Static Probabilistic Timing Analysis of Random Replacement Caches using Lossy Compression

David Griffin
University of York, UK
david.griffin@york.ac.uk

Benjamin Lesage
University of York, UK
benjamin.lesage@york.ac.uk

Alan Burns
University of York, UK
alan.burns@york.ac.uk

Robert I. Davis
University of York, UK
rob.davis@york.ac.uk

ABSTRACT

The analysis of random replacement caches is an area that has recently attracted considerable attention in the field of probabilistic real-time systems. A major problem with performing static analysis on such a cache is that the relatively large number of successor states on a cache miss (equal to the cache associativity) renders approaches such as Collecting Semantics intractable. Other approaches must contend with non-trivial behaviours, such as the non-independence of accesses to the cache, which tends to lead to overly pessimistic or computationally expensive analyses.

Utilising techniques from the field of Lossy Compression, where compactly representing large volumes of data without losing valuable data is the norm, this paper outlines a technique for applying compression to the Collecting Semantics of a Random Replacement Cache. This yields a Must and May analysis. Experimental evaluation shows that, with appropriate parameters, this technique is more accurate and significantly faster than current state-of-the-art techniques.

1. INTRODUCTION

Static deterministic timing analysis aims to provide a single upper bound value for the Worst-Case Execution Time (WCET) of a task which can then be used in higher level schedulability analysis to determine if the task's timing constraints, usually expressed as a deadline, will *always* be met. However, in practice, for example in the avionics and automotive industries, such absolute guarantees are not required, rather, the failure rate must be below a certain threshold. For example, ISO-26262 requires that Automotive Safety Integrity Level (ASIL) D applications must have a failure rate below 10^{-9} per hour (below 10^{-7} per hour for ASIL A). Assuming that the maximum permitted failure rate for a task is 10^{-9} per hour, and the task is executed every 20ms, then this translates into a requirement that each invocation of the task can fail with a probability of no more than 10^{-13}, provided that such failures are independent of each other. In order to meet these requirements, analysis of *probabilistic real-time systems* [1, 2, 3] provides a means to quantify the probability of failure.

In order to achieve the necessary statistical properties, recent research [3, 4, 5] has focused on the use of randomised hardware, in particular caches with a random replacement policy. Approaches to *Static Probabilistic Timing Analysis* (SPTA) of the random replacement cache fall into two categories. The first, such as the reuse-distance metric of Davis et al. [5] is overly simplistic, resulting in high pessimism. The second, such as the focus block approach of Altmeyer and Davis [6], provides good accuracy, but is computationally expensive. Hence current approaches can be thought of as being extremes in the trade-off between accuracy and tractability. Ideally, the competing concerns of accuracy and tractability of analysis should be balanced, and effort only expended when it yields an increase in accuracy.

Lossy Compression [7] is a branch of information theory which outlines how one can identify a specific goal and then discard data which is of low value to that goal. Famously, lossy compression is used to great effect in audio/visual compression, for example MP3 [8]. Recently, Griffin et al. [9] used the principles of Lossy Compression to derive an accurate and efficient analysis technique for the Pseudo Least Recently Used (PLRU) cache, demonstrating how lossy compression can be applied to collecting semantics [10] to yield a new analysis technique. This paper applies the same principles to the collecting semantics of the random replacement cache, as defined by [6], to yield a full Must/May analysis [11].

RTNS 2014 , October 8 - 10 2014, Versailles, France
Copyright 2014 ACM 978-1-4503-2727-5/14/10 ...$15.00.
http://dx.doi.org/10.1145/2659787.2659809.

1.1 Related Work

Static Probabilistic Timing Analyses (SPTA) [2, 12, 3, 4, 5, 6] use a model of the system architecture and information from the code of the task under analysis to derive their results. SPTA methods have the goal of producing a probabilistic Worst-Case Execution Time (pWCET) distribution for the task. (See [13] for a discussion of the difference between pWCET and a probabilistic execution time (pET) distribution). To accomplish this, methods for the analysis of cache [5, 6] produce a Probability Mass Function (PMF) for the number of cache hits and misses for a given sequence of memory accesses. Such a distribution can be coupled with the hit and miss latencies of the cache to give the overall contribution of the cache accesses to the pWCET of the task.

Recently, Davis et al. [5] produced a simple analysis for random replacement caches, which takes into account the problems of dependencies between accesses due to the finite size of a cache. Subsequently, this approach was proved to be optimal [6] with respect to the limited information it used (cache associativity and reuse distance).

In 2014, Altmeyer and Davis [6] introduced an enhanced analysis which uses an additional high-accuracy mode, based on collecting semantics. A set of *focus blocks* are identified, selected by a heuristic that identifies blocks with frequent accesses. These focus blocks are analysed using collecting semantics, with any non-focus blocks represented as unknown memory accesses. For non-focus blocks, a lower-accuracy mode, is employed based on re-use distances and the concept of cache contention.

In the context of static deterministic timing analysis, the terms Must and May analysis were first used by Mueller [11]. A *Must Analysis* determines which accesses to the cache are guaranteed to be hits. A *May Analysis* is the inverse, determining which memory accesses are guaranteed to be misses. These can be seen as providing upper and lower bounds on the cache state respectively. A May analysis becomes important when timing anomalies [14] are considered, as locally maximal execution times may not feature in the true worst-case.

Recently, Griffin et al. [9] introduced lossy compression for static deterministic timing analysis and demonstrated its use by providing a Must/May analysis for the PLRU cache. The PLRU cache is typically hard to analyse due to the complex behaviour it exhibits as an approximation of the LRU cache. Previous attempts at PLRU cache analysis [15] only provided partial analysis. However, by reasoning about the types and value of information within each PLRU cache state, Griffin et al. provided an efficient and accurate cache analysis. This result demonstrates that lossy compression is a useful technique to apply in determining suitable approximations for use in static analyses, especially in handling non-trivial behaviours.

1.2 Organisation

Section 2 begins with an overview of the collecting semantics of the random replacement cache. This is then extended via lossy compression techniques: a notion of uncertainty is introduced which is used to implement compression via multiple parameterisable methods. Examples of the compression methods are given in Section 3. Section 4 takes the ideas presented in Section 3 and constructs a formal definition of the analysis. Section 5 examines performance in relation to existing solutions, in terms of both the accuracy of the results and the runtime of the analysis. Finally, Section 6 summarises the work and presents conclusions regarding the benefits of the new approach and possible future work.

1.3 Notation and Assumptions

To represent PMFs, or portions of PMFs, in the analysis, a mapping function such a f is used. Such functions are represented using the notation $f(x) = p$, denoting that the probability of x events is p. For compactness, when $f(x) = 0$, i.e. there is zero probability of a certain number of events, $f(x)$ is not explicitly stated. Addition is defined on such map functions as $(f_1 + f_2)(x) = f_1(x) + f_2(x)$. Similarly, division by a fixed number d is defined by $(\frac{f}{d})(x) = \frac{f(x)}{d}$.

To represent probability values, *Fixed Precision Fractions* are used. A Fixed Precision Fraction is defined to be a fraction of the form $\frac{n}{d}$ such that n and d are co-prime integers less than the maximum available precision α. In the case that n or d exceed α, the fraction is simplified to $\frac{n'}{d'}$ such that $\frac{n'}{d'}$ provides an appropriate bound (upper or lower, depending on context) of $\frac{n}{d}$. Fixed Precision Fractions are used instead of other representations, such as floating point, as they provide control over simplification and avoid problems such as floating point errors.

