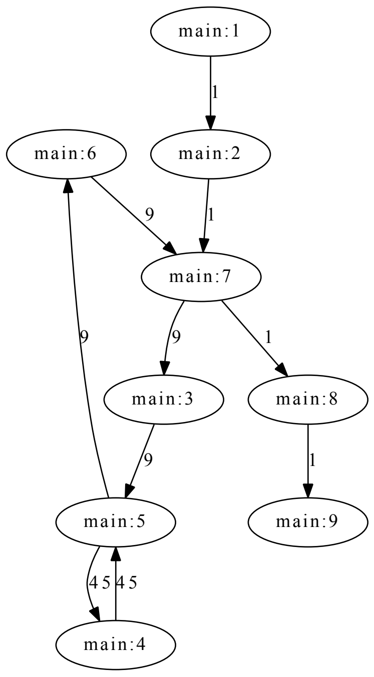
expint.c Size of Input State Space: 100. States Explored: 67. WCET Found: 67. Actual WCET: 67. Annotated CFG:



fibcall.c Size of Input State Space: 30. States Explored: 7946. WCET Found: 253. Actual WCET: 253. Annotated CFG:

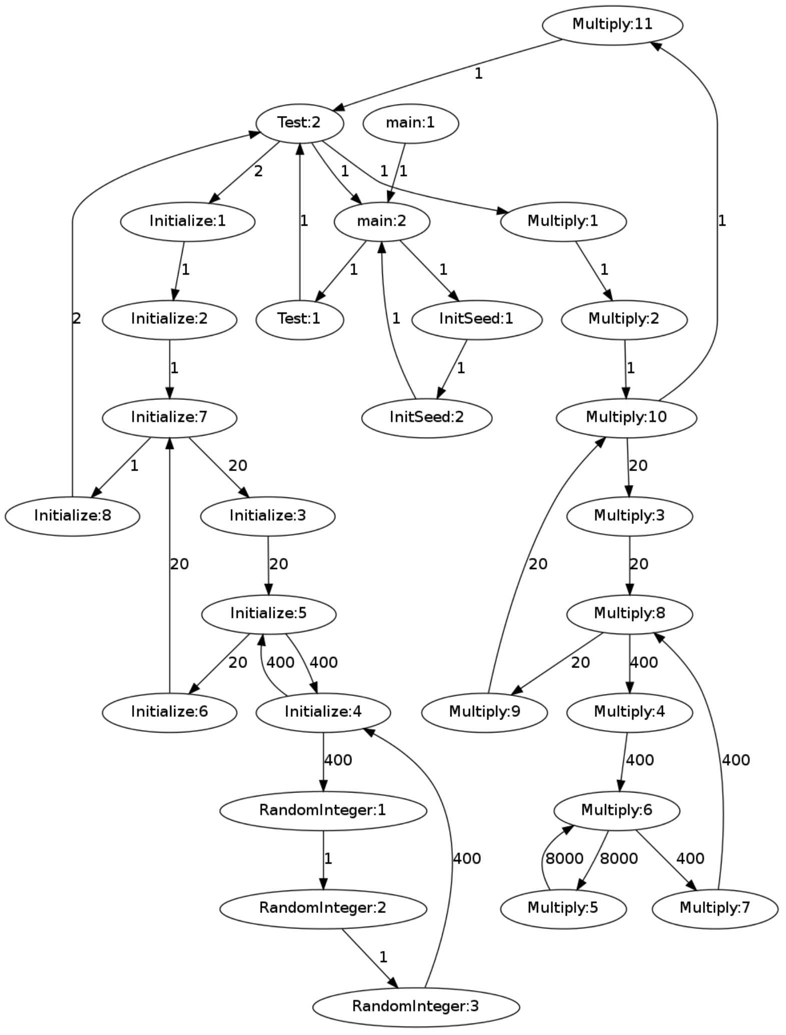


insertsort.c Size of Input State Space: 10!. States Explored: 214864036. WCET Found: 749. Actual WCET: 749. Annotated CFG:

