Array-aware Cache Analysis for Write-through and Write-back Caches

Tobias Blaß

Master Thesis

Real-Time and Embedded Systems Lab
Saarland University

Supervisor:	Prof. Dr. Jan Reineke
Advisor:	Sebastian Hahn
Reviewers:	Prof. Dr. Jan Reineke
	Prof. Dr. Sebastian Hack

Submitted: December 2016

Declaration of Authorship

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Datum/Date: ______________________________

Unterschrift/Signature: ______________________________

SAARLAND UNIVERSITY

Abstract

Real-Time and Embedded Systems Lab

Master of Science

by Tobias Blaß

Caches are an important component of modern computers, bridging the performance gap between processor and memory. Consequently, cache analysis is a crucial part of precise worst-case execution time (WCET) analysis. Prior work has focused on *write-through* caches, which directly relay stores to main memory. In this thesis, we develop an analysis for *write-back* caches, which handle stores locally and forward them later. While write-back caches avoid slow memory accesses, they are also harder to analyze since the memory access is decoupled from the store instruction. We identify *dirtifying stores* as an upper bound to the number of write backs in the program. In many of our testcases the analysis predicts lower WCET bounds than for an equally-sized write-through cache. However, memory accesses to unknown locations turn out to pose a serious threat to write-back analysis precision. We therefore also develop array-aware variations of traditional cache analyses (namely must analysis and persistence analysis). While the array-aware must analysis is comparatively ineffective, identifying persistent arrays reduces the number of misses by more than 60% on half our benchmarks. In combination, array-aware write-back analysis improves on the write-through analysis in all but two of our testcases.

Acknowledgements

First and foremost I thank Sebastian Hahn for many fruitful discussions and his vast knowledge of the *llvmta* internals.

I also thank Prof. Jan Reineke for his supervision and for always asking the right questions.

I thank Prof. Sebastian Hack for reviewing this thesis.

I thank Tomasz Dudziak for many helpful discussions about C++, abstract interpretation and polish grammar.

Finally, I want to thank Kathrin Stark for proofreading this thesis. I simply cannot overstate how many completely useless filler words are on your conscience by now.

Content

CHAPTER 1

INTRODUCTION

Many of today's safety-critical systems are controlled by microprocessors. In this domain, timeliness is an essential part of correctness: programs not only have to compute correctly, they also have to finish in time. For example, the airbag in a car has to deploy within 15-25 milliseconds after the impact [1]; any later reaction is useless or even harmful. In many jurisdictions, the car manufacturer therefore has to prove to the certification authorities (and the customers) that the controller always inflates the airbag within this timeframe.

An integral part of this proof is computing an upper bound on the worst-case execution time (WCET) of the control program. This bound is often determined by measuring the executing time with as many inputs and in as many situations as possible. However, this technique is potentially unsafe. As indicated in Figure 1.1, the gap between the best-case runtime and the worst-case runtime can be large, and measuring might fail to find the rare outliers that have a significantly higher execution time.

The WCET estimation in this thesis is based on static analysis. Unlike measurement-based analysis (also called "dynamic analysis"), static analysis looks at the program itself without assuming any particular input or sequence of events. It therefore never

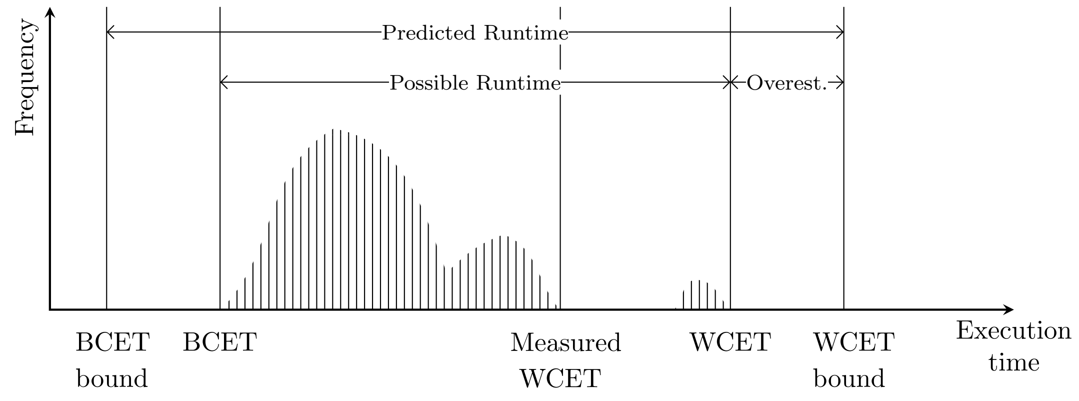


FIGURE 1.1: Potential probability distribution of runtimes for a given program.

overlooks any program paths and always yields sound results. On the downside, static analysis always errs on the side of caution and overestimates the true WCET. Minimizing the amount of overestimation while still being efficient is one of the greatest difficulties in static WCET analysis.

Contributions

The execution time of a program heavily depends on the underlying hardware; every program ends up as a sequence of assembly instructions interpreted by the processor. Of all processor components, caches are arguably one of the most important sources of overestimation. In this thesis, we develop a novel analysis for *write-back caches*. The write-back policy promises to reduce the number of main memory accesses and thereby increase performance. However, it is difficult to analyze and, to our knowledge, there is no analysis that can do so without tremendous overestimation.

Further, motivated by difficulties in the write-back analysis, we turn towards dynamic array accesses. A *dynamic array access* is a memory access into an array where the address is only computed at runtime. This thesis presents techniques allowing different cache analyses to retain more information across dynamic accesses.

Structure Chapter 2 introduces caches and how to analyze them. We then (Chapter 3) present our timing analyzer *llvmta*, which we use to implement and evaluate our analyses. We also explain step by step how we compute a WCET bound for a given program. Chapters 4 and 5 present the write-back cache analysis and the array access analysis, respectively. As they are independent of each other, they can be read in any order.

Chapter 2

Background

Due to the ever-increasing gap between processor and memory speed modern computer systems use local caches. These caches are small and fast memories, which contain a small subset of the data in main memory. If the core requests a piece of data which is *cached* (i.e. available in the cache), the request can be served much faster than an equivalent request to a non-cached location. Caches are subject to a range of design tradeoffs:

Line Size. Memory instructions usually access memory on a byte or word granularity. Requesting only a word of memory at a time is natural for a processor, as it has to fit the data into a register. Main memory, on the other hand, is optimized for throughput. Although it is expensive to load a piece of data, loading additional contiguous data is cheap. By eagerly loading more than requested, the cache can quickly serve subsequent requests for data in the close vicinity. According to the principle of spatial locality [2, p. 45], this is a worthwhile tradeoff.

Caches implement this technique by operating on a larger granularity than words, so-called *cache lines*. To optimally exploit spatial locality, the cache line should be as large as possible. However, increasing the line size has its cost: First, one has to transfer more data per memory access. Since the memory bus has a fixed width this requires more cycles per transfer and thereby higher access latencies. Second, the size of cache lines also determines their number; a cache containing few but large lines can only hold a limited number of separate memory locations. It therefore might be forced to evict useful data to make room for speculatively loaded useless data. As a compromise between these conflicting goals, 32 and 64 byte are popular line size choices [2, p. B-28].

An important consequence of operating at larger granularities than words is *false sharing*. If two variables fall into the same cache line, they are indistinguishable from the perspective of the cache. To avoid this technical detail we only consider variables at cache line granularity in this thesis. Our implementation does not have this restriction. We denote the set of cache line-sized memory blocks as $\mathcal{B}$.

Associativity (k) and Cache Sets. When a memory block is loaded into the cache, the cache has to choose a place for it. One possible strategy is to allow cache lines to hold any block. However, caches critically depend on a low access latency. As the cache gets larger, finding a memory block under this policy becomes more and more costly as more cache lines have to be searched.

Another strategy is to map each memory block to a single cache line. Finding the block then becomes trivial, i.e. access latency is as low as it can be. On the other hand, the cache loses all flexibility which cache lines to hold: if two heavily-used blocks map to the same cache line, they evict each other repeatedly, even if the rest of the cache is empty. As a compromise, the cache is usually split into multiple *cache sets*. Depending on its address, the memory block can only be placed in a subset of the available cache lines, namely those reserved for the cache set of the block. At lookup time only the corresponding cache set has to be searched. The number of cache blocks reserved for each cache set is called the *associativity* of the cache. Since the associativity plays a paramount role in cache analysis we concisely refer to it by the letter k. The two extreme cases elaborated above are the special cases where there is only one cache set ("fully associative") and where associativity is one ("directly mapped") .

Replacement Policy. Unless the cache is directly mapped, there is a choice which block to evict when the cache has to make room for a new memory block. In this thesis we only consider the Least-Recently-Used replacement policy (LRU). Under LRU, caches evict the memory block not accessed for the longest time. Temporal locality [2, p. 45] suggests that this is the block least likely to be needed in the near future.

To implement this policy, the cache has to remember the most recent access to each block. This state can be modelled as a function $age : \mathcal{B} \to \mathbb{N}_{\leq k}$. The age of a block b states how many different blocks have been accessed since the last access to b. After an access to block x the age of b changes as follows:

$$update(age, x) = \lambda b. \begin{cases} 0 & x = b \\ age(b) & age(b) > age(x) \\ \min(k, age(b) + 1) & \text{otherwise} \end{cases}$$

As the cache set can only hold k blocks, only the blocks with an age between 0 and $k-1$ are present in the cache. It is therefore unnecessary to distinguish ages larger than k.

So far we (implicitly) assumed that all memory accesses are loads. Stores are different in that the processor is not interested in the current data but only wants to overwrite it. This opens up two new degrees of freedom:

Write-hit Policy. If the destination of the store is already present in the cache, one either updates the original block in main memory together with the local copy ("write-through") or one defers updating main memory and just updates the local copy. Since future loads always ask the cache first, the program can never observe the outdated original. The change is only applied to main memory if and when the block is evicted from the cache. This policy is called write-back.

The advantage of the write-back strategy is that multiple stores to the same cache line are handled in the fast cache, requiring only a single access to the slow memory to write back the final result. In exchange, the cache becomes more complex as it has to differentiate *dirty* data that disagrees with the version in main memory from *clean*, unchanged data. The former has to be written back while the latter can be safely overwritten.

Write-miss Policy. If the destination of the store is not cached, loading the data is optional. Instead, the cache can relay the store to main memory, avoiding an allocation and therefore a possible eviction. This policy is called "non-write-allocate", while the opposite policy is called "write-allocate".

Even though there are four potential combinations of these policies, only two are commonly found in practice. Write-back caches usually follow the write-allocate policy, hoping that future stores to the same cache line can be handled locally. Write-through caches on the other hand enjoy no such performance benefit on future stores and therefore often follow the non-write-allocate policy. We will omit the write-miss policy throughout this thesis and assume the usual combination.

2.1 Static WCET Analysis

In the early days of computing, worst-case execution time calculation was simple. Processors came with handbooks containing execution times for all supported instructions [3, p. 8], and computing the WCET consisted of summing up the lengths of the individual instructions. But in the perpetual quest for ever-increasing performance, this simplicity has long been lost. As early as 1968, high-end mainframe computers had gained caches and overlapped consecutive instructions. For such a system it is impossible to give per-instruction worst-case execution times without extreme overestimation, and a computer reference manual from the time has to state that "specific programs evaluated to date have shown the time computed from the list to vary from the actual time by as much as 28.5 percent." [4, p. 27]. By now, virtually all processors are equipped with pipelines and caches, and the resulting system complexity has rendered the simple approach to worst-case execution time analysis useless.