Two special symbols are used to represent the possible contents of a cache line, in addition to memory blocks. First, the symbol $\times$ represents the contents of an empty cache line. Second, the symbol $\varnothing$ represents a memory block for which no information is known. The concept of $\varnothing$, representing complete uncertainty, is useful in defining lossy compression.

Throughout this paper, a fully associative cache is assumed. We note that this restriction is easily lifted, as a set-associative cache with N sets can be analysed as N parallel and fully independent fully associative caches.

2. COMPRESSING COLLECTING SEMANTICS

In this paper, Lossy Compression is applied to the Collecting Semantics [10] of a Random Replacement Cache [6] to enable Must and May analysis. This section provides the motivation for doing so, as well as an outline of how these methods are derived. Collecting Semantics refers to a minimal representation that describes the behaviours of a specific system, utilising a map to show that all concrete states representing the same logical behaviour are represented by the same abstract state. The types of analysis desired, Must and May analysis, refer to analyses which calculate upper bound probability distributions for hits and misses respectively.

2.1 Collecting Semantics of the Random Replacement Cache

The Random Replacement Cache is a simple cache algorithm which, on requiring an eviction, randomly selects a cache line to evict. This behaviour means that aside from the contents of the cache, the Random Replacement cache stores no additional data. A cache may be specified in terms of its behaviour via *evictAndReplace* and *touch* functions, for a random replacement cache these functions are as follows:

- $\mathit{evictAndReplace}(\mathit{state}, \mathit{newLine})$: Select a cache line in *state* at random and replace it with *newLine*.
- $\mathit{touch}(\mathit{line})$: Do nothing.

The simplicity of the cache's data structures and eviction/touch operations lends itself to a correspondingly simple definition of collecting semantics. As the only data stored in the cache structure is the contents of the cache lines themselves, every cache line is equivalent. Hence, two cache states will exhibit the same behaviours if and only if they contain the same memory blocks. A naïve definition of the collecting semantics would be to simply take the set of memory blocks contained in a cache state to define the behaviour of the cache state, and the probability of the state occurring. However, this fails to take into account that due to the finite size of the cache, accesses are not independent, as outlined by Altmeyer and Davis [6]. Hence, it is not sound to merely calculate the probability of each access being a hit or a miss. Rather, in order to get useful results, it is necessary to extend the definition of a state to include a history as follows:

$$\langle S, p, \mathit{hist} \rangle$$

Where S is the set of memory blocks contained in the cache state, p is the probability of encountering the cache state, and *hist* stores information on the probabilities of hits/misses previously encountered.For example, in performing the analysis, *hist* assigns a probability to the number of hits and misses based on the probabilities of predecessor states where hits would have occurred. This is illustrated via the examples in Section 3.

To merge states in the collecting semantics, it is sufficient to combine the states which contain the same memory blocks (i.e. states which have the same S). The probability p of being in the merged state is simply the sum of the p values for the states that were combined to produce it. Similarly, each individual $h(x)$ value (representing the probability of a certain number of observed events e.g. hits) for the merged state is obtained by summing the corresponding $h(x)$ values for the states that were combined to produce it.

In performing a state exploration based static analysis on a random replacement cache, such as collecting semantics, the major issue to deal with is the number of successor states required after an eviction. As an eviction could evict any possible cache line, each eviction results in N successor states, where N is the size of the cache. Hence an exhaustive enumeration of j accesses would result in j^N states. The collecting semantics improves upon this, by ensuring that each set of memory blocks is only represented once, resulting in the binomial term ${}^jC_k$ states, which has $O(j^N)$ complexity. As j is typically very large, this is still unacceptably high.

2.2 Applying Lossy Compression to Collecting Semantics

In order to combat state explosion in the collecting semantics, it is necessary to lose information which may be useful, and hence it is necessary to introduce a notion of how useful data may be. Given that a random replacement cache results in any given observation having a probability of occurrence, it is easy to make a link between the probability of a given observation and the usefulness of the observation to the analysis. Hence the assumption used in devising compression is that any observation with a sufficiently low probability of occurrence can be discarded.

To determine observations which could have low probabilities of occurrence, it is necessary to revisit the previously introduced collecting semantics. This yields three primary observations that can be made on cache states or sets of cache states:

- **O1**: The probability that a given memory block is in any given cache state under analysis
- **O2**: The probability of observing a given cache state in the analysis
- **O3**: The probability of observing a given number of hits in a cache state history

The basis for **O1** being a target for compression is that after a number of accesses, the probability of a given memory block remaining in the cache is increasingly remote. Similarly, **O2** is justified by observing that certain cache states have a very small chance of occurring, due to the random replacement caches inherent ability to reduce the probability of pathological cases. Similarly, for **O3** the probability of observing a given number of cache hits may be similarly low. However, the justification for **O3** also reflects a problem inherent in the collecting semantics: as the length of the traces grows, so does the length of the history. Hence, the space complexity of the collecting semantics is bounded by $O(j * s)$ where j is the number of accesses and s is the number of states under consideration. As j grows, the cost of updating after each access grows linearly, and hence the time complexity of the analysis is of $O(j^2)$ with respect to the length of the trace. As j is typically large, this is unacceptable, and hence compression in line with **O3** is useful in bringing this complexity down.

In order to implement compression on **O1**, it is necessary to use the idea of an unknown memory block, $\varnothing$. While in a Must analysis an empty cache line provides a bounding behaviour, this is not true with regards to a May analysis as this could optimistically[1] indicate more misses than were possible[11]. Hence nothing can be assumed about a memory block which has been compressed. A cache line containing $\varnothing$ is defined such that the cache line could contain anything but is not guaranteed to contain any single item; therefore it cannot guarantee either a Hit or a Miss in Must or May analysis. Noting this, $\varnothing$ can be used to replace any sufficiently unlikely memory block and still have a sound approximation as this operation can only decrease the number of guaranteed hits or misses.

Implementing **O2** requires the use of a bounding state, which can trivially be constructed by filling the cache contents with $\varnothing$. However, the bounding state can be improved by inserting the last memory block accessed into the state, as this memory block is guaranteed to be in the cache. This also provides a guarantee that should compression occur in-between memory accesses to the same memory block, the compression will not reduce the probability of a hit.

In order to perform the compression on probabilities for **O2** and **O3**, it is necessary to define how to simplify the fixed precision fractions which represent the probabilities. When a fraction $\frac{a}{b}$ exceeds the maximum available precision, the new denominator is set to $b' = \lfloor \frac{b}{f} \rfloor$ where f is the *simplification factor*. Then the numerator a' is picked such that $\frac{a'}{b'} < \frac{a}{b}$. This ensures that the approximation is always pessimistic, by ensuring that the probability of an event decreases. Any error that the simplification introduces is combined into a single bounding state, with an appropriate history.

Given these observations, we can now define the types of compression used:

- *Memory Block Compression*: Memory blocks which are sufficiently unlikely can be replaced with $\varnothing$; compresses **O1**.
- *Cache State Compression*: Cache states which are sufficiently unlikely can be replaced with a bounding state; compresses **O2**.
- *History Compression*: Results in a History which are sufficiently unlikely can be replaced with a bounding history; compresses **O3**.