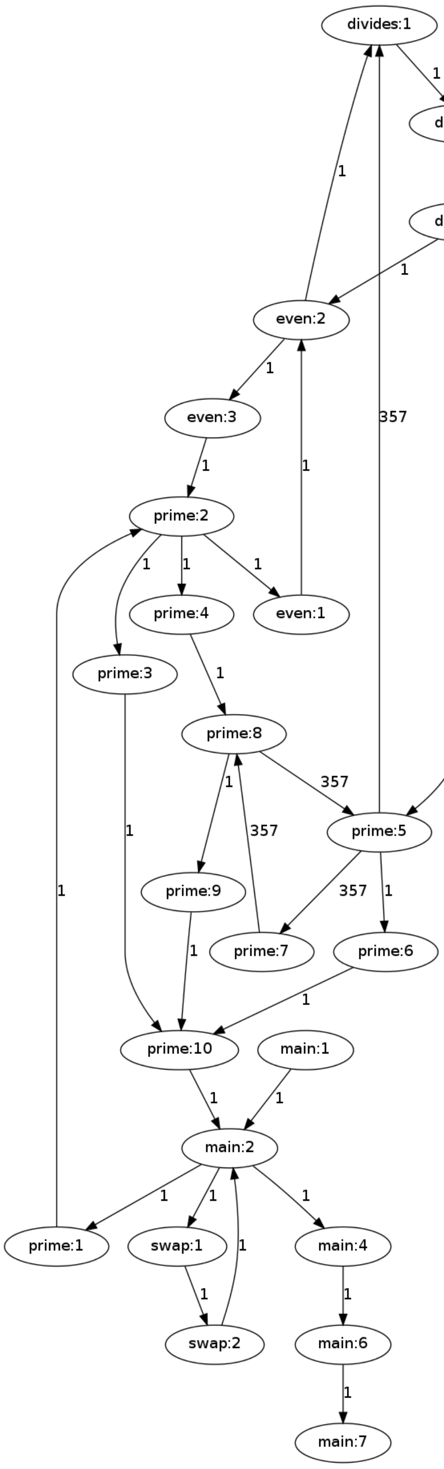


janne_complex.c Analysis did not terminate. As memory usage was stable, it appears that an infinite loop was encountered.

matmult.c Size of Input State Space: $O(2^{20^4})$. States Explored: 188879. WCET Found: 188879. Actual WCET: 188879. Annotated CFG:



prime.c Size of Input State Space: 1111111111111. States Explored: 2732670. WCET Found: 5054. Actual WCET: 5054. Annotated CFG:



recursion.c Analysis did not terminate. Memory usage was growing, so it appears that important information was discarded and the analysis unable to recover.

Appendix C

Digital Appendix

All program source code and experiments can be downloaded from the York Real Time Systems wiki which can be found at `http://www.cs.york.ac.uk/rts/`.

Bibliography

[1] N. Ahmed, T. Natarajan, and K. R. Rao. Discrete cosine transform. *IEEE Transactions on Computers*, 100(1):90–93, 1974.

[2] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42nd annual Southeast regional conference*, ACM-SE 42, pages 267–272, New York, NY, USA, 2004. ACM.

[3] F. E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, 1970.

[4] S. Altmeyer, R.I. Davis, and C. Maiza. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 261–271. IEEE, 2011.

[5] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. *YCS*, 164, December 1991.

[6] J. Bach and K. Ishizaka. Open Goldberg Variations. `http://www.opengoldbergvariations.org/`, 2012. Accessed on 1st September 2013.

[7] T. Badgett, T. M. Thomas, and C. Sandler. *The Art of Software Testing*. John Wiley and Sons, Inc, second edition, 2004.

[8] T. P. Baker and A. Shaw. The cyclic executive model and ada. *Real-Time Systems*, 1(1):7–25, 1989.

[9] S. K. Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-Time Systems*, 32(1-2):9–20, February 2006.

[10] C. Berg. Plru cache domino effects. In F. Mueller, editor, *6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum f'ur Informatik (IBFI), Schloss Dagstuhl, Germany.

[11] G. Bernat, A. Burns, and M. Newby. Probabilistic timing analysis: An approach using copulas. *J. Embedded Computer*, 1(2):179–194, 2005.

[12] G. Bernat, A. Colin, and S. Petters. WCET analysis of probabilistic hard real-time systems. In *Real-Time Systems Symposium, 2002. 23rd IEEE*, pages 279–288.

[13] G. Bernat, A. Colin, and S. Petters. pWCET: a tool for probabilistic worst-case execution time analysis of real-time systems. Techical report ycs-2003-353, January 2003.

[14] T. Boutell, T. Lane, M. Adler, C. Brunschen, A. M. Costello, L. D. Crocker, A. Dilger, O. Fromme, J. Gailly, C. Herborth, A. Jakulin, N. Kettler, A. Lehmann, C. Lilley, D. Martindale, O. Mortensen, K. S. Pickens, R. P. Poole, G. Randers-Pehrson, G. Roelofs, W. V. Schaik, G. Schalnat, P. Schmidt, T. Wegner, and J. Wohl. PNG (portable network graphics) specification. `http://www.w3.org/TR/REC-png-multi.html`, 1998. Accessed on 1st September 2013.