The only way to give reasonable bounds in modern systems is by taking the hardware state into account. If the system contains a pipeline, the analysis must remember the previously executed instructions still in flight. If the system contains a cache, the analysis must always remember the currently cached memory blocks. Unfortunately, tracking the hardware state this way carries a whole series of additional complications. These complications are best explained by example:

```
/* a is cached */
if (...) access(b)
else     complex_calculation();
access(a);
```

Assume that a memory access takes 20 clock cycles and the complex calculation requires 30 cycles. For the sake of simplicity, we only consider the data memory accesses.

Without a cache we could compute the WCET using the 1960's approach:

$$\underbrace{\max(20,30)}_{\text{if}} + \underbrace{20}_{a} = 50 \text{ cycles}$$

However, assume the system contains a cache containing one line. Which block is cached after the *if* statement has been executed? If the *then* branch has been executed it is b, but if the *else* branch has been executed it is a.

It is tempting to follow either the *then* branch or the *else* branch, depending on which takes longer. This approach is overly simplistic and in fact produces incorrect results, as our example shows:

With 30 cycles, the *else* branch is longer than the *then* branch. Afterwards, *a* is cached. The memory access therefore hits the cache and we compute a WCET bound of

$$\underbrace{30}_{\text{else}} + \underbrace{1}_{a} = 31 \text{cycles}$$

If the program executes the *then* branch instead, b would be cached.

$$\underbrace{20}_{\text{then}} + \underbrace{20}_{a} = 40\text{cycles} > 31\text{cycles}$$

The *then* case, despite taking less time than the *else* case, causes a cache misses in the future that more than offsets the original difference. Clearly, assuming that a was cached is unsound. Yet assuming the worst cache state possible instead yields 50 cycles again, meaning that the cache does not improve our WCET at all. In our case, the bound is only 10 cycles higher than the true WCET. In real programs, though, this overestimation quickly grows.

To sum up, there are two sound ways of handling control flow: For one, the analysis assumes nothing about the hardware state after the two paths merge. It assumes the slower branch in the *if* as well as the higher execution time after the merge, a path never occurring in actual execution.

Alternatively, the analysis remembers the hardware state through the if-statement, allowing it to recognize the infeasible path. However, this requires the analysis to track *both* paths of the *if*, since only the seemingly faster path leads to the true worst-case execution time. In general, the analysis has to follow each of the exponentially-many paths in the program.

As it turns out, having to follow all paths is not limited to control-flow induced decisions. In many processor-internal decisions, the seemingly faster option can be slower on a global scale: a cache hit worse than a cache miss, overtaking a pending memory access in the pipeline might be worse than waiting for the access to finish, etc. These effects are also called *timing anomalies*. They make the number of paths not only exponential in the number of decisions made by the program but exponential in the number of decisions made by the processor, i.e. roughly exponential in the number of assembly instructions in the program.

Although following exponentially many paths is infeasible in general, it can be done in special cases. When two paths reach the exact same state they behave identically in the future and can be collapsed into one path again. The number of paths is therefore bounded by the number of states of the processor. An in-order pipeline, for example, can only be in a very limited number of states at each program point, making it likely that two paths end up in the same state. Caches, however can be in at least $\frac{|B|!}{(|B|-k)!}$ different states per cache set. Experimental evidence further suggests that the number of possible cache states is too large to handle; when Dalsgaard et al. explicitly modeled two 16 KiB 64-way caches in their METAMOC paper [5], the tool ran into their 4 GiB memory limit while analyzing the bsort100 benchmark[1]. Clearly, cache analysis requires a different approach.

2.1.1 Abstract Interpretation

Abstract interpretation [6] is a formal static analysis framework popular in compiler research and formal software verification. It provides a way to reduce the number of states to a manageable level without sacrificing too much analysis precision. Instead of modelling the complex hardware, it defines an abstract machine whose states correspond to multiple hardware states at once. Each operation in the real hardware has a corresponding operation on the abstract machine (often called *update*). The most important feature of the abstract machine is the ability to join two states, forming a new state that overapproximates the union of the two concrete state sets. Whenever there are two possible successors to a state, the analysis cuts the size of the state space in half by joining them into a single state. It pays for this reduction with a loss of analysis precision, i.e. there are properties unfulfilled by the abstract machine but true for the concrete machine.

As a simplified example, let the concrete hardware state be the value of a single register. To avoid dealing with 2^{2^n} possible states, we might choose our abstract state to be an interval of possible values. If the register can have any value from 1 to 5, we represent this set of hardware states by $[1, 5]$. Note that there are many sets of hardware states that cannot be described precisely – the set $\{1, 3\}$ has to be approximated by $[1, 3]$. When asked whether the value of the register is always odd, the analysis has to concede that it does not know, even though the property holds.

For reasons of space we only give a short introduction into the underlying mathematics. The abstract interpretation framework requires the abstract states to form a lattice under a partial order $\sqsubseteq$. Two states can be joined by taking their least upper bound $\sqcup$. The translation between abstract states and sets of concrete states is performed by an abstraction function $\alpha : \mathcal{P}(\mathit{ConcreteStates}) \to \mathit{AbstractStates}$ and a concretization function $\gamma : \mathit{AbstractStates} \to \mathcal{P}(\mathit{ConcreteStates})$. The correctness of the analysis hinges on two basic properties. First, the abstraction function has to perform an overestimation, i.e. must not lose any concrete states.

$$\forall S \in \mathcal{P}(\mathit{ConcreteStates}).\ \gamma(\alpha(S)) \supseteq S$$

[1]The testcase sorts a 100-element list using bubblesort. For comparison, our tool requires 54 MiB in this setup.

Second, the operations of the abstract machine have to be consistent with the ones on the concrete machine. This guarantees that a correct abstraction stays correct across state transitions.

$$\forall S \in \mathcal{P}(\mathit{ConcreteStates}).\ \gamma(\mathit{update}(\alpha(S)) \supseteq \bigcup_{s \in S} \mathit{concreteupdate}(s)$$

2.2 Cache Analysis

As far as WCET estimation is concerned, caches are arguably the most critical component in the processor – the difference between a cache hit and a cache miss can easily amount to a factor of 20 or more [7, p. 132]. The central goal of cache analysis is to statically prove that a memory access hits the cache. As a secondary goal, proving that an access misses the cache reduces the number of paths and therefore benefits analysis runtime, even though it usually does not improve the WCET estimate. Since cache analysis is a large field we only present a short summary of the cache analyses required for the understanding of our work. A more complete and detailed overview can be found in [8].

2.2.1 Must and May Analysis

The must analysis [9] is an analysis aimed at proving cache hits. It does so by overestimating the age of memory blocks. Thus, whenever a block has an age $< k$ in the must cache it also *must* be in the concrete cache. We formalize this property by the concretization function

$$\gamma(S) = \{\mathit{age} \mid \forall b \in \mathcal{B}.\ \mathit{age}(b) \leq S(b)\}$$

The analysis operates on $\mathcal{B} \to \mathbb{N}_{\leq k}$, mapping each block to an upper bound on its age. The update function closely resembles the update function for concrete LRU caches but two blocks can have the same age.

$$\mathit{update}(S, x) = \lambda b. \begin{cases} 0 & x = b \\ S(b) & S(x) \leq S(b) \\ \min(k, S(b) + 1) & S(x) > S(b) \end{cases}$$

Since we maintain upper bounds, the join function takes the maximum of both bounds, guaranteeing that both incoming states are still represented by the new state.

$$S \sqcup T = \lambda b. \max(S(b), T(b))$$

The may analysis complements the must analysis by providing a lower bound on the age of a memory block. It therefore states which blocks *may* be in the cache at any given time or – by complement – which blocks are definitely not cached. Its concretization function is the must concretization with $\leq$ replaced by $\geq$.

The update function looks similar to the must update function. It only differs if two blocks have the same age bound.

$$update(S, x) = \lambda b. \begin{cases} 0 & x = b \\ S(b) & S(x) < S(b) \\ \min(k, S(b) + 1) & S(x) \geq S(b) \end{cases}$$

The join function takes the minimum of both bounds.

$$S \sqcup T = \lambda b. \min(S(b), T(b))$$

For reasons of brevity and clarity, we use a graphical notation for must and may states. Instead of mapping blocks to ages, the notation turns the mapping around and maps ages to blocks. In addition to being easier to read, this notation avoids enumerating all blocks in $\mathcal{B}$. Furthermore, the number of sets implicitly encodes the associativity of the cache.

$$[\{set_0\}, \{set_1\}, \ldots, \{set_{k-1}\}] := \lambda b. \begin{cases} 0 & b \in set_0 \\ 1 & b \in set_1 \\ \ldots & \\ k & \text{otherwise} \end{cases}$$

2.2.2 Persistence Analysis

The two previous analyses operate on abstracted cache states, trying to prove or disprove the presence of a block in the cache at a given program point. However, this is not always enough. Consider Figure 2.1: For a cache of associativity 2, this loop can only have up to two misses. As soon as both elements have been loaded into the cache they will not

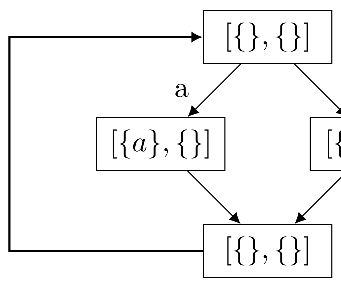


Figure 2.1: The must analysis predicts a miss in each loop iteration for this program. The loop can only produce one miss for a and one for b, independently of the number of iterations

get evicted and thus will *persist* in the cache. However, the must-analysis is unable to prove this. This is unsurprising, since both program locations can cause a cache miss. The observation, that both blocks can be missed only once *but we do not know when* is inexpressible in the framework of the must-analysis.

Persistence analysis takes a different approach. It does not decide between miss and hit locally by only considering the current state. Instead, it takes a more global perspective and gives bounds on the total number of cache misses in an execution of the loop, irrespective of where they are. In this framework, the above issue is resolved by stating that this loop only produces up to two misses whenever entered. This is realized by adding the following two constraints:

The edge corresponding to accessing a (resp. b) and then missing the cache can only be taken as often as the edges entering the loop from the outside.

Since this type of constraint is linear, it can be enforced by adding it to the Path Analysis ILP. See Section 3.4 for details.

In the literature, analyses defined by only considering local state are called *state-based*, while analyses that state properties about execution paths are called *trace-based* [10]. These names refer to the concrete domain the analysis abstracts from. It is outside of the scope of this thesis to formally define traces. We will therefore not give explicit concretization or abstraction functions for the persistence analyses.

The example above proves that limiting misses this way works in principle, yet it still remains unclear how to find persistent elements automatically. There are two basic techniques: *conflict sets* (also called conflict counting) and *conditional must* (both first published in [11]). Both are based on the idea that a block b is persistent if no valid path through the loop that starts with an access to b and ends with the eviction of b. In LRU, an element is evicted if k different blocks are accessed without an intervening refresh of b. It thus suffices to prove that there is no valid path that starts at an access to b and then encounters k different blocks without encountering b again.

Conflict sets

Conflict sets exploit that the processor needs to access *different* blocks to evict b. These different blocks are also called *conflicting blocks* since they compete with b for space in the cache. If we count the number of conflicting blocks and notice that there are less than k on any path, eviction of b is impossible.

As an example, consider again Figure 2.1. A conflict set analysis yields the set $\{a, b\}$ in all program points. Therefore, both a and b are persistent for any cache of associativity ≥ 2.