In order to implement memory block compression it is necessary to define a notion of how unlikely a memory block is across all analysis states under consideration. Two such methods are proposed:

- *Hit Probability*: Calculating the hit probability of each memory block under analysis will naturally find memory blocks which are less likely. Hence all memory blocks with a hit probability below a given threshold can be discarded and replaced with $\varnothing$. However, this may lead to memory blocks being kept under analysis for longer than is reasonably required for Must analysis e.g. in the case that no further accesses occur to a memory block, which leads to a higher runtime for the analysis.
- *Forward Reuse Distance*[2]: Defining the forward reuse distance of a memory block to be the number of accesses before that memory block is reused, it is clear

[1]Assuming that there exists a timing anomaly where a miss gives better overall performance than a hit.

[2]The term forward reuse distance is different to the term reuse distance as used in previous work.

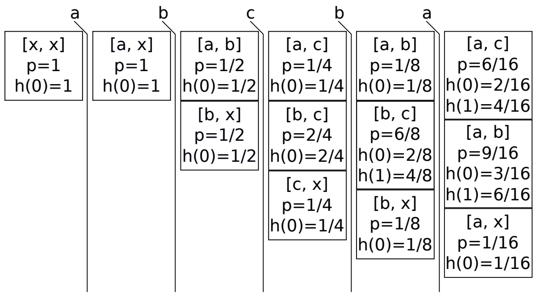

Figure 1: Applying collecting semantics to a 2-way cache for Must Analysis

that as the forward reuse distance increases, the minimum hit probability of the memory block decreases. Hence when the forward reuse distance of a memory block exceeds a certain threshold, it can be inferred that the memory block will become sufficiently unlikely to be retained before its next use, and hence can be discarded and replaced by ∅ immediately. The effect of this when compared to compression based on hit probability is that the earlier discarding decreases the runtime of the analysis, but by introducing ∅ to cache states earlier provides a significant impediment to the May analysis.

Implementing cache state compression and history compression is closely tied to the use of fixed precision fractions. Specifically, in both cases when fixed precision fractions are simplified, the difference between the original and simplified values is accumulated across all simplifications and added to an appropriate bounding state. For history compression, this is the least number of hits or misses in the distribution as appropriate. When applying cache state compression, it is necessary to construct a bounding cache state. This comprises a cache contents which only contains the last memory block accessed (as this provides a safe bound on all possible cache states) and a history which bounds all histories of merged states. Discarding the least likely history outcomes and cache states reduces the amount of information for the analysis to consider, and hence reduces the runtime of the analysis.

One side effect of utilising compression is that accesses are introduced which cannot be classified as either a hit or a miss. Hence it is necessary to change the representation of the history, as the miss distribution can no longer be found by treating every access not classified as a hit as a miss. Therefore, in order to obtain an accurate miss distribution, the history represented in the analysis must track both hits and misses separately.

3. EXAMPLES

To illustrate the techniques outlined, we apply the following sequence of accesses to a two-way random replacement cache, with the effects of each technique being illustrated:

$$a, b, c, b, a$$

As this is a 2-way cache, the maximum number of hits that can be observed for this sequence of accesses is 1. This is due to the fact that the access to c will ensure that at best, only one of a or b can remain in cache, and hence only one of these can be a hit. For reference, Figure 1 illustrates utilising the collecting semantics for analysis, and clearly shows state explosion when observing the total length of histories. However, as previously stated, utilising histories is necessary as simply computing hit/miss probabilities based

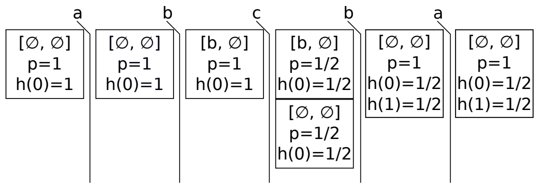

Figure 2: Applying Memory Block compression using forward reuse distance to a 2-way cache for Must Analysis

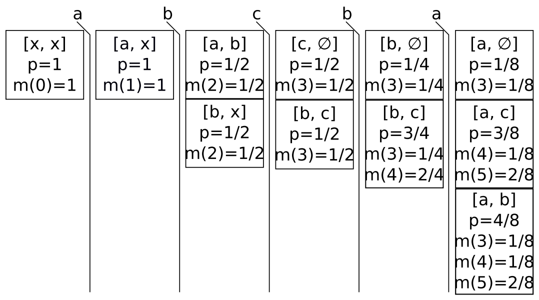

Figure 3: Applying Memory Block compression using hit probability to a 2-way cache for May Analysis

on current states under analysis may result in an impossible number of hits (in this case, 2) being assigned a non-zero probability.

Figure 2 demonstrates forward reuse distance compression on the same sequence of memory accesses, discarding any memory block which is not reused within 4 accesses, This causes the first access to a to be discarded, as well as the last accesses to any memory block, resulting in smaller number of states to explore. However, by discarding the first access to a, the analysis must assume a hit probability of zero for the second access, resulting in pessimism when compared to the collecting semantics. However, we note that in case of a May analysis, as ∅ is introduced immediately, forward reuse distance is unable to determine any misses in this case.

Figure 3 demonstrates hit probability compression, discarding all memory blocks whose hit probability is less than $\frac{1}{2}$. Note that in this example, it is the miss history that is illustrated. While more states are encountered than with forward reuse distance compression, ∅ is introduced much later. In turn, this allows a May analysis to classify some accesses to some states as misses, resulting in a much more accurate miss distribution at the expense of a more complicated analysis.

Figure 4 illustrates the use of Cache State and History compression, using fixed precision fractions with a maximum precision of 8 and simplification factor 4. After the final access to a, the precision exceeds that which is available and hence compression is applied. In comparison with the collecting semantics (Figure 1), this results in the state $[a, \times]$

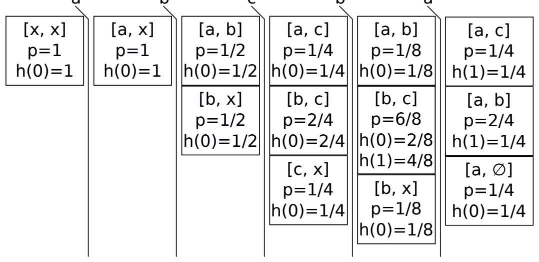

Figure 4: Applying Cache State and History compression to a 2-way cache for Must analysis

being removed entirely and the probabilities and histories of the other states being pessimistically simplified. Then the bounding state $[a, \varnothing]$ is created which provides a safe bound for all states which were simplified, and ensures that the total probability obtained by summing all states remains unchanged (i.e. remains equal to 1, since the cache has a probability of 1 of being in some state). Due to low cache associativity in this example, this does not change the total number of states as only a single state is sufficiently unlikely. However, in normal usage with higher cache associativity, greater compression would be expected. Further, as can be seen, the complexity of the histories attached to each state are simplified, reducing the time taken in updating the cache histories and the space complexity of the algorithm.