[15] V. Braitenberg. *Vehicles: Experiments in Synthetic Pysychology.* MIT Press, Jan 1984.

[16] K. Brandenburg and M. Bosi. " iso/iec mpeg-2 advanced audio coding: Overview and applications. In *AES 103rd convention, AES preprint*, volume 4641, 2012.

[17] K. S. D. Brandenburg. Ocf: Coding high quality audio with data rates of 64 kbit/sec. In *Audio Engineering Society Convention 85*, 11 1988.

[18] K. S. G. Brandenburg. ISO/MPEG-1 audio: A generic standard for coding of high-quality digital audio. *J. Audio Engineering Soc*, 42(10):780–792, 1994.

[19] M. Brudno. Proof of optimality of huffman codes. `http://www.cs.utoronto.ca/~brudno/csc373w09/huffman.pdf`, 2009. Accessed on 1st September 2013.

[20] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[21] S. Bullock and E. Silverman. Levins and the legitimacy of artificial worlds. In N. David, editor, *Third Workshop on Epistemological Perspectives on Simulation*, volume 13, page 14, 2008.

[22] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Infinite Computing*, 98(2):142–170, 1992.

[23] A. Burns and R.I. Davis. Mixed criticality systems - a review. Technical report, 2014.

[24] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison Wesley, fourth edition, May 2009.

[25] R. W. Butler and G. B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19(1):3–12, 1993.

[26] J. F. Cantin and M. D. Hill. Cache performance of SPEC 2000 CPU. `http://research.cs.wisc.edu/multifacet/misc/spec2000cache-data`, May 2003. Accessed on 15th August 2013.

[27] R. Chapman, A. Burns, and A. Wellings. Integrated program proof and worst-case timing analysis of spark ada. In *In Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'94*. ACM Press, 1994.

[28] K. Chernik, J. Lánskỳ, and L. Galamboš. Syllable-based compression for xml documents. In *Proc. Int'l Workshop on Databases, Texts, Specifications, and Objects, V. Snášel, K. Richta, and J. Pokornỳ, Eds*, pages 21–31. Citeseer, 2006.

[29] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[30] L. Colavito and D. Silage. Efficient pga lfsr implementation whitens pseudorandom numbers. In *Reconfigurable Computing and FPGAs, 2009. ReConFig '09. International Conference on*, pages 308 –313, dec. 2009.

[31] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2/3):249–274, 2000.

[32] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *13th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 37–44, 2001.

[33] Contributors. 7-zip lzma benchmark. `http://www.7-cpu.com/`. Accessed on 3rd March 2014.

[34] Contributors. Gnu compiler collection. http://gcc.gnu.org/. Accessed on 1st September 2013.

[35] Contributors. Mälardalen WCET benchmarks. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html. Accessed on 1st September 2013.

[36] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[37] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, 1992.

[38] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quinones, and F. Cazorla.

Measurement-based probabilistic timing analysis for multi-path programs. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 91–101, 2012.

[39] N. J. Cutland. *Computability: An introduction to recursive function theory.* Cambridge University Press, 1980.

[40] R. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35:1 –35:44, October 2011. DOI: 10.1145/1978802.1978814.

[41] P. Deutsch. Deflate compressed data format specification version 1.3. `http://tools.ietf.org/html/rfc1951`, 1996. Accessed on 1st September 2013.

[42] S. Edgar. *Estimation of Worst-Case Execution Time Using Statistical Analysis.* PhD thesis, University of York, 2002.

[43] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Proceedings of the Third International Euro-Par Conference on Parallel Processing*, pages 1298–1307, London, UK, 1997. Springer-Verlag.

[44] Nemer F., Cassé H., Sainrat P., Bahsoun J., and Michiel M. Papabench: a free real-time benchmark. In *In WCET '06*, 2006.

[45] H. Falk, P. Lokuciejewski, and H. Theiling. Design of a WCET-aware c compiler. In F. Mueller, editor, *WCET*, volume 06902 of *Dagstuhl Seminar Proceedings.* Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[46] C. Ferdinand and R. Heckmann. ait: worst case execution time prediction by static program analysis. In R. Jacquart, editor, *IFIP Congress Topical Sessions*, pages 377–384. Kluwer, 2004.