There are two flavors of conflict-set based persistence analysis: First, one maintains a single global conflict set in an all-or-nothing approach - if the loop only accesses $\leq k$ different memory blocks, the analysis classifies all blocks as persistent, else none. If a more expensive analysis is acceptable, one can also track a conflict set per memory block, the so-called block-wise conflict set approach. Whenever the block itself is accessed the conflict set can be cleared as we are only interested in paths without an intervening refresh to b.

Conditional must

Conditional must exploits that the processor needs to access k blocks to evict b. Consequently, the analysis counts how many blocks have been accessed since the last access to b. If this value never reaches k, b cannot be evicted and must therefore be persistent.

The analysis domain is $\mathcal{B} \to \mathbb{N}_{\leq k} \cup \{-\infty\}$. The $-\infty$ serves as the initial age, signaling that a block has never been encountered before. The update function counts all intervening accesses without remembering the block that caused the aging.

$$update(S, x) = \lambda b. \begin{cases} 0 & x = b \\ \min(k, S(b) + 1) & \text{else} \end{cases}$$

This update function also hints at the origin of the name: it is almost identical to the must update function. The join is identical and consists of taking the maximum of both counts.

How is this analysis different from the normal must? The decisive difference is the initial state of the blocks. The must analysis initializes blocks with the *maximal* possible age

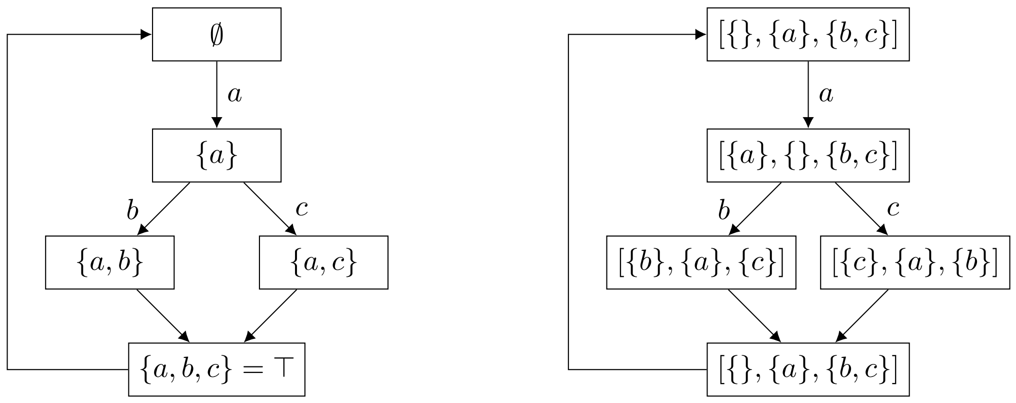


FIGURE 2.2: Conflict sets do not dominate conditional must. Note that only the last iteration is shown

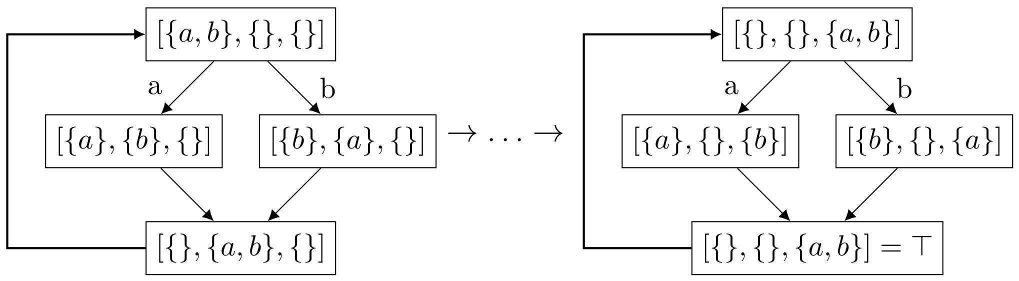


FIGURE 2.3: Conditional must does not dominate conflict sets. Note that this is the introductory example for which conflict sets were able to prove persistence for a and b.

while the conditional must analysis initializes blocks with the *minimal* possible age. This is only sound because we change our interpretation of the state as well: the analysis bounds the age under the *condition* that the block has already been seen (i.e. has been loaded into the cache). In particular, the analysis can claim anything as long as the precondition is not fulfilled.

Surprisingly, none of the two analyses dominates the other. Figure 2.2 and 2.3 show two programs where persistence is proven by one analysis but not the other. Only running both analyses ensures maximal effectiveness.

2.2.3 Persistence Scopes

So far we only talked about single loops. However, applying persistence analysis to an entire program is bound to fail as memory blocks rarely remain in the cache for so long. Instead, persistence analysis is often performed on contiguous parts of the program called *persistence scopes*. There is no restriction on what constitutes a scope, only that the probability of benefiting from persistence analysis is high enough to justify the analysis effort. In our timing analyzer, all loop bodies are persistence scopes – they are executed

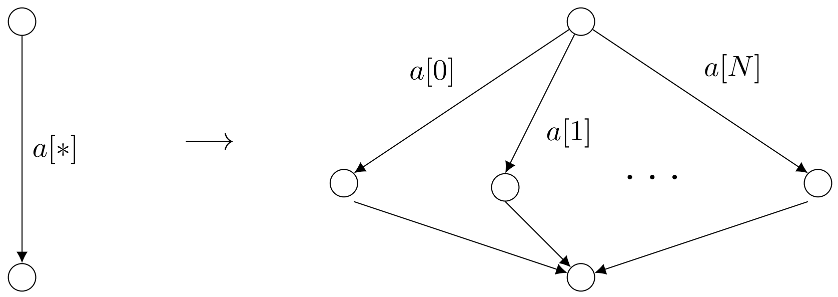


Figure 2.4: Handling Array accesses

more than once and likely reuse memory blocks in each iteration. More sophisticated heuristics are possible.

This idea of running the analysis on small parts of the program is only enabled by the conditional nature of the analysis. An unconditional analysis, like the must analysis, starts in the *worst* possible state to accommodate an arbitrary and unknown history. It is essential to remember as much history as possible and to avoid resetting the state. Consequently, these analyses are performed on the entire program.

A Conditional analysis, on the other hand, does not claim anything about the unknown history and therefore starts in the *best* possible state. Consequently, there is no harm in starting and stopping the analysis anywhere in the program. The results of the conditional analysis only apply to the part of the program the analysis ran on. The persistence scope heuristic therefore has to make the scopes large enough for the persistence constraint to matter but small enough to allow persistence at all.

2.3 Unknown Accesses

All analyses considered so far had an update function of the form $state \rightarrow \mathcal{B} \rightarrow state$, which implicitly assumes that we know the memory block we are accessing. Consider the following program:

```
int index = read_sensor();
return array[index];
```

Static analysis is unable to determine which memory block the return statement accesses because the value of *index* is only known at runtime. However, the access clearly influences the cache so it is unsound to ignore it. We define an update function to handle this case.

As shown in Figure 2.4, this new update function takes a set of blocks instead of a single block and joins all potential results.

$$update(S, B) = \bigcup_{b \in B} update(S, b)$$

However, this definition is hard to compute and hard to reason about. For specific analyses, we give simpler but equivalent formulas:

Must	$update(S, B) = \lambda b.min(S(b) + 1, k)$
Conditional must	$update(S, B) = \lambda b.min(S(b) + 1, k)$
May	$update(S, B) = \lambda b. \begin{cases} 0 & b \in B \\ S(b) & \text{otherwise} \end{cases}$
Conflict counting	$update(S, B) = S \cup B$

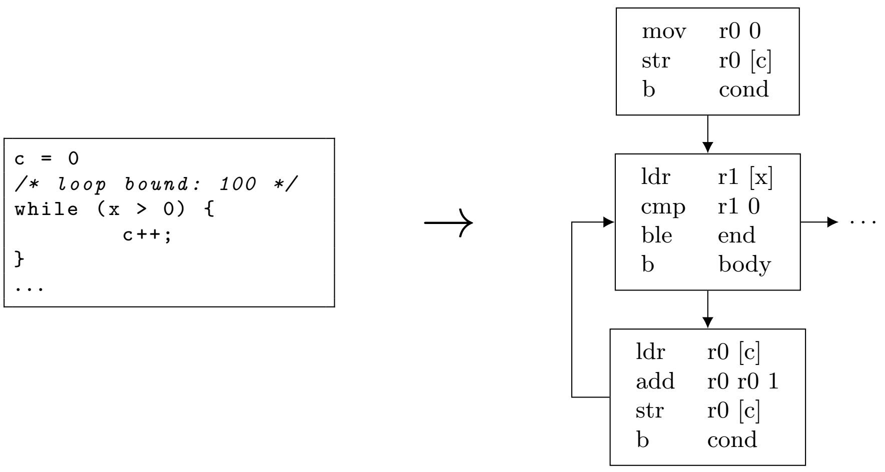
These unknown accesses pose a significant danger to precision: Especially for the latter two, the loss of information is enormous. One of the central goals of this thesis is to make these accesses more tangible and reduce their harmful impact on analysis precision.

Chapter 3

WCET Analysis by Abstract Interpretation

In this chapter, we present our timing analyzer *llvmta*. All analyses developed in this thesis were implemented and evaluated using this tool. It supports the *ARM* instruction set. As an academic tool we chose not to model any particular hardware. Instead our target hardware is freely configurable and supports different generic hardware concepts.

In order to understand the inner workings of our analyzer, we follow a small example program as it passes through the four stages of analysis.



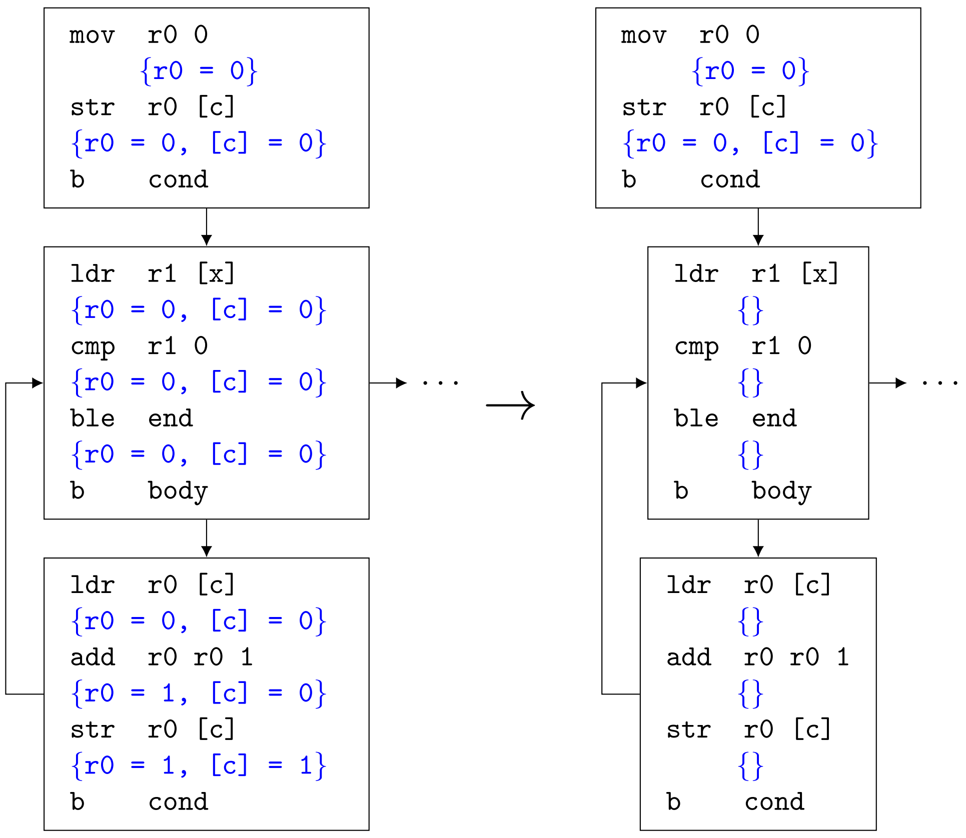
```
c = 0
/* loop bound: 100 */
while (x > 0) {
        c++;
}
...
```

```
mov   r0 0
str   r0 [c]
b     cond
```

```
ldr   r1 [x]
cmp   r1 0
ble   end
b     body
```

```
ldr   r0 [c]
add   r0 r0 1
str   r0 [c]
b     cond
```