4. FORMALISM

Having demonstrated the compression methods, it now follows to present the formal definition. Implementing the novel step of the lossy compression approach requires a method which utilises heuristics to determine the value of data so it can be discarded and therefore mitigate the effect of state explosion. In order to implement discarding, the symbol $\varnothing$ is used to represent complete uncertainty of the content of a memory block. Using the symbol $\varnothing$ it is possible to discard specific memory blocks from analysis, as well as define a notion of a bounding state.

Three areas of the collecting semantics have been identified which yield three types of compression: *Memory Block Compression* (removing sufficiently unlikely memory blocks from analysis), *Cache State Compression* (removing sufficiently
unlikely cache states from analysis) and *History Compression* (removing sufficiently unlikely history outcomes from analysis). This section provides a detailed formalism of how these methods can be implemented. Firstly, the system model is introduced, which enabled the definition of collecting semantics which can operate in the presence of unknown information. Next, the three compression methods are implemented on the set of states under analysis, with hit probability and forward reuse distance based heuristics for memory block compression. Due to the implementation, these compression methods provide a sound approximation by design. Finally, these functions are combined to give a complete definition of the analysis, including its link to the original collecting semantics.

4.1 System Model

To start the formalisation of our approach, it is necessary to define the representation of the states of the analysis. Given L, the set of possible memory blocks that may reside in the cache, the abstract *cache content*, $\mathbf{s}$, is represented as on ordered tuple of size N equal to the cache associativity as follows:

$$\mathbf{s} = \langle s_0 ... s_N \rangle | s_i \in L \cup \{\varnothing, \times\} \tag{1}$$

Depending on context, the cache content is interchangeably represented as $\mathbf{s}$ or $\langle s_0 ... s_N \rangle$.

As the analysis utilises additional information, it is also necessary to track both probabilities of given cache contents and their history. Hence, the states used by the analysis also contains a probability, p, which represents the probability of the state occurring and a portion of the overall history of all cache states. As previously specified, probabilities are represented as fixed precision fractions with the parameters α, f which control the precision of the representation also being used to implement compression. The history of an analysis state is a pair of maps h, m which map the probabilities, again given as fixed precision fractions, of specified numbers of hits and misses occurring respectively. These maps are defined as portions of the PMF of all analysis states, and hence adding all hit history maps h under analysis recovers the complete hit history PMF. Hence an *analysis state*, $\mathbf{st}$, is represented by the tuple:

$$\mathbf{st} = \langle s, p, h, m \rangle \tag{2}$$

Similarly to the cache content, the analysis states are represented interchangeably as $\mathbf{st}$ or $\langle s, p, h, m \rangle$. Initially, the analysis starts with the state $\langle \langle \times .. \times \rangle, 1, h(0) = 1, m(0) = 1$, i.e. with probability 1 the cache is empty, and with probability 1 no hits or misses have occurred.

4.2 Concretisation and Classification

The next step is to define the map between abstract cache content and concrete cache contents. The concretisation function needs to map any instance of $\varnothing$ to an appropriate member of $L \cup \{\times\}$, such that no element other than $\times$ can be repeated, and provide all permutations of such tuples. Hence the concretisation function c can be defined as:

$$c'(\langle s_1 .. s_N \rangle) = \left\{ \langle x_1 .. x_N \rangle \middle| \begin{array}{l} x_i \in \begin{cases} L \cup \{\times\} \textbf{ if } s_i = \varnothing \\ \{s_i\} \textbf{ otherwise} \end{cases} \\ x_i = \times \textbf{ or } x_i \neq x_j, j \in [1, N] \\ i \in [1, N] \end{array} \right\} \tag{3}$$

$$c(\mathbf{s}) = \bigcup \{ permutations(\mathbf{x}) \mid \mathbf{x} \in c'(\mathbf{s}) \} \tag{4}$$

Note that the function c also defines the map between the standard collecting semantics and concrete states, and is only provided for completeness. In practice, one would use only the function c' to map from the compressed representation to the collecting semantics.

Given the concretisation function, it is trivial to define the classification function on an abstract cache state $\langle s, p, h, m \rangle$. For a given memory block $l \in L$, one of three possibilities must be true. Firstly, if $l \in s$, the access is a hit. Alternatively, if $\varnothing \in s$, then it is not possible to classify if the access is a hit or a miss as nothing is known about the memory block represented by $\varnothing$; hence the classification must return not classified (*N.C.*). Finally, if neither of these is true, the access can be classified as a miss. Hence the classification function cs can be defined as:

$$cs(\langle s, p, h, m \rangle, l) = \begin{cases} \text{hit } \textbf{if } l \in s \\ \text{N.C. } \textbf{if } \varnothing \in s \\ \text{miss } \textbf{otherwise} \end{cases} \tag{5}$$

4.3 Update Operations

Next we need to define the behaviour that the algorithm must take in each of these cases. For clarity of presentation each of these functions maps a single abstract state onto a set of successor states. In the case of a *hit*, the cache contents survives unchanged. Hence, the only action required is to update the hit history h to take into account that a hit must have occurred. Hence the *hit* function can be defined as follows:

$$hit(\langle \mathbf{s}, p, h, m \rangle, l) = \{ \langle \mathbf{s}, p, h', m \rangle | h'(x+1) = h(x) \forall x \} \tag{6}$$

Recalling that by the convention introduced earlier, as the number of hits has incremented, the map h' will return 0 for any number of hits not possible and in particular $h'(0) = 0$. The *miss* and *N.C.* (not classified) actions share some behaviour. In the case of a miss, a member of the cache is evicted leading to N successor states, each having their probabilities multiplied by $\frac{1}{N}$ as well as increasing the miss history. In the case of $N.C.$, the behaviour must bound both a *hit* or a *miss*. In the case that the access is actually a miss, then the behaviour defined for a miss without incrementing the number of misses clearly provides a bound. In the case that the access is actually a hit, this same behaviour provides a bound as the actual outcome, a cache hit, has a probability which is lower by a factor of $\frac{1}{N}$, and the probability of a hit for any other element of the cache has been lowered by the additional eviction. Hence the $N.C.$ behaviour can be soundly expressed as:

$$nc'(\langle s_1..s_n\rangle, l) = \left\{ sort(\langle s'_1..s'_n\rangle) \middle| s'_i = \begin{cases} s_i \textbf{ if } i \neq j \\ l \textbf{ if } i = j \end{cases} j \in [1, N] \right\} \tag{7}$$

$$nc(\langle \mathbf{s}, p, h, m\rangle, l) = \{\langle \mathbf{s}', \tfrac{p}{N}, \tfrac{h}{N}, \tfrac{m}{N} | \mathbf{s}' \in nc'(\langle s_1..s_k\rangle, l)\} \tag{8}$$

Where *sort* is a function that takes a tuple and returns a tuple whose elements have been ordered, N is the size of the cache and dividing a history by an integer represents all probabilities in the history being divided by that integer.