[47] 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. In T. A. Henzinger and C. M. Kirsch, editors, *EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer, 2001.

[48] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Program. Lang. Systems*, 9(3):319–349, 1987.

[49] A. Gersho and R. M. Gray. *Vector quantization and signal compression*, volume 159. Springer, 1992.

[50] R. L. Glass. Real-time: the "lost world" of software debugging and testing. *Commun. ACM*, 23(5):264–271, 1980.

[51] D. Griffin and A. Burns. Realism in statistical analysis of worst case execution times. In *10th International Workshop on Worst-Case Execution Time Analysis*, pages 49–57, July 2010.

[52] D. Griffin, B. Lesage, A. Burns, and R.I. Davis. Lossy compression for worst-case execution time analysis of plru caches. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 203:203–203:212, New York, NY, USA, 2014. ACM.

[53] D. Griffin, B. Lesage, A. Burns, and R.I. Davis. Static probabilistic timing analysis of random replacement caches using lossy compression. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 289:289–289:298, New York, NY, USA, 2014. ACM.

[54] D. Grund and J. Reineke. Toward precise PLRU cache analysis. In B. Lisper, editor, *Proceedings of 10th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 28–39. Austrian Computer Society, July 2010.

[55] J. Handy. *The cache memory book.* Morgan Kaufmann, Burlington, Massachusetts, USA, 2nd edition, Jan 1998.

[56] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

[57] N. Hillary and K. Madsen. You can't control what you can't measure, or why it's close to impossible to guarantee real-time software performance on a CPU with on-chip cache. In *2nd International Workshop*

On Worst-Case Execution Time Analysis, pages 45–48, Technical University of Vienna, Austria, June 2002.

[58] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098 –1101, sept. 1952.

[59] CompuServe Incorporated. Graphics interchange format programming references. `http://www.w3.org/Graphics/GIF/spec-gif89a.txt`, 1990. Accessed on 1st September 2013.

[60] N. P. Jouppi. Cache write policies and performance. `http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-91-12.pdf` retrieved on 26/10/2012.

[61] R. Kirner. The programming language wcetC. `http://www.wcet.at/~raimund/calc_wcet/intro_wcetc.pdf`, 2001. Accessed on 1st September 2013.

[62] R. Kirner and P. Puschner. Transformation of path information for WCET analysis during compilation. *Real-Time Systems, Euromicro Conference on*, pages 29–36, 2001.

[63] S. Kotz and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. Imperial College Press, 2000.

[64] M. A. Krasner. Digital encoding of speech and audio signals based on the perceptual requirements of the auditory system. Technical report, DTIC Document, 1979.

[65] R. Levins. The strategy of model building in population biology. *American Scientist*, 54(4):421–431, 1966.

[66] R. Levins. A response to Orzack and Sober: Formal analysis and the fluidity of science. *The Quarterly Review of Biology*, 68(4):547–555, 1993.

[67] Y. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *in Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 456–461, 1995.

[68] S. Lim, Y. H. Bae, G. T. Jang, B. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S. Moon, and C. S. Kim. An accurate worst case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, 1995.

[69] C. Liu, H. Hsu, and W. Lee. Compression artifacts in perceptual audio coding. *Audio, Speech, and Language Processing, IEEE Transactions on*, 16(4):681–695, 2008.

[70] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[71] A. Marref. *Predicated Worst-Case Execution-Time Analysis.* PhD thesis, University of York, 2009.

[72] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of loops. In K. Koskimies, editor, *CC*, volume 1383 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 1998.

[73] A. Metzner. Why model checking can improve WCET analysis. In *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 334–357. Springer Berlin / Heidelberg, 2004.

[74] V. Miller and M. Wegman. Variations on a theme by ziv and lempel (data compression). In *Communications, 1988. ICC '88. Digital Technology - Spanning the Universe. Conference Record., IEEE International Conference on*, pages 390 –394 vol.1, june 1988.

[75] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18:217–247, May 2000.

[76] R. Nelsen. *An Introduction to Copulas.* Lecture Notes in Statistics - Springer. Springer New York, 1999.

[77] J. Odenbaugh. The strategy of "the strategy of model building in population biology". *Biology and Philosophy*, 5:607–621, Nov 2006.

[78] S. H. Orzack and E. Sober. A critical assessment of Levins's the strategy of model building in population biology (1966). *The Quarterly Review of Biology*, 68(4):533–546, 1993.

[79] C. Y. Park. *Predicting deterministic execution times of real-time programs.* PhD thesis, Seattle, WA, USA, 1992.