Figure 3.1: Compilation



```
mov   r0 0
     {r0 = 0}
str   r0 [c]
{r0 = 0, [c] = 0}
b     cond

ldr   r1 [x]
{r0 = 0, [c] = 0}
cmp   r1 0
{r0 = 0, [c] = 0}
ble   end
{r0 = 0, [c] = 0}
b     body

ldr   r0 [c]
{r0 = 0, [c] = 0}
add   r0 r0 1
{r0 = 1, [c] = 0}
str   r0 [c]
{r0 = 1, [c] = 1}
b     cond
```

```
mov   r0 0
     {r0 = 0}
str   r0 [c]
{r0 = 0, [c] = 0}
b     cond

ldr   r1 [x]
        {}
cmp   r1 0
        {}
ble   end
        {}
b     body

ldr   r0 [c]
        {}
add   r0 r0 1
        {}
str   r0 [c]
        {}
b     cond
```

Figure 3.2: Value Analysis

3.1 Compilation

Most timing analyzers perform their analysis on the machine language binary, applying decompilation techniques to recover high-level properties like control-flow graphs. However, performing timing analysis as part of the compilation has many advantages: we can easily exploit the high-level description of the program and reuse analyses already performed by the compiler. Our analyzer is therefore implemented inside the *LLVM* compiler framework [12].

This first step transforms the example code into ARM assembly language (Figure 3.1). High-level features (like global variable names and the control flow graph) have been retained. The program is now in a form our analysis can operate on.

3.2 Value and Loop Bound Analysis

We then perform a value analysis (Figure 3.2). In our tool, this is a constant propagation analysis on register contents and memory. The most important class of constants are addresses – knowing the target address of memory accesses is paramount (Section 2.3).

The other purpose of the value analysis is to determine *loop bounds*, i.e. upper bounds on the number of loop iterations. Although loop bounds can and sometimes must be specified manually, the analysis automatically infers them where possible as a matter of convenience. Our tool uses LLVM's *Scalar Evolution* analysis pass for this purpose.

The result is depicted in Figure 3.2. Our example program is tagged with constant value annotations. The example also demonstrates the fixed-point iteration technique: in the initial iteration, the analysis determines wrong information that is corrected by the subsequent iterations.

3.3 Microarchitectural Analysis

The program is analyzed at the microarchitectural state level (Figure 3.3). For the sake of clarity we only present a data cache must analysis, ignoring other components like instruction caches and pipelines. Every memory access that cannot be unanimously classified as a hit leads to a state graph split. Afterwards, the states are joined again in order to keep the number of states at bay. Where a hit can be guaranteed (e.g. the loop body), the graph stays linear and the hit access time can be assumed.

In reality, the states not only contain the must cache state but also pipeline state, must and may cache state for the instruction and the data cache, persistence state for the instruction and data cache, etc. Computing one state graph for each component and combining the results in the end unfortunately introduces too much overestimation to be worthwhile [13].

The state graph is further complicated by *trace partitioning* [14], which we use to peel the first loop iteration (*virtual loop peeling*) and to distinguish multiple calls of the same function. For the sake of simplicity we ignore trace partitioning in this thesis.

3.4 Path Analysis

Finally, the tool determines the worst-case path through the state graph (Figure 3.4). In order to do so, we encode the state graph as an integer linear program (ILP) and then maximize the execution time. We encode the graph by a *frequency variable* $f(e)$ for each edge indicating how often the edge is taken and a *path constraint* for each node, forcing the result to be a valid path (i.e. if the program flow enters a node n times it also has to leave the node n times). The loop bounds constrain the number of times a backwards edge can be taken to be proportional to the number of times the loop is entered. Other analyses may add further constraints, e.g. persistence constraints.

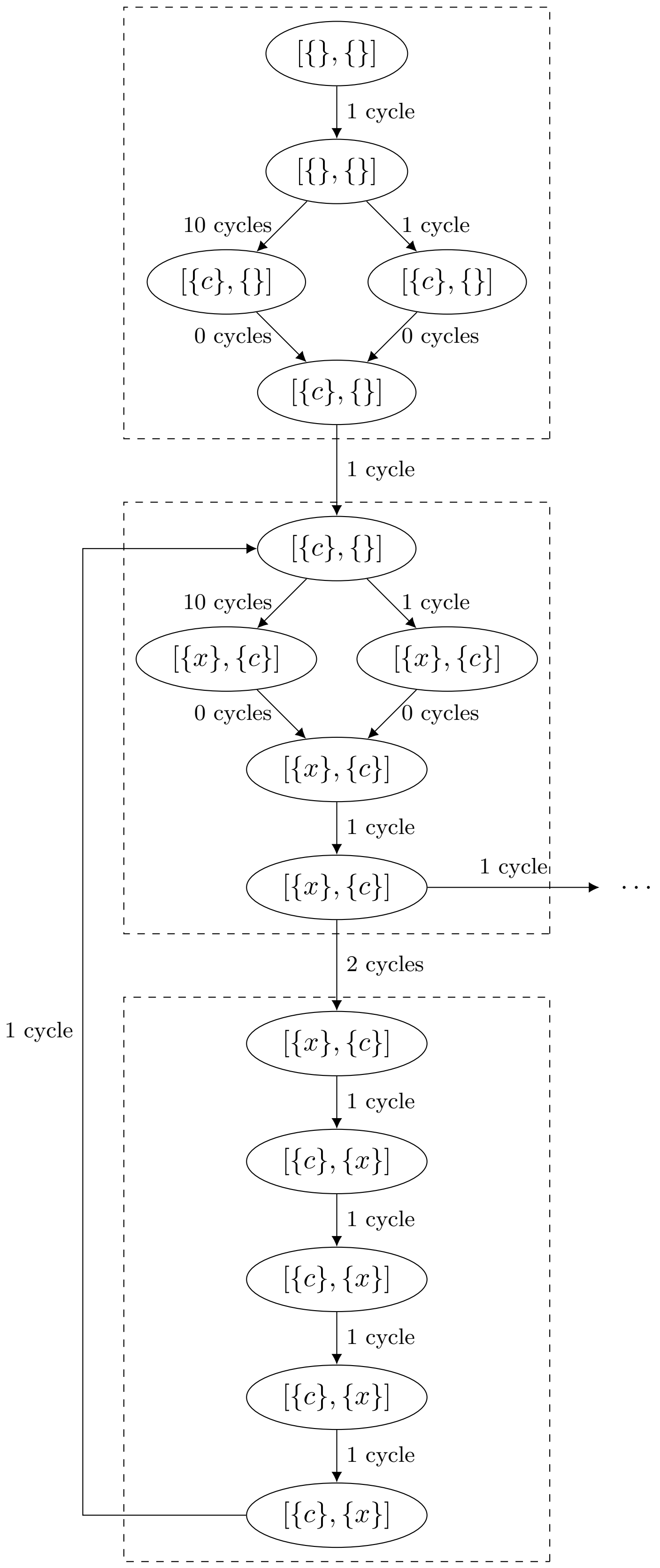
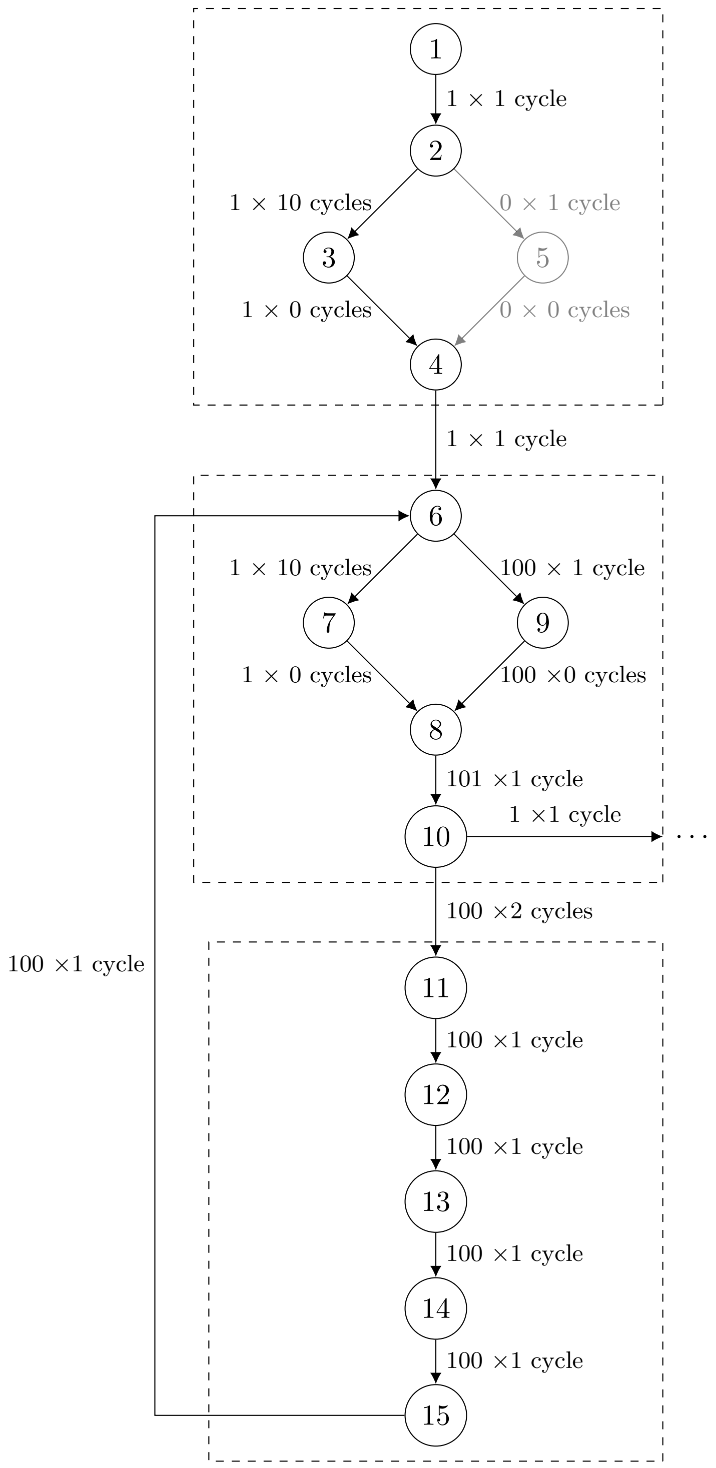


Figure 3.3: Microarchitectural Analysis



maximize $\sum_{e \in edges} cost(e)f(e)$

subject to:

Flow Constraints

$$\ldots$$
$$f(4,6) + f(15,6) = f(6,7) + f(6,9)$$
$$\ldots$$

Loop Bound

$$f(15,6) \leq 100f(4,6)$$

Persistence Constraint

$$f(6,7) \leq f(4,6)$$

Start Constraint

$$f(1,2) = 1$$

Figure 3.4: Path Analysis

Chapter 4

Write-back Cache Analysis

Despite their performance benefits, write-back caches are mostly ignored in static WCET estimation. The main reason is that write-back caches are more difficult to analyze. Knowing when a store ends up accessing memory requires knowing when the corresponding cache line is evicted from the cache. Unfortunately, there are often multiple possible eviction points due to uncertainty about the state of the cache. Without a dedicated analysis, the analysis is forced to assume a write back on each possible eviction point (i.e. on each cache miss). This significantly overestimates the WCET, defeating the point of using a write-back cache in the first place.