As a *miss* shares the same successor cache contents as $N.C.$, the miss function only has to handle updating the miss history appropriately. Hence the miss function can be defined using nc (8) as follows, again noting that by convention $m'(0) = 0$:

$$miss(\langle \mathbf{s}, p, h, m\rangle, l) = \left\{ \langle \mathbf{s}', p', h, m'\rangle \middle| \begin{array}{c} \langle \mathbf{s}', p', h, m\rangle \in \\ nc(\langle \mathbf{s}, p, h, m\rangle, l) \\ m'(x+1) = m(x) \forall x \end{array} \right\} \tag{9}$$

For convenience, the function *next* can be defined which, given an analysis state **st** and memory block access l, applies the appropriate update operation:

$$next(\mathbf{st}, l) = \begin{cases} hit(\mathbf{st}, l) \textbf{ if } cs(\mathbf{st}, l) = \text{hit} \\ nc(\mathbf{st}, l) \textbf{ if } cs(\mathbf{st}, l) = \text{N.C.} \\ miss(\mathbf{st}, l) \textbf{ if } cs(\mathbf{st}, l) = \text{miss} \end{cases} \tag{10}$$

Similarly, *next* can be extended to accept a set of states as follows:

$$nexts(S, l) = \bigcup\{next(\mathbf{st}, l) | \mathbf{st} \in S\} \tag{11}$$

4.4 Combining Cache States

If two analysis states represent the same cache contents, then they can be combined. Practically, this means that for two analysis states $\langle \mathbf{s_1}, p_1, h_1, m_1\rangle$ and $\langle \mathbf{s_2}, p_2, h_2, m_2\rangle$, the states can be combined if $\mathbf{s_1} = \mathbf{s_2}$. Combining two states simply takes the sum of their probabilities and history distributions. This is valid as each states' history distribution represents a portion of the overall history distribution, and hence summing merely provides a method to condense the data. Therefore the combining function *cmb* for a set of k states representing the same cache contents can be written as:

$$cmb\left(\left\{ \begin{array}{c} \langle \mathbf{s_1}, p_1, h_1, m_1\rangle \\ .. \\ \langle \mathbf{s_k}, p_k, h_k, m_k\rangle \end{array} \right\}\right) = \langle \mathbf{s_1}, p', h', m'\rangle \left| \begin{array}{l} p' = \sum_{i=0}^{k} p_i \\ h'(x) = \sum_{i=0}^{k} h_i(x) \\ m'(x) = \sum_{i=0}^{k} m_i(x) \end{array} \right. \tag{12}$$

Defining the combining function to construct the minimal set of states required to represent an arbitrary set of analysis states S can be accomplished by splitting the analysis states into sets sharing the same cache contents as follows:

$$same(S, \mathbf{s_j}) = \left\{ \langle \mathbf{s_i}, p_i, h_i, m_i\rangle \middle| \begin{array}{c} \mathbf{s_i} = \mathbf{s_j} \\ \langle \mathbf{s_i}, p_i, h_i, m_i\rangle \in S \end{array} \right\} \tag{13}$$

$$cmbs(S) = \{cmb(same(S, \mathbf{s_j})) | \langle \mathbf{s_j}, p_j, h_j, m_j\rangle \in S\} \tag{14}$$

Together, the classification, update and combination functions can be used to define the collecting semantics.

4.5 Compression

Having defined the collecting semantics with histories, we now define compression. The first form of compression, memory block compression, operates by replacing memory blocks with the unknown value $\varnothing$. A memory block is replaced when a given heuristic determines that it is not sufficiently significant.The actual replacement can be defined by the function r', which compares each element of the cache contents to a set of values which can remain, R, and replaces any that do not match with $\varnothing$. To apply this to an analysis state, the function r simply applies r' to the cache contents of the state:

$$r'(\langle s_1..s_N\rangle, R) = \langle s'_1..s'_n\rangle \left| s'_i = \begin{cases} \varnothing \textbf{ if } s_i \notin R \\ s_i \textbf{ otherwise} \end{cases} \right. \tag{15}$$

$$r(\langle \mathbf{s}, p, h, m\rangle, R) = \langle r'(\mathbf{s}, R), p, h, m\rangle \tag{16}$$

The first family of heuristic functions under consideration is *prb*, which keeps only those memory blocks deemed likely enough to significantly affect the analysis. Given a threshold, t, the prb_t function identifies all memory blocks which have a hit probability of greater than or equal to t and selects these to keep under analysis. Hence, where L represents the set of all possible memory blocks, prb_t can be defined as follows:

$$prb_t(S) = \left\{ l_i \middle| \Sigma\{p | \langle \mathbf{s}, p, h, m\rangle \in S, l_i \in s\} \geq t, l_i \in L, \right\} \tag{17}$$

The second family of heuristics, frd, is based on the forward reuse distance; given a forward reuse distance threshold d, the function frd_d discards all memory blocks whose forward reuse distance exceeds d. Assuming a function nu which returns the distance to the next use of a memory blocks, the function frd_d, which returns the set of memory blocks to keep under analysis, can be defined as follows:

$$frd_d(S, A) = \{l_i | l_i \in L, nu(A, l_i) \leq d\} \tag{18}$$

Given parameters t and d for the functions prb_t or frd_d, and the replacement function r (16), a function to define the first form of compression $comp1$ regardless of chosen heuristic can be defined as follows:

$$comp1_{(t,d)}(S, A) = \{r(s, frd_d(S, A) \cap prb_t(S)) | s \in S\} \tag{19}$$

Noting that as using the functions frd_∞ or prb_0 will return all memory blocks in S, $comp1$ is capable of implementing both forms of compression.

Cache State compression is related to the probability values of each state. As probability values have fixed precision, it is necessary to reduce the accuracy of the probability values soundly. If a probability value is reduced to zero due to loss of accuracy, then the state can be removed and combined into a common bounding state. Firstly, it is necessary to define a function $rpf_{(\alpha,f)}$ which inspects a fraction $\frac{a}{b}$ and if $b > \alpha$ (where α is the maximum available precision), reduces the precision of $\frac{a}{b}$. This is accomplished by setting the new denominator as $\lfloor \frac{b}{f} \rfloor$, where f is a fixed factor, and returning the greatest value possible using the new denominator which is still less than the original. As all probabilities can be expressed[3] as $\frac{x}{N^y}$, the factor f should be picked to be sufficiently large to provide meaningful compression and of the form N^z, to minimise unnecessary loss of accuracy. If $b \leq \alpha$, then no action is required.

$$rpf_{(\alpha,f)}(\tfrac{a}{b}) = \begin{cases} max(\{\frac{x}{\lfloor \frac{b}{f} \rfloor} \big| \frac{x}{\lfloor \frac{b}{f} \rfloor} < \frac{a}{b}\}) \textbf{ if } b > \alpha \\ \frac{a}{b} \textbf{ otherwise} \end{cases} \tag{20}$$

History compression is similar to cache state compression in that it relates to compressing fractions, this time in the histories. If an outcome requires too much precision to represent, it should have its precision reduced using the *rpf* function. As with cache histories, if this would reduce the fraction to 0, then the outcome can be removed in its entirety and any error introduced combined and added to the number of the least possible observed hits (or misses[4]) in a bounding state which combines all other such errors. Therefore, the history compression function $rph_{(\alpha,f)}$, using the maximum precision α and reduction factor f, can be defined as follows:

$$rph'_{(\alpha,f)}(h) = h' \big| h'(x) = rpf_{(\alpha,f)}(h(x)) \tag{21}$$

Given *rpf* and *rph* (i.e. (20), (21)), the Cache State and History compression can be combined into a single function applied to cache states, *rps*, as follows:

$$rps_{(\alpha,f)}(\langle \mathbf{s}, p, h, m\rangle) = \langle \mathbf{s}, rpf_{(\alpha,f)}(p), rph_{(\alpha,f)}(h), rph_{(\alpha,f)}(m)\rangle \tag{22}$$