[80] I. Pavlov. LZMA SDK. `http://www.7-zip.org`, 2001. Accessed on 1st September 2013.

[81] A. Prantl, J. Knoop, R. Kirner, A. Kadlec, and M. Schordan. From trusted annotations to verified knowledge. In N. Holsti, editor, *9th International Workshop on Worst-Case Execution Time Analysis, Dublin, Ireland*, volume 252. Austrian Computer Society, 2009.

[82] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1(2):159–176, 1989.

[83] P. Puschner and A. Schedl. Computing maximum task execution times - a graph-based approach. *Journal of Real-Time Systems*, 13:67–91, 1997.

[84] J. Reineke and D. Grund. Relative competitiveness of cache replacement policies. In *SIGMETRICS '08: Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 431–432, New York, NY, USA, June 2008. ACM.

[85] I. Richardson. White paper: H.264 / avc loop filter. `http://www.vcodex.com/files/H264_loopfilter_wp.pdf`, 2002. Accessed on 1st September 2013.

[86] S. Roy. H-NMRU: An efficient cache replacement policy with low area. *International Journal of Parallel Programming*, 38:277–287, 2010.

[87] RTCA SC-167, EUROCAE WG-12. *DO-178B / ED-12B: Software Considerations in Airborne Systems and Equipment Certification*, 1992.

[88] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper. Static timing analysis of real-time operating system code. In T. Margaria and B. Steffen, editors, *ISoLA*, volume 4313 of *Lecture Notes in Computer Science*, pages 146–160. Springer, 2004.

[89] M. Schoeberl and R. Pedersen. WCET analysis for a java processor. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 202–211, New York, NY, USA, 2006. ACM.

[90] M. R. Schroeder, B. S. Atal, and J. L. Hall. Optimizing digital speech coders by exploiting masking properties of the human ear. *The Journal of the Acoustical Society of America*, 66(6):1647–1652, 1979.

[91] R. Sen and Y. N. Srikant. WCET estimation for executables in the presence of data caches. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 203–212, New York, NY, USA, 2007. ACM.

[92] C. E. Shannon. A mathematical theory of communication. *Bell system technical journal*, 27, 1948.

[93] C. E. Shannon. Communication in the presence of noise. *Proceedings of the IRE*, 37(1):10–21, 1949.

[94] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989.

[95] W. Stallings. *Operating systems : internals and design principles.* Pearson/Prentice Hall, Upper Saddle River, N.J, 2005.

[96] L. J. Stephens. *Schaum's Outlines: Beginning Statistics.* McGraw-Hill, 2nd edition, 2006.

[97] J. A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, 1988.

[98] T. S. B. Sudarshan, R. A. Mir, and S. Vijayalakshmi. Highly efficient LRU implementations for high associativity cache memory.

[99] H. Theiling and C. Ferdinand. Combining abstract interpretation and ilp for microarchitecture modelling and program path analysis. In *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*, pages 144–153, Washington, DC, USA, 1998. IEEE Computer Society.

[100] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *Dependable Systems and Networks, International Conference on*, pages 625–632, June 2003.

[101] L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, November 2004.

[102] M. Usher. *Information Theory for Information Technologists.* Macmillan Publishers Ltd, 1984.

[103] J. Valin. The Speex Codec Manual, 2007.

[104] A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*, pages 429–528, London, UK, 1998. Springer-Verlag.

[105] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 239–243, 2007.

[106] J. Watkinson. *Compression in Video and Audio.* Focal Press, 1995.

[107] T. Welch. A technique for high-performance data compression. *Computer*, 17(6):8 –19, june 1984.

[108] J. Whitham and N. C. Audsley. Using Trace Scratchpads to Reduce Execution Times in Predictable Real-Time Architectures. In *Proc. RTAS*, pages 305–316, 2008.

[109] J. Whitham, R.I. Davis, N.C. Audsley, S. Altmeyer, and C. Maiza. Investigation of scratchpad memory for preemptive multitasking. In *RTSS*, pages 3–13, 2012.

[110] T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h.264/avc video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):560 –576, July 2003.

[111] R. Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In B. Steffen and G. Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 309–322. Springer, 2004.

[112] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embed. Computer Systems*, 7(3):1–53, 2008.

[113] R. Xu, D. Mossé, and R. Melhem. Minimizing expected energy consumption in real-time systems through dynamic voltage scaling. *ACM Transactions on Computer Systems*, 25(4), December 2007.

[114] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[115] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.