In 1996, Alt et al [9] published the first and to our knowledge only abstract interpretation-based write-back analysis. It is based on the must and may analyses and attempts to track which cache lines are dirty and which are clean. However, 8 years later Stephan Thesing states that "Write-back cannot be analyzed precisely due to the bad worst-case behavior" [15, p. 69]. And indeed, as we will see in the evaluation, the analysis is often ineffective. In this thesis we develop and evaluate a novel analyses for write-back caches. In addition to the "dirtiness analysis", a slight variation of the analysis by Alt et al., we perform a trace-based analysis we call the "store bound". It exploits that all write backs are only delayed executions of earlier stores. The number of stores in the program thereby bounds the number of write backs. Since this observation is a linear constraint over edge frequencies, the path analysis can efficiently enforce it.

Both analyses give poor results on their own. They unlock their true potential when combined. Storing to an already dirty cache line cannot cause additional write backs, allowing us to ignore these stores in our bound. Our analysis therefore employs the dirtiness analysis to identify provably dirty cache lines in addition to its original purpose.

Notation Write-back caches have to differentiate between loads and stores. The following analyses will therefore have *two* update functions instead of one: The function

$update_L$ performs a load of the given block while the function $update_S$ performs a store to the given block.

4.1 Dirtiness Analysis

One central purpose of the Dirtiness Analysis is to statically prove or refute whether a given memory access causes a write back. This part is already handled by Alt et al.'s analysis [9], which contains three parts:

1. A function mapping blocks to a boolean lattice $\{C, \top\}$[1], (where C stands for clean and $\top$ stands for potentially dirty),
2. A must analysis, and
3. A may analysis.

For notational convenience we only describe the update function for the first part and assume that the may and the must analysis are updated independently. We use the terms *S.may* and *S.must* to refer to the state of the may and must analysis *before* the current update. Furthermore, we define the following helper function:

$$victims(S, x) := \{b \in \mathcal{B} | S(b) < k \wedge update(S, x)(b) = k\}$$

The analysis follows two rules: a cache element is clean until there is a store to it and it becomes clean again when it is provably evicted (i.e. evicted from the may state). Formally:

$$update_L(S, x) = \lambda b. \begin{cases} C & \text{if } b \in victims(S.may, x) \\ S(b) & \text{else} \end{cases} \tag{4.1}$$

$$update_S(S, x) = \lambda b. \begin{cases} \top & \text{if } b = x \\ update_L(S, x) & \text{otherwise} \end{cases} \tag{4.2}$$

The join is the point-wise join of the functions.

The core part of the analysis is the write-back classification function, which states whether an access writes back. We classify an access as potential write back if it might evict a potentially dirty element.

[1]Names are changed to our notation for consistency.

$$\textit{may-wb}(S, x) = \exists b \in \mathcal{B}.\ \textit{may-evict}(S, x, b) \wedge S(b) \neq C$$

The predicate *may-evict*(S, x, b) states whether accessing x might evict b from cache state S. In our implementation

$$\textit{may-evict}(S, x, b) = S.\textit{may}(b) < k \wedge \textit{update}(S.\textit{must}, x)(b) = k$$

Improving this classification (e.g. by taking persistence into account) remains future work.

The dirtiness analysis also identifies provably dirty cache lines. For this purpose, the per-block classification is supplemented with an additional lattice element D ("dirty"). To obtain a lattice, we additionally define a $\bot$ element. This is just a technicality – no block is ever mapped to $\bot$. With $\top := \{D, C\}$ and $\bot := \emptyset$ our domain is $\mathcal{P}(\{D, C\})$ ordered by the subset-relation.

The update function changes in two ways: First, we assign an element to D if it is written to. Second, a block might become clean if it is evicted from the must cache as only cached blocks can be dirty.

$$\textit{update}_L(S, x) = \lambda b. \begin{cases} C & \text{if } b \in \textit{victims}(S.\textit{may}, x) \\ S(b) \cup C & \text{if } b \in \textit{victims}(S.\textit{must}, x) \wedge b \notin \textit{victims}(S.\textit{may}, x) \\ S(b) & \text{otherwise} \end{cases} \tag{4.3}$$

$$\textit{update}_S(S, x) = \lambda b. \begin{cases} D & \text{if } b = x \\ \textit{update}_L(S, x) & \text{otherwise} \end{cases} \tag{4.4}$$

The *may-wb* classification function remains unchanged. For now, the only improvement due to the D-classification is the ability to detect guaranteed write-backs.

$$\textit{must-wb}(S, x) = \forall b \in \textit{blocks}.\ \textit{may-evict}(S, x, b) \Rightarrow S(b) = D$$

4.2 Dirtifying Stores

The biggest weakness of the dirtiness analysis becomes apparent in the following example program. Assume a directly mapped cache where x, a, b, and c map to the same cache set.

```
/* The cache is clean at this point */
store(x);
if (...) load(a);
if (...) load(b);
if (...) load(c);
...
```

Our dirtiness-analysis states that every single memory access can be a write back. This is impossible since the cache only defers stores, it does not generate new ones. With only one store, there can only be one write back. In general, the number of write backs in the program is upper-bounded by the number of stores in the program.

The constraint is as a linear relationship between state graph edges: the sum of all edges representing stores is larger than the sum of all edges representing write backs. We formulate this constraint in the path analysis ILP, making the solver pick the worst-case locations for the write backs automatically. All other accesses are safely assumed not to write back. We write down the constraint formally, with *WB* referring to the set of write-back edges and *STR* referring to the set of store edges:

$$\sum_{e \in WB} f(e) \leq \sum_{e \in STR} f(e) \qquad \text{(store bound)}$$

The above constraint is only applicable in limited circumstances, as the cache is assumed to be clean at the beginning of the program. We refer to this as the *initial cleanness assumption*. In a multi-tasked system (which includes most real-time systems) it is unfortunately false. Previous tasks might have deferred some stores beyond their own runtime by leaving dirty data in the cache.

There are two ways to handle task interference. First, one can assume it does not happen and account for it later. This usually involves a *response time analysis* that knows about the different tasks in the system and determines how they can interfere. In the context of the write-back analysis, this means that the WCET analysis assumes a clean cache while the higher-level analysis determines how many and which additional write backs can occur.

The alternative is to already assume arbitrary interference during the WCET analysis. In our case, this means that the dirtiness analysis has to initially assume that all cached blocks might be dirty. We also generalize the store bound to take third-party induced write backs into account.

$$\sum_{e \in WB} f(e) \leq \sum_{e \in STR} f(e) + IDCB \qquad \text{(generalized store bound)}$$

IDCB is an upper bound on the *i*nitially *d*irty *c*ache*b*locks. In our case it will always be the size of the cache, but a more sophisticated analysis might be able to determine a better bound.

While the store bound successfully limits the number of write backs, it fails to make a strong argument for write-back caches. After all, a write-through cache causes one memory access per store as well while being much simpler to analyze. Where is the advantage of the write-back mechanism? Let us recall the core idea of write-back: when a store happens, the result is not written to memory immediately. Instead, the modified data is kept in the cache until the cache line is evicted, getting further stores into this cache line *for free*. To prove that these free stores happen (which is the only way to carry over the performance benefit into WCET reductions), we need to identify provably dirty cache lines. Luckily, we already have the tool for this: the dirtiness analysis. The analysis can separate stores that go to provably dirty cache lines from stores to potentially clean cache lines (which we call "dirtifying stores" as they might turn a cache line dirty). As the former stores are for free, the ILP constraint only needs to consider the latter ones. This insight leads to the final ILP constraint (*DFS* being the set of dirtifying store edges):

$$\sum_{e \in WB} f(e) \leq \sum_{e \in DFS} f(e) + IDCB \qquad \text{(dirtifying store bound)}$$

4.3 Evaluation

In this section, we evaluate our write-back analysis. First, we compare the bounds on a write-back system with the bounds on a write-through system. We show that write-back caches lower bounds by 20% or more in many cases, making write-back caches not only possible but also desirable for hard real-time systems.

Next, we evaluate the initial cleanness assumption. We define *cleanup costs* as the number of cache lines still dirty at the end of the program. Taking these hidden costs into account, we show that neither assumption consistently yields better results. The optimal choice varies from program to program.

Third, we show that the dirtifying store bound is superior to both its constituents. To this end we also prove our claim that the dirtiness analysis is ineffective on its own.

Fourth and last we evaluate the analysis cost of our write-back analysis. We show that the analysis introduces reasonable cost (it takes twice as long in the worst observed testcase) and is sometimes even cheaper than a write-through analysis.

Preliminaries

Our evaluation is based on the Mälardalen benchmark suite [16]. We additionally generated seven testcases[2] with *SCADE* [17], an industry-strength commercial model-based design tool.

For reasons of space and clarity, we cannot present all testcases for a given measurement. However, omitted benchmarks are still included in the geometric mean. The complete evaluation data can be found online[3]

All evaluation results were generated by our timing analyzer *llvmta*. For this evaluation, we model a 2-way and an 8-way LRU data cache with a cache line size of 16 bytes and 32 cache sets (for a total of 1 and 4 KiB). The modelled machine contains a 5 stage inorder pipeline and a dedicated 2-way LRU instruction cache (also with a line size of 16 bytes and 32 cache sets). Unless explicitly stated otherwise, we assumed nothing about the initial cache state.

Since our analyzer cannot handle some of the more involved features used by the LLVM optimizer, all testcases are compiled without optimization. This is not necessarily unrealistic [18]. However, the lack of efficient register allocation frequently causes needless spills and reloads. While a write-back cache can handle both accesses inside the cache, a write-through cache waits for memory on each register spill. For optimized programs, our evaluation should therefore be taken with a grain of salt.