As stated before, a bounding cache content can be constructed from the last memory block to be accessed l and $\varnothing$, by constructing a state which contains l and no other certain values. Hence, for a cache associativity of N, the bounding cache content after the last access l can be expressed as the function $bndc_N$:

$$bndc_N(l) = \langle s_1..s_N\rangle | s_i = \begin{cases} l \textbf{ if } i = 1 \\ \varnothing \textbf{ otherwise} \end{cases} \tag{23}$$

To calculate the histories for the bounding analysis state, it is necessary to find the amount that has been lost due to rounding. This is performed by summing all probabilities in all states and subtracting the result from one to determine the amount not accounted for. This value is then assigned to the minimum number of hits (or misses) present, and is expressed by the function *bndh* which operates on a list of histories.

$$smh(h) = \Sigma_{x=0}^{\infty} h(x) \tag{24}$$

$$minh(h_1...h_j) = min(\left\{ min(x \big| h(x) \neq 0) \big| i \in [1,k] \right\} \tag{25}$$

$$bndh(h_1...h_j) = h' \left| h'(x) = \begin{cases} 1 - \Sigma_{i=1}^{k} smh(h_i) \textbf{ if} \\ \quad x = minh(h_1...h_j) \\ 0 \textbf{ otherwise} \end{cases} \right. \tag{26}$$

Similarly, the probability of the bounding state can be found by subtracting the sum of the probabilities of all other states from one. Hence, the bounding state which takes into account the errors from applying $rps_{(\alpha,f)}$ to all states can be defined as the result of the bnd_N function, which operates on a list of all analysis states representing a cache of size N, as follows:

$$bnd_N\left(\begin{matrix} \langle \mathbf{s_1}, p_1, h_1, m_1\rangle \\ ... \\ \langle \mathbf{s_k}, p_k, h_k, m_k\rangle \end{matrix}, l \right) = \begin{matrix} \langle bndc_N(l), 1 - \Sigma_{i=1}^{k} p_i, \\ bndh(h_1..h_k), bndh(m_1..m_k)\rangle \end{matrix} \tag{27}$$

As Cache State and History Compression both utilise the same parameters, the function $comp23_{(\alpha,f)}$, which applies both methods to all elements of a set of analysis states, can be defined as follows:

$$comp23'_{(\alpha,f)}(S) = \{rps_{(\alpha,f)}(\mathbf{st}) \big| st \in S\} \tag{28}$$

$$comp23_{(\alpha,f)}(S) = comp23'_{(\alpha,f)}(S) \cup \{bnd(comp23'_{(\alpha,f)}(S))\} \tag{29}$$

Where the function *bnd* is the appropriate bnd_N function (27) for the cache under analysis. For convenience, the *comp*1 (19) and *comp*23 (29) functions can be combined, as follows:

$$comp(S,A)_{(t,d,\alpha,f)} = comp23_{(\alpha,f)}(comp1_{(t,d)}(S,A,hr)) \tag{30}$$

Where S is the set of analysis states to apply compression to, A the remaining list of accesses, *hr* the heuristic function used by *comp*1 and f and α the parameters for simplification of fractions used by *comp*23.

4.6 Complete Analysis

Having defined the behaviour of the collecting semantics and the compression functions, all that remains is to combine all of these to define the complete analysis. The complete analysis use the rules of the collecting semantics to update the current cache state with the next access, before applying compression to the resulting cache states and finally combining cache states which represent the same items. Once all accesses have been completed, all the partial histories attached to each cache state are combined to give the overall history for the analysis, and hence the hit/miss distributions determined by the analysis.

Hence, using the previously defined functions *nexts* (11), which returns the set of next states for a given memory block access), *cmbs* (14), which combines analysis states representing the same cache items, and $comp = comp_{(t,d,\alpha,f)}$ (30), which performs compression on the cache, with fixed parameters t, d, α, f), the analysis step function *step* can be defined recursively as:

$$step(S,A,comp) = \begin{cases} S \textbf{ if } A = [] \\ step(cmbs(comp(nexts(S, head(A)), \\ tail(A))), tail(A), comp) \end{cases} \tag{31}$$

Where the function $head(A)$ returns the first element of A and $tail(A)$ returns all other elements of A.

Once the *step* function returns the final set of analysis states, the partial histories can be extracted and combined to find the overall hit and miss distributions, using the *cmb* function (12); Note that unlike *cmbs*, *cmb* combines all states presented regardless of the items that they represent):

$$dists(S) = \langle h', m'\rangle | \langle \mathbf{s}', p', h', m'\rangle = cmb(S) \tag{32}$$

Hence, the hit and miss distributions, for a given sequence of memory block accesses A, with compression parameters given by the function *comp*, is given by:

$$analyse(A) = dists(step(\{\mathbf{i}\}, A, comp)) \tag{33}$$

Where $\mathbf{i}$ represents the initial state $\langle\langle s_0..s_k\rangle, 1, h_0, m_0\rangle$, with $s_i = \times \forall i$, $h_0(0) = m_0(0) = 1$. For completeness, a special case is where $comp = comp_{(0,\infty,\infty,1)}$. In this case, no

[3]As a probability in the collecting semantics can be divided by the cache size or added to another probability
[4]For May Analysis

compression is performed and hence the analyse function implements the collecting semantics.

5. EVALUATION

5.1 Experimental Setup

For evaluation, two new families of algorithms are defined, based on the lossy compression techniques introduced in this paper. The $FRD(r)$ family uses the forward reuse distance heuristic frd with reuse threshold r (implemented using $compl_{(r,0)}$, and the $PRB(t)$ family uses the hit probability heuristic prb with threshold t (implemented using $compl_{(\infty,t)}$. In both cases, the parameters $\alpha = 2^{31}$ and $f = 2^{6}$ were used for the fixed precision fraction compression, giving an accuracy of approximately 10^{-9} for this analysis. If greater precision were required, this could be obtained by picking larger values for α and f. For Must analysis, the methods were compared against the result of 10^{9} simulations. The hit/miss distributions from a large number of simulations will converge on the worst case distribution of the benchmark, as due to the single path nature of the benchmark there is only a single behaviour it can exhibit. Hence it can be inferred that simulations will also converge onto the uncompressed collecting semantics. In addition, the current techniques of reuse distance rd from Davis et al. [5], and the focus blocks scheme with x focus blocks, $FB(x)$, from Altmeyer and Davis [6] are also compared against. As no previously existing method can perform a May analysis, the methods introduced in this paper are only compared against simulations in that case.

Experiments were performed on traces from a subset of the Maladärlen benchmarks [16], as in Altmeyer and Davis' work [6]. In order to test the performance on both smaller and larger traces, 4 of the traces contained system calls which increase the size of the trace substantially. The cache simulated was a 16-way random replacement cache with a cache line size of 8.

Results are presented as a graph plotting the complement of the cumulative distribution function $(1-CDF)$ produced by the analysis, giving the number of hits (for Must analysis) or the number of misses (for May analysis). This differs from previous works [5, 6], which effectively show the number of misses for a Must analysis; technically, this is slightly misleading, as the analysis used does not predict the number of guaranteed misses, but rather the number of accesses not guaranteed to be a hit. Hence in this paper, an optimistic analysis would produce a result greater than that of the simulation, rather than less than that of the simulation as in previous works. During testing, optimism when compared against the results of the simulation was specifically tested for and not observed, validating that the compression employed only introduces pessimism.

5.2 Results

Figure 5 shows the results for the *insertsort* benchmark. The results show that, in this case, $FRD(24)$, $PRB(0.6)$ and $FB(12)$ all produce roughly equivalent results, although $FRD(24)$ takes noticeably less time to complete than other approaches at this accuracy. As expected, higher accuracy is obtained by $PRB(0.5)$ and $FRD(116)$, with $FRD(116)$ being nearly identical to the simulation, and hence the uncompressed collecting semantics, although noticeably faster. While there is a significant difference in parameter between the two FRD methods, this is because the parameters inbetween the two do not produce additional accuracy. Due to the nature of the approach, this also means that, for ex-

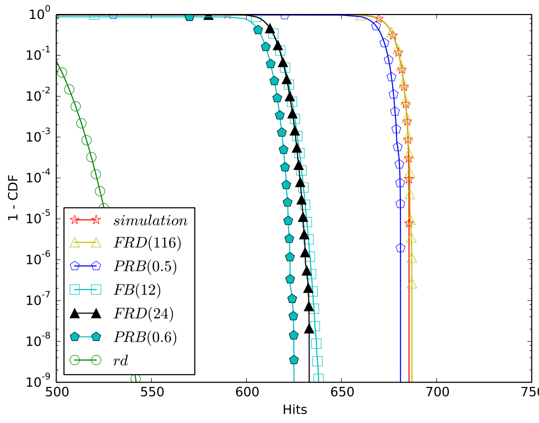

Figure 5: Comparison of approaches for Must analysis on the *insertsort* benchmark

ample $FRD(80)$ takes the same amount of time to analyse the trace as $FRD(24)$, as no additional work is performed.

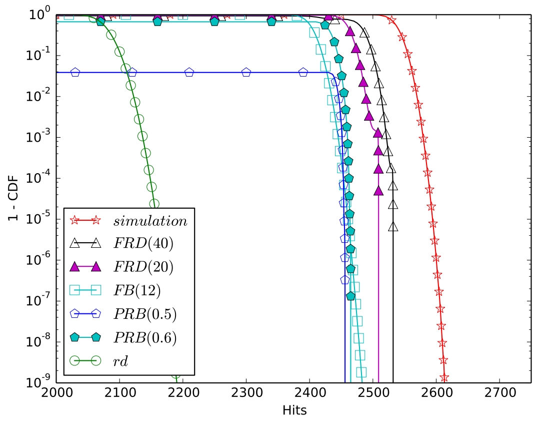

Figure 6: Comparison of approaches for Must analysis on the fir benchmark

Figure 6, shows the results for the longer fir benchmark, which demonstrates a property of applying this form of compression: compression artefacts. Over a large number of accesses, compression artefacts accrue due to the rounding error when simplifying finite fractions; these can be seen as the lines for the FRD and PRB approaches are pessimistic at high probabilities (e.g. 10^{-1} to 10^{-2}) when compared to FB. While not shown, an example of this is that $PRB(0.5)$ predicts less than 1000 hits at a probability of 10^{-1}. As these artefacts come from rounding error, which has been designed to be pessimistic, the compression artefacts do not affect the soundness of the analysis. While this results in poor performance for high probabilities when compared with the FB or rd methods, at the lower probability levels of interest (e.g. $10^{-}9$), FRD and PRB yield higher accuracy.

Figure 7 again illustrates the sensitivity of the threshold for forward reuse distance. As the reuse distances in the bs benchmark are relatively high, it is necessary to select a relatively high threshold. As expected, this increases the execution time and memory usage of the analysis substantially. In particular, one weakness is highlighted in that as the reuse distances of bs fall into a very tight range, the difference between $FRD(24)$ and $FRD(28)$ is so substantial

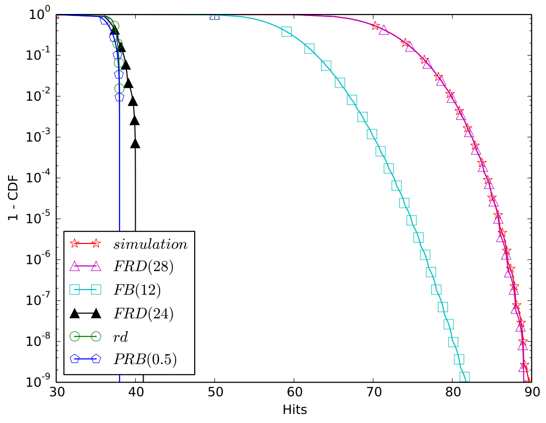

Figure 7: Comparison of approaches for Must analysis on the *bs* benchmark

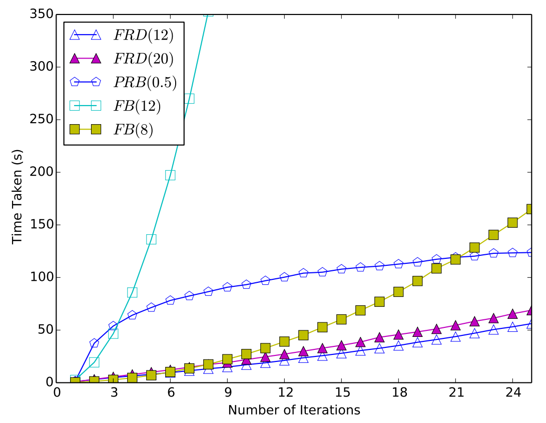

Figure 8: Comparison of Execution Time for analysis of iterations of *fibcall*

that $FRD(24)$ performs relatively poorly while $FRD(28)$ is nearly equivalent to performing the collecting semantics without compression.

Figure 8 compares the runtime of the analysis for $FRD(12)$, $FRD(20)$, $PRB(0.5)$ and $FB(12)$ on repeated iterations of the $fibcall$ benchmark; the parameters for the methods were picked as they all produce similar distributions. In addition, $FB(8)$ is presented for comparison, although this gives a noticeably more pessimistic distribution. Due to the use of finite precision fractions throughout lossy compression based analysis, a cache state has a fixed size. As expected, Figure 8 reflects this, as the runtime of the lossy compression based analyses for a given set of parameters varies approximately linearly with the length of the supplied trace. Further, with respect to the threshold, the complexity is at most exponential; however, if the threshold given is set to a value which gives no additional accuracy, there is little or no penalty with respect to analysis runtime, as seen with $FRD(12)$ and $FRD(20)$. The focus block approaches, $FB(8)$ and $FB(12)$ (which is of similar accuracy to $FRD(12)$), do not observe this linear relationship with the size of the trace, as they use histories with potentially unbounded size. This comparison was also attempted with the larger fir benchmark, but memory usage prohibited $FB(12)$ from completing.