4.3.1 Write-back versus Write-through

Before delving into any details, we first present that the dirtifying store bound reduces the WCET bounds compared to the write-through case (Figure 4.1). The y-axis is normalized to the WCET bound of the write-through case. The bars show the bound increase in percent (i.e. lower is better). The graph shows that most testcases profit from the write-back cache, with an average bound reduction of 15-20%. The graph also exhibits two surprising properties:

Increasing the cache size sometimes increases the worst-case execution time. This can be observed in *insertsort*, for example. The reason for this increase is that we assume nothing about the initial cache. In particular, the dirtifying store bound assumes it has to write back the entire initial content of the cache unless it can prove otherwise. Increasing the size of the cache from 1 KiB to 4 KiB adds 192 potentially

[2] *cruise-control, digital-stopwatch, es-lift, flight-control, pilot, roboDog, trolleybus*
[3] `http://embedded.cs.uni-saarland.de/data/tblass_master.tar.gz`

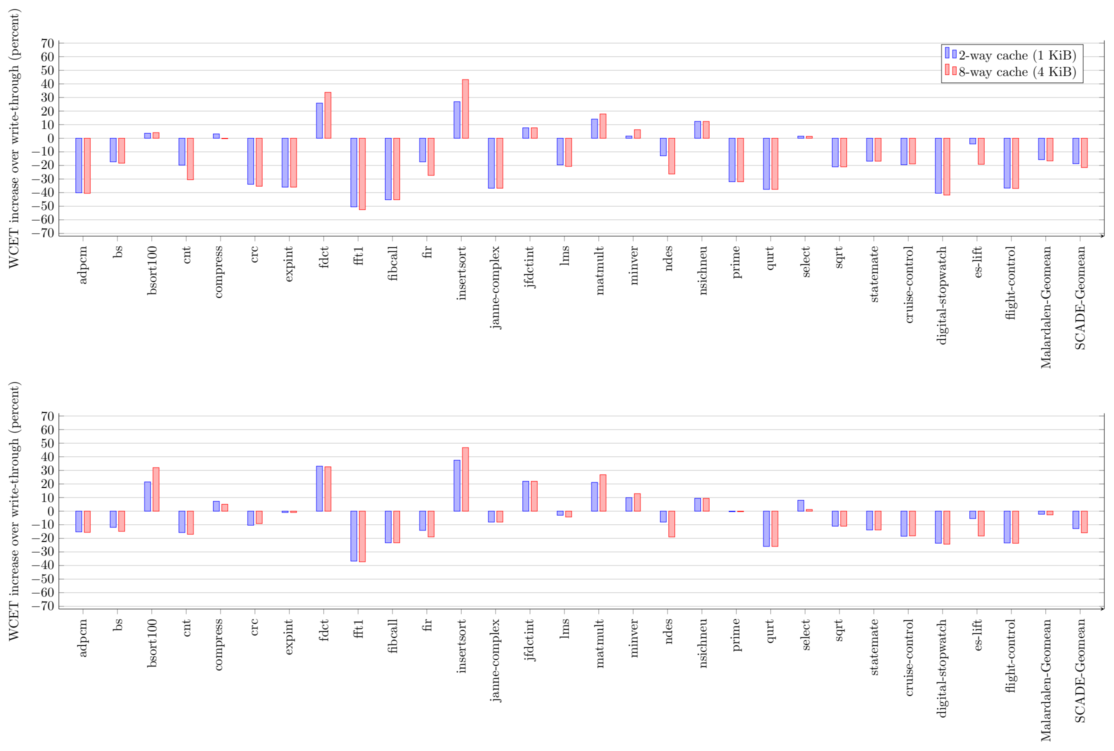


FIGURE 4.1: Effectiveness of the write-back analysis. Both diagrams are normalized to the write-through case. Top: blocking stores, Bottom: unblocked stores

dirty cache lines. However, virtually all memory accesses in *insertsort* are unknown and therefore do not benefit from the write-back policy. In the end, the analysis predicts 104 additional write backs (the other 88 lines are never written back). These additional write backs cancel out the reduced number of misses caused by the larger cache. The write-through system, on the other hand, is only affected positively by the larger cache and therefore improves compared to the write-back system.

Write-through often achieves lower bounds than write-back. This mainly concerns the testcases that also become worse with a greater cache size. Figure 4.2 shows the underlying cause: the left side contains the testcases that worsened under the write-back analysis while the right side contains the testcases that improved or stayed the same. Performing badly under write-back apparently correlates with performing many unknown accesses. The testcases on the right side that also have low address precision tend to have only small improvements as well. This difference is probably due to unknown stores. In a write-through cache, the cache state is only influenced by the store if the access was a cache hit. An unknown store can therefore never evict an element from the cache. For a write-back cache, on the other hand, an unknown store is disastrous: not only does it fail to gain any advantage over write-through (an unknown store is

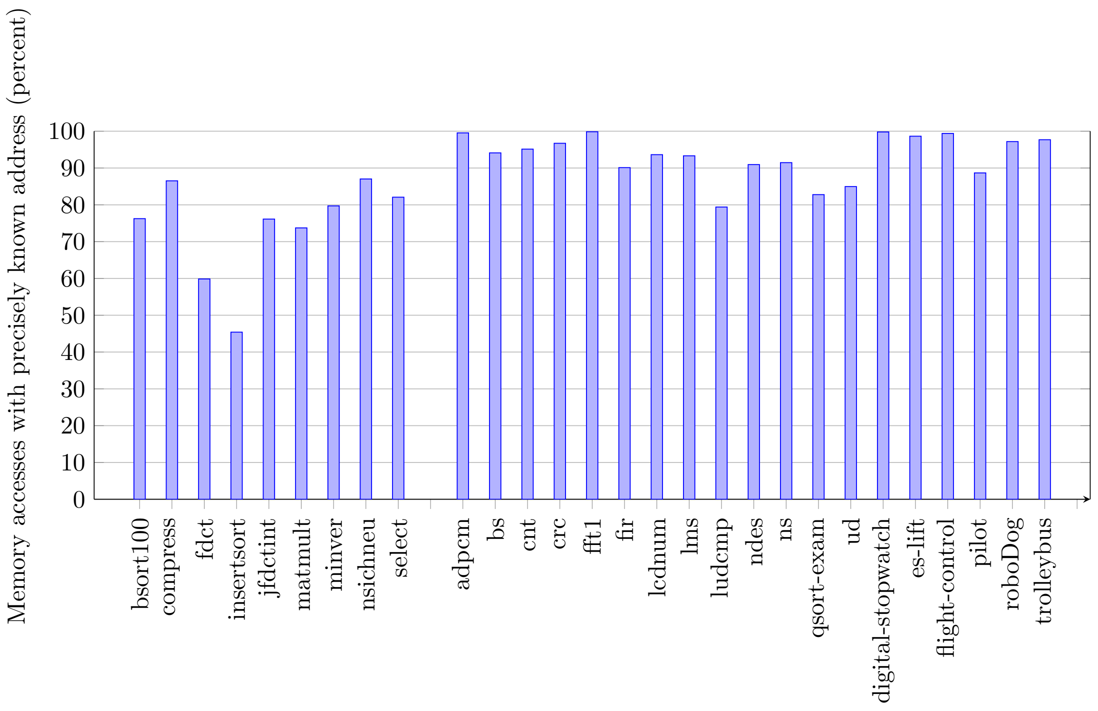


Figure 4.2: Diagram of the address precision distribution

always dirtifying), it also has to mark *all* memory blocks as potentially dirty, crippling the dirtiness analysis.

The bottom part of Figure 4.1 contains the WCET bounds if the *unblocked stores* option is enabled. In the default machine model the processor waits for stores to complete. The unblocked stores option allows the processor to asynchronously schedule the store and immediately continue execution. It only has to wait if another memory access is performed before the store completes, as only one memory access can be in flight at any time.

Allowing the processor to do useful work instead of waiting for the store reduces the lead of the write-back system. Write-through now performs its stores almost for free as well, assuming the stores are sufficiently far away from other memory accesses. Write-through therefore benefits more from the unblocked stores than write-back. The graph clearly shows that the write-back bound grows relative to the write-through bound.

In conclusion, we claim that using a write-back cache together with our analysis improves the bounds on all programs that do not have too many unknown accesses. Our evaluation also shows that adding a store buffer already achieves a sizeable WCET improvement while avoiding the complexity of write-back caches.

It is striking, though, that testcases like the insertsort benchmark perform so poorly under write-back caches. The testcase consists of two nested loops that operate on a 44

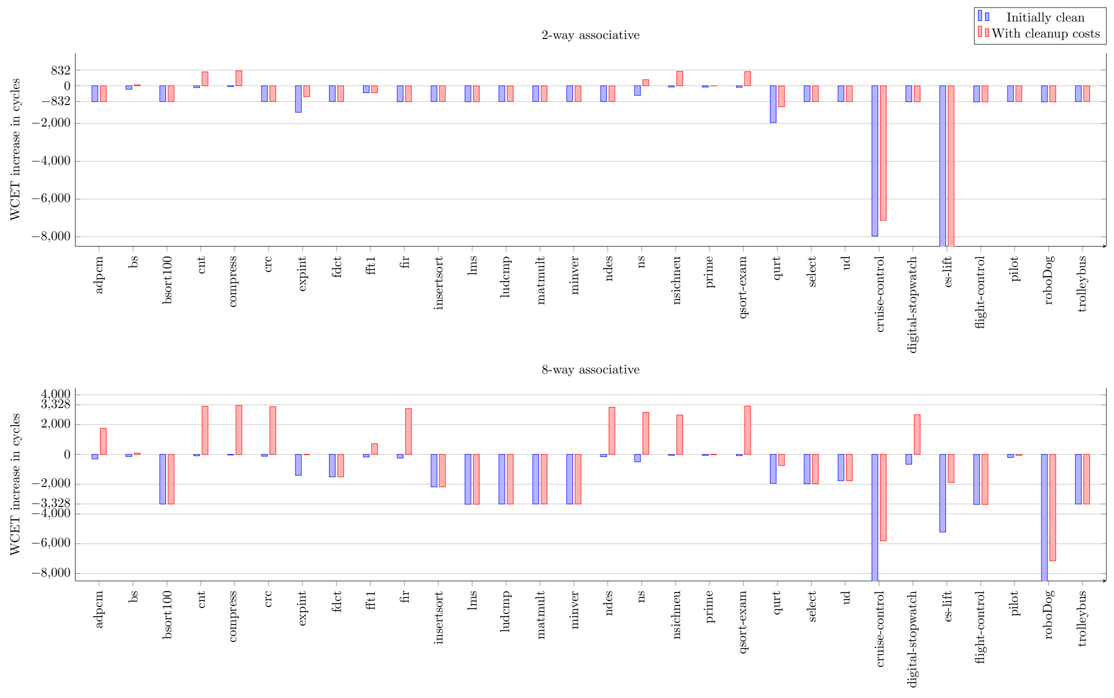


Figure 4.3: The effect of assuming an initially clean cache. The baseline is the write-back WCET assuming a potentially dirty initial cache. The left bar shows the change of bound after assuming the initially clean cache while the right bar additionally accounts for the cost of writing back all outstanding dirty lines at the end of the program

byte array. The program does not touch more than three cache lines and extensively writes on them. One would expect write-back caches to excel in this scenario; they handle all stores in the cache, thereby reducing the number of memory access from over a hundred to three. In the next section, we present an array-aware cache analysis that addresses this issue.

4.3.2 The Initial Cleanness Assumption

In Section 4.2 we presented two possible assumptions for the initial cache state. Either we assume the cache is initially clean or we assume the cache might initially contain dirty cache lines (i.e. assume nothing). In this section, we evaluate the effect of these assumptions on the WCET bound.

The WCET of the program only increases if we assume the cache might contain dirty lines. However, it does not make sense to directly compare these WCET bounds anyway: assuming a clean cache and leaving dirty lines in the cache allows a program to avoid ever storing the data to memory. A rental car has to be returned at the same fuel level as when it was borrowed; the customer cannot use up all the fuel and then avoid ever filling the tank. In the same vein, programs should either accept a dirty cache in the beginning or clean it up in the end. This is accounted for in the *cleanup cost*, which states how many cache lines are still dirty by the end. Technically, we compute the cleanup cost during the path analysis(cf. Section 3.4) as

$$\min(FDCB, \sum_{e \in DFS} f(e) - \sum_{e \in WB} f(e))$$

FDCB (*F*inal *D*irty *C*ache*b*locks) is defined as the number of cache lines the dirtiness analysis considers potentially dirty at the end of the program. Figure 4.3 shows the WCET bound under both assumptions. The baseline assumes nothing about the initial cache. The left bar present the WCET under the clean-cache assumption, whereas the right bar additionally takes the cleanup costs into account. Note that the diagram shows the absolute difference in cycles rather than the relative difference we usually show to account for the different testcase sizes. In this case, the absolute value is more appropriate because the initial cache state represents a one-time cost that does not scale with program size.