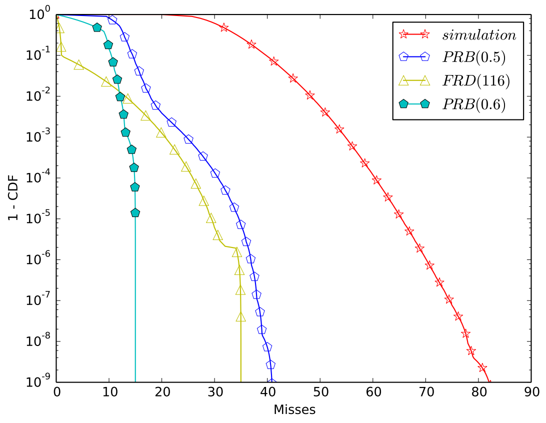

Figure 9: Comparison of approaches for May analysis on the *insertsort* benchmark

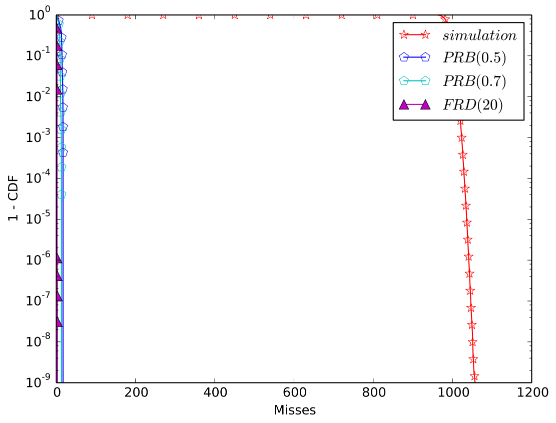

Figure 10: Comparison of approaches for May analysis on the *statemate* benchmark

Figure 9 shows the performance of May analysis on the $insertsort$ benchmark. Despite $FRD(116)$ providing a highly accurate Must analysis, it is beaten by $PRB(0.5)$ on the May analysis. This is as expected, as the FRD method is more aggressive in discarding information. To illustrate the problem of FRD for May analysis, $FRD(24)$ which provided an acceptable Must analysis is unable to predict a single cache miss. While the PRB technique takes longer to perform, it is able to provide an effective May analysis.

Unfortunately, Figure 10 indicates the difficulty of scaling the PRB approach to larger traces. As the distance between consecutive accesses to the same memory block grows, the probability threshold used by PRB has to be set lower. As the probability threshold decreases, the time taken for analysis grows, and hence conducting an accurate May analysis on larger benchmarks proves intractable.

6. SUMMARY & CONCLUSIONS

This paper introduced the idea of applying lossy compression to the collecting semantics of the random replacement cache, identifying three separate types of information which could be approximated with low impact on the accuracy of the analysis. It utilised two different heuristics to implement the compression, (i) based on hit probability and (ii) based on forward reuse distance, yielding two different analyses.

In comparison with the approach of Altmeyer and Davis

[6], then with appropriate choice of parameter, the new approach using the forward reuse distance heuristic yields a superior Must analysis. Further, when a parameter is selected that provides comparable accuracy to Altmeyer and Davis's approach, this new approach produced a speed-up of between two and fourteen times depending on the length of the trace, with greater speed-ups anticipated for longer traces. This is due to the fact that lossy compression ensures a linear relationship between the runtime of the analysis and the length of the trace, improving upon the quadratic relationship of previous work. Further, parameter selection is aided by the fact that the parameter controls the maximum precision available, and hence increasing the parameter only increases the runtime of the analysis if the accuracy of the analysis would also increase. The drawback of the forward reuse distance heuristic is that its aggressiveness yields a poor May analysis.

The new approach using the hit probability heuristic results in Must analysis that is slower than using the forward reuse distance heuristic; however, it maintains a linear relationship with the length of trace, and hence for sufficiently long traces is shown to outperform previous work in terms of its runtime for comparable accuracy. The May analysis is shown to be useful for smaller traces, although due to increased reuse distances it loses accuracy in larger traces.

We note that May analysis bounding the number of cache misses is only required for systems with timing anomalies, where misses could sometimes contribute more to the overall probabilistic worst-case execution time than hits. For systems without timing anomalies, only Must analysis is needed. The lossy compression based techniques introduced in this paper improve upon the state-of-the-art Must analysis in terms of both precision and runtime.

6.1 Future Work

In order for the analysis to be relevant for more complicated systems, it is necessary to extend the analysis to handle multiple paths. Further, lossy compression can typically be used to define two major modes of compression. This paper has looked at *constant quality compression*, where results are guaranteed to be of a fixed quality specified by a parameter. An alternative is *fixed effort compression* (analogous to *fixed bitrate compression* [7]), where a metric is specified for the amount of effort that can be expended. With the amount of effort fixed, the compression strength is varied such that the best result is achieved without exceeding the effort limit given. Given that this allows control over the runtime of the analysis, fixed effort compression could be useful in many situations.

Acknowledgement

This work was partially funded by the EU FP7 Integrated Project PROXIMA (611085).

7. REFERENCES

[1] S. Edgar, "Estimation of worst-case execution time using statistical analysis," Ph.D. dissertation, University of York, 2002.

[2] L. David and I. Puaut, "Static determination of probabilistic execution times," in *Proceedings. 16th Euromicro Conference on Real-Time Systems (ECRTS)*, June 2004, pp. 223–230.

[3] F. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim, "Proartis: Probabilistically analysable real-time systems," *Transactions on Embedded Computing Systems*, 2012.

[4] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones, and F. J. Cazorla, "Measurement-based probabilistic timing analysis for multi-path programs." in *ECRTS*, R. Davis, Ed. IEEE Computer Society, 2012, pp. 91–101.

[5] R. I. Davis, L. Santinelli, S. Altmeyer, C. Maiza, and L. Cucu-Grosjean, "Analysis of probabilistic cache related pre-emption delays," in *25th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2013, pp. 168–179.

[6] S. Altmeyer and R. I. Davis, "On the correctness, optimality and precision of static probabilistic timing analysis," in *17th Design, Automation and Test in Europe Conference (DATE)*. EDAA, 2014.

[7] J. Watkinson, *Compression in Video and Audio.* Focal Press, 1995.

[8] K. S. G. Brandenburg, "ISO/MPEG-1 audio: A generic standard for coding of high-quality digital audio," *J. Audio Engineering Soc*, vol. 42, no. 10, pp. 780–792, 1994.

[9] D. Griffin, B. Lesage, A. Burns, and R. I. Davis, "Lossy compression for worst-case execution time analysis of PLRU caches," in *RTNS '14: Proceedings of the 22nd International Conference on Real-Time and Network Systems*. Versailles, France: ACM, New York, NY, USA, 2014.

[10] 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*. Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252.

[11] F. Mueller, "Timing analysis for instruction caches," *Real-Time Systems*, vol. 18, pp. 217–247, May 2000.

[12] Y. Liang and T. Mitra, "Cache modeling in probabilistic execution time analysis," in *Proceedings of the 45th Annual Design Automation Conference*, ser. DAC '08. New York, NY, USA: ACM, 2008, pp. 319–324.

[13] L. Cucu-Grosjean, "Independance - a misunderstood property of and for probabilistic real-time systems," in *Alan Burns 60th Anniversary*, N. Audsley and S. Baruah, Eds., Mar. 2013.

[14] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 12–21.

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

[16] Contributors, "Maladärlen WCET benchmarks," http://www.mrtc.mdh.se/projects/wcet/, accessed on 1st September 2013.