We can see that many programs apparently account for writing back the entire cache. There are $32 \cdot 2 = 64$ lines in the cache and writing back one of them takes 13 cycles in our model[4], so writing back the entire cache takes $13 \cdot 64 = 832$ cycles, which is the most common bound reduction. We can also see that the cleanup costs do not make any difference in these cases. For those testcases, assuming an initially clean cache is a net win. For *es-lift* the difference is even too large to include into the diagram and amounts to 19000 cycles including cleanup costs. The reason for this extreme difference is that the dirtiness analysis is crucial for *es-lift* (Figure 4.4): on a 2-way cache, it refutes almost 90% of the potential write backs. Without the initial cleanness assumption, if only refutes about 2%. Testcases like *compress*, on the other hand, do not benefit at all from the assumption. For these testcases, assuming a clean cache is a net loss as it needlessly weakens the dirtifying store bound.

As we see in the lower part of Figure 4.3, increasing the associativity to 8 aggravates these effects. Since it quadruples the number of lines in the cache, writing back the entire cache now takes 3328 cycles. Consequently, the cleanup costs grow as well. About a third of the testcases perform worse under the clean-cache assumption. However, *roboDog* and *cruise-control* reduce their bound by more than 8000 cycles if the cache can be assumed clean. Apparently, it depends on the program whether assuming the cache to be initially clean increases or reduces the bound.

[4]9 cycles per memory access + 1 cycle per word

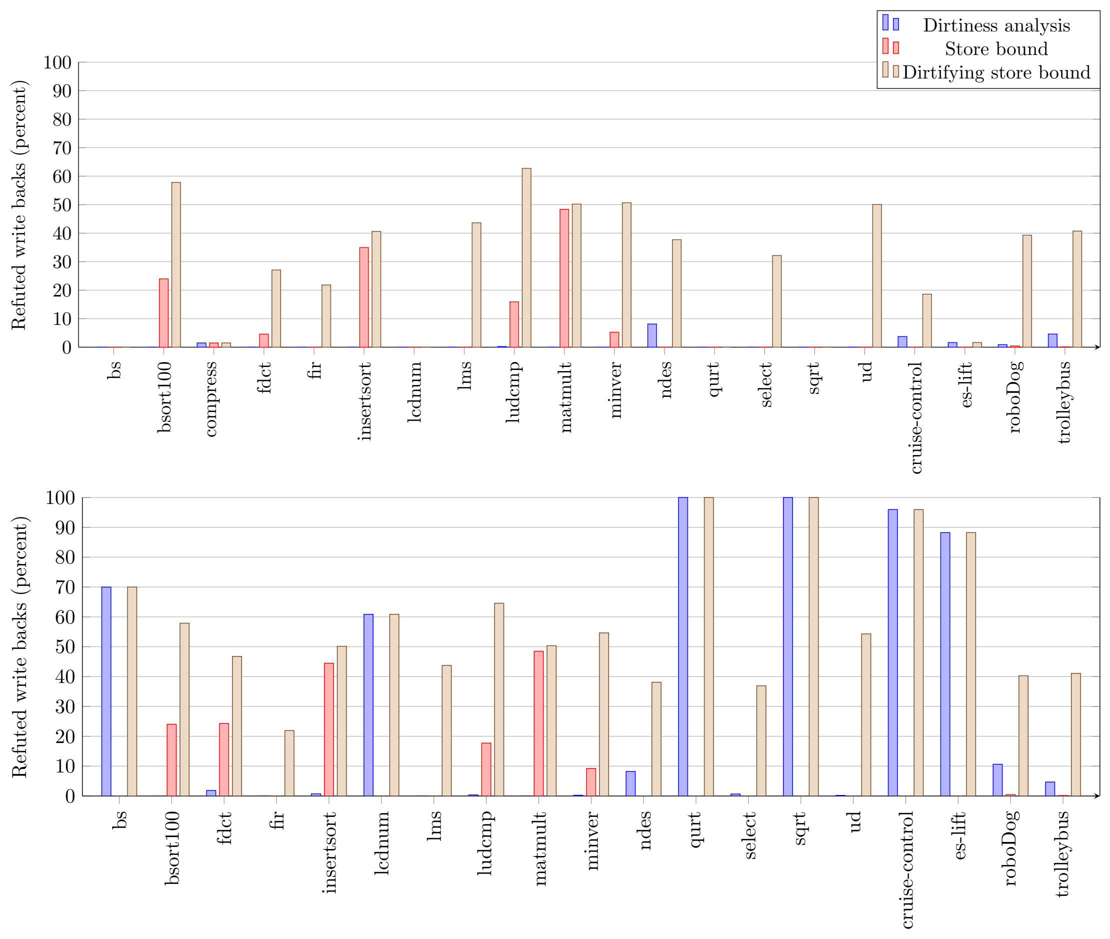
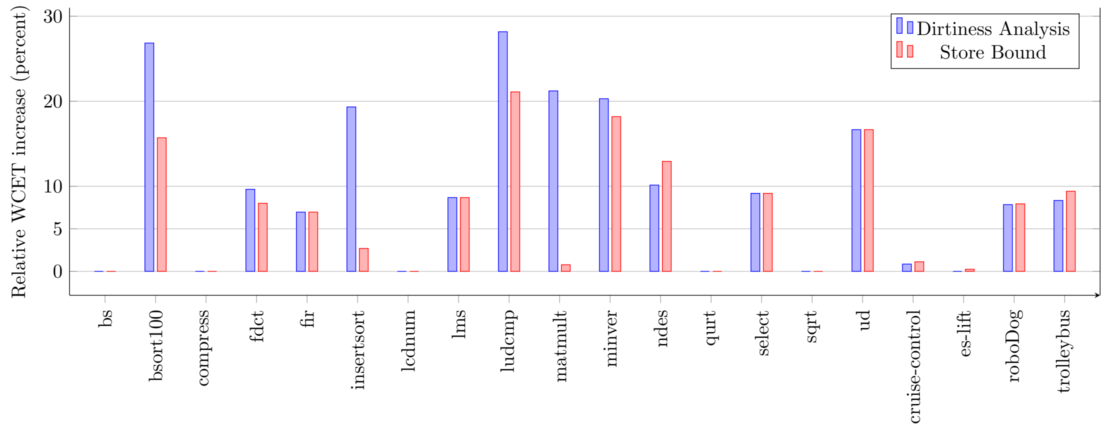


FIGURE 4.4: Comparison of the sub-analyses. The diagram shows for how many cache misses the analysis proved a write back impossible.
Top: Initially unknown, Bottom: Initially clean

4.3.3 The analysis components

After evaluating the analysis as a whole, we now focus on the two components of the dirtifying store bound: the dirtiness analysis and the store bound. The upper part of Figure 4.5 shows the bound increase compared to the combined analysis. Running the Dirtiness Analysis on its own means allowing the analysis to predict more write backs than the sum of stores in the program and initially dirty cache lines. Running only the store bound means that the analysis does not attempt to track dirty cache lines; it assumes that each store may cause a write back.

The graph clearly shows that combining the two analyses is beneficial. Both sub-analyses perform significantly worse in isolation. This confirms that the dirtifying store bound is necessary for tight WCET estimation: while the dirtiness analysis is too sensitive to unknown accesses, the store bound fails to take advantage of multiple stores onto the same cache line.



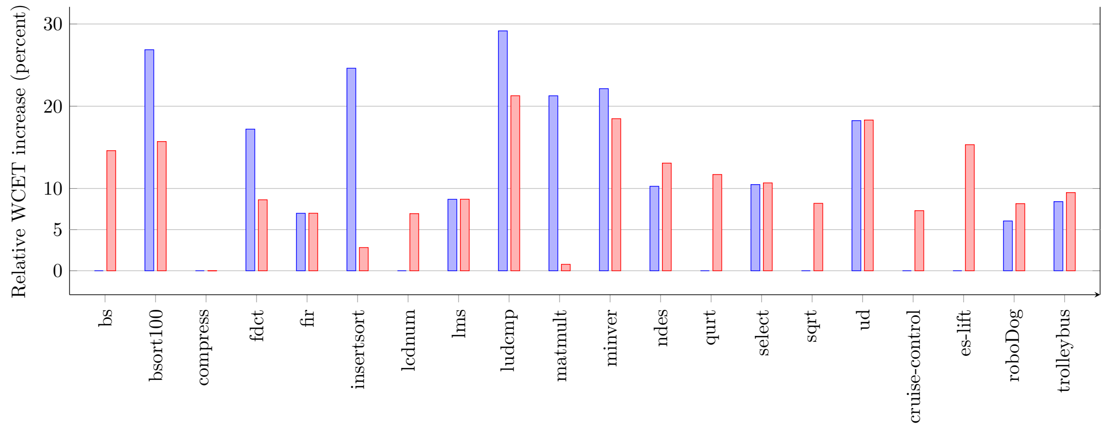


FIGURE 4.5: Relative WCET increase of the dirtiness analysis and the store bound compared to the dirtifying store bound. Testcases differing from the baseline by less than 5% have been left out. Top: initially unknown 2-way cache, Bottom: initially clean 2-way cache

Another observation is that the store bound acts as a safety net. If the dirtiness analysis produces extremely bad results (like in *insertsort*), the store bound prevents the bound from growing too high. In particular, the store bound never allows more memory accesses than the same program on a write-through, write-allocate cache.

4.3.4 Analysis Cost

Finally, we present the analysis runtime costs of our write-back analysis. Even though we did not focus on performance, it is important that the analysis is not prohibitively expensive. Figure 4.6 shows the change in analysis runtime. As expected, write back analyses takes longer in most testcases, taking on average about 15% additional runtime with a maximum of about 40%. The memory consumption (Figure 4.7) increases by averagely 3-5%, ranging up to 20% for some testcases. Interestingly, the memory consumption and the runtime are even reduced in some cases. The write-back system probably produces a state graph that allows for more joins and is consequently smaller.

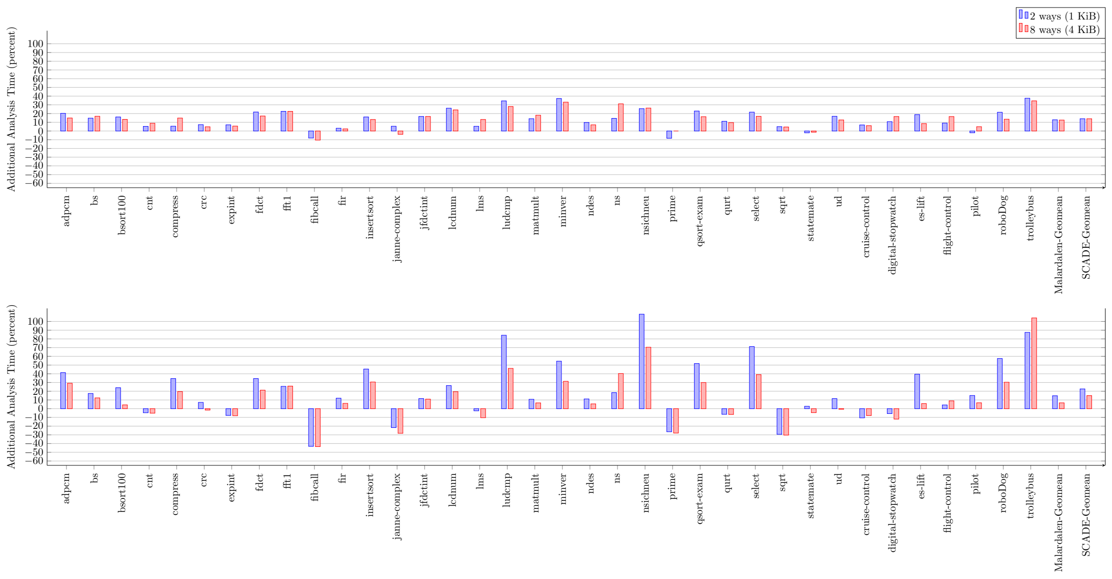


FIGURE 4.6: Analysis runtime. The graphs are normalized to the write-through case. Top: blocked stores, Bottom: unblocked stores

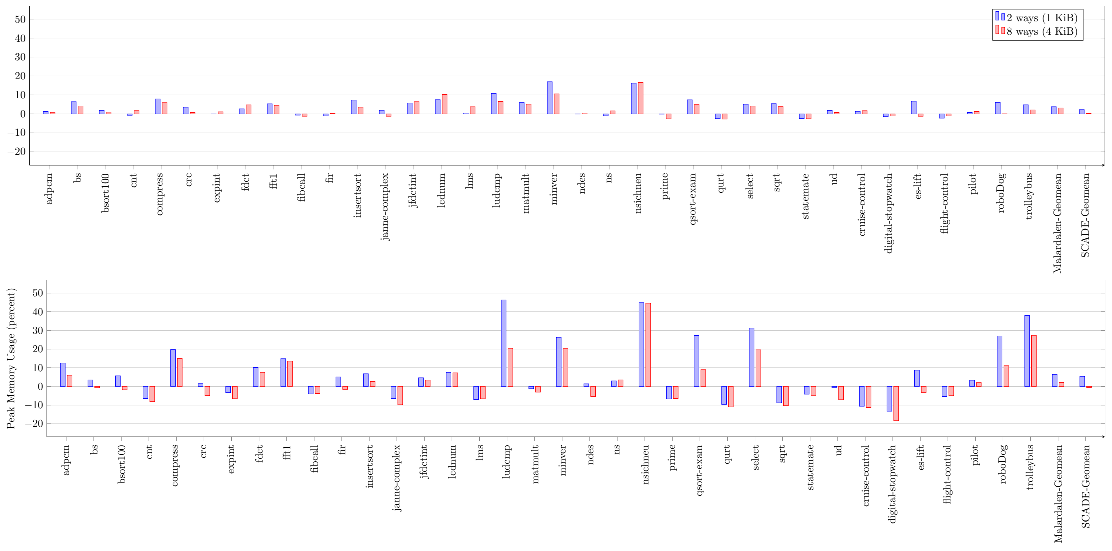


FIGURE 4.7: Peak memory consumption. The graphs are normalized to the write-through case. Top: blocked stores, Bottom: unblocked stores

Both, the path analysis and the microarchitectural analysis scale with the size of the graph; it therefore dominates analysis runtime.

We also present the changes in runtime and memory consumption if we enable the *unblocked stores* mechanism. Enabling this option has an important effect on the microarchitectural state graph: two microarchitectural states that disagree about the pending memory accesses cannot be joined. In particular, the *write-back* and *no write-back* paths are joined much later, increasing the computational load of the analysis. The bottom part of Figure 4.6 shows how the write back analysis performs under this machine model. The runtime cost stays at 10-20% on average, reaching up to 100% for some testcases.

The change mainly seems to increase the variance of the runtime. Memory consumption behaves similarly, confirming our hypothesis that a larger state graph is the underlying cause.

Chapter 5

Array-aware Cache Analysis

Real-world programs often contain memory accesses whose address is computed at runtime. These *dynamic* accesses occur in a variety of circumstances. The most common ones are (from less structured to more structured): absolute addressing, pointer accesses, and array addressing. Examples for each are shown in Figure 5.1. Absolute addressing only occurs in low-level code, which sometimes circumvents the type system and other high-level language features. Unless the address computations can be performed statically by the value analysis, there is no way to determine the target of the access. In this case we cannot do better than the general unknown access update (Section 2.3).

The other two access types are both core features of many higher-level languages. Pointers are a common abstraction for addresses and allow type-safe and platform-agnostic use of indirection. Arrays, a fundamental data structure in imperative languages, represent a fixed-size collection of equal elements. Unlike pointers, arrays are restricted to a statically known interval $[base, base + size]$ in a well-defined program[1]. Only the exact

[1]This observation might not hold on systems that load data speculatively. In Section 6 we discuss solutions to this problem

```
unsigned char Get_Data_Byte (uint32_t addr)
{
   return *(unsigned char*)(DATA_MEM_BASE+addr);
}
```

(a) Absolute Addressing

```
ptr = cond ? &x : &y;
value += *ptr;
*ptr = 0;
```

(b) Pointer Access

```
for (int i=0; i<N; i++)
        sum += array[i];
```

(c) Array Access

Figure 5.1: Kinds of dynamic accesses

destination of the access inside this interval is unknown. It is this restriction we exploit in our array-aware analyses. We therefore disregard pointers in this thesis.

Prior work on improving unknown-address handling has concentrated on devising new analyses capturing the relation between two accesses. Simon Wegener [19] developed an analysis that virtually unrolled loops to identify array accesses in subsequent iterations targeting the same block. In Figure 5.1(c), the analysis discovers that $a[i+1]$ targets the same block as $a[i]$[2] and therefore always hits the cache. No knowledge about the address $a[i]$ refers to is required.

Another branch of research focuses on understanding the array access pattern. Huynh et al. [20], for example, developed an analysis based on *Cache Miss Equations* [21]. By analysing the loop iteration variables, they determine fine-grained persistence information.

Both approaches are highly dependent on a regular loop structure. In contrast, our analyses make no assumptions about the shape of loops; they do not even recognize loops. Instead, they are based on *conflict sets* of arrays. By tracking conflicting array accesses, they restrict how often an array can age other memory blocks. In this thesis, we develop an array-aware must analysis based on this principle. We also develop an array-aware persistence analysis that is able to prove persistence of regular memory blocks as well as entire arrays. These analyses are less powerful since they are oblivious to the loop iteration order. In exchange, they are able to analyze structured as well as unstructured code without depending on any patterns.

Notation We refer to the set of all arrays by $\mathcal{A}$. An *array* is defined as a contiguous chunk of memory with a known starting address. It is important that accesses to this array can be identified. We define a function $\mathit{blocks} : \mathcal{A} \to \mathcal{P}(\mathcal{B})$ to map all arrays to the set of their blocks and the inverse function $\mathit{arrayof} : \mathcal{B} \to \mathcal{P}(\mathcal{A})$ that maps all blocks to the arrays it belongs to (potentially $\emptyset$). For now, we assume that no block belongs to more than one array. We discuss this restriction in Section 6.

In order to formalize abstract domains without excessive case distinctions, we additionally define *bounded multisets*. These are sets that might contain elements multiple times (i.e. $\{x\} \neq \{x,x\}$), but for each element there is a bound function that limits how often an element might occur (i.e. $\mathrm{bound}(x) = 2 \Rightarrow \{x\} \neq \{x,x\} = \{x,x,x\}$).

A bounded multiset S_M is modeled by a multiplicity function. The usual set operations are defined in Figure 5.2. To avoid confusion with traditional sets we add the subscript M to all multiset operations.

[2]Assuming i is aligned to the cache line and elements of a are smaller than cache lines.

$$\begin{aligned}
S_M &:= S \to \mathbb{N} && (5.1)\\
x \cup_M y &:= \lambda s.\, \min(x(s) + y(s), \mathit{bound}(s)) && (5.2)\\
x \cap_M y &:= \lambda s.\, \min(x(s), y(s)) && (5.3)\\
x \subseteq_M y &:= \forall s. x(s) \leq y(s) && (5.4)\\
x -_M y &:= \lambda s.\, \max(x(s) - y(s), 0) && (5.5)
\end{aligned}$$

FIGURE 5.2: Definition of bounded multisets

In addition, we define a *saturated* predicate that is true iff the multiset is saturated with a given element.

$$\mathit{saturated}(S_M, x) :\Leftrightarrow (S_M \cup_M \{x\} = S_M) \tag{5.6}$$

5.1 Improving the Must Analysis

First, we present how dynamic array accesses affect the traditional must analysis and point out the specific problems we want to tackle. Subsequently we present our first approach and explain the subtle issues that make it unsuitable for a robust and precise analysis. Finally, we present our revised analyses.

5.1.1 The Status Quo

The principal mechanism for dealing with unknown accesses has already been presented in Section 2.3. However, the must update function completely ignores the accessed blocks and unconditionally ages all blocks by one. Consider again the example in Figure 5.1(c). Assume that the array consists of exactly two memory blocks per cache set and $N > k$. After the loop, the must analysis will have aged all blocks N times, i.e. no previously cached element will survive the loop. In reality, each block can only be aged once, though, namely by the one block of the array that falls into the same cache set. In the remainder of the chapter, we present a new analysis based on *conflict sets* which prevents this over-aging and retains valuable must information across array-accessing loops.

5.1.2 Array-Aware Must

Recall that the must analysis tracks the maximal age of a memory block (domain $\mathcal{B} \to \mathbb{N}_{\leq k}$). Additionally, the new analysis remembers a conflict (multi-)set of arrays that have aged the block in the past. This allows us to recognize them and avoid the double-aging discussed above. The domain is thus extended to $\mathcal{B} \to \mathbb{N}_{\leq k} \times \mathcal{P}_M(\mathcal{A})$. For the sake of clarity and brevity, we denote the elements of the per-block information as $f(b).maxage$ and $f(b).\mathit{conflict}$, respectively.

Before formally defining the analysis we explain it by example. We apply the analysis to Figure 5.1(c). Recall that we assumed the array to occupy two blocks per cache set. This implies that the array can only age each block in the cache twice. This relationship is expressed in the conflict set. We define *bound*(array) := 2. Whenever the array is accessed it is also added to the set. If the set is already *saturated*, we know that the block has already been aged twice and cannot be aged further. If, for example, the block x maps to $(0, \emptyset)$ before the loop, the first iteration ages it to $(1, \{\mathit{array}\}_M)$. the next iteration ages it to $(2, \{\mathit{array}, \mathit{array}\}_M)$. Subsequent iterations, however, do not age x further since *array* is already *saturated* in the conflict set.

We specify our abstraction by a concretization function. The meaning of the *maxage* bound remains the same as in the must analysis. The *conflict* multiset then restricts the concretization. By our considerations above we know that the revised concretization must fulfill two properties

1. If $S(b).conflict = \emptyset$, the concretization is identical to the must analysis
2. If *saturated*$(S(b).conflicts, X)$ holds it must be impossible to age b beyond $S(b).maxage$ by accessing X arbitrarily often. In particular, this means that the analysis can ignore accesses to X.

In addition, we require that any other values of the conflict set provide a smooth transition between the extreme values. This means that the age bound should only grow by one after each access. Bounded multisets suit this requirement; elements can be added one by one until the saturation point is reached, at which the set does not grow any more.

At a first glance, the requirements above suggest that any element A in the conflict set of block x translates to a block of A being younger than x. However, in this concretization joining could only result in the intersection of the two conflict sets. In particular, the analysis allows unlimited aging in the example above: after the first iteration, the state at the end of the loop $(1, \{\mathit{array}\})$ is joined with the state at the beginning of the loop $(0, \emptyset)$. Since the latter contains a state where all elements of *array* are older than x, the join of the two states must be $(1, \emptyset)$; all the progress of the conflict set is erased.

This behaviour is clearly undesirable, in particular because the bound of the joined state has increased compared to the initial state. Effectively, the analysis has paid a cache slot for the array but has not gotten anything in return. To avoid this issue, we allow the analysis to pay in advance for future array accesses. This allows the join partner with lesser *maxage* to achieve a more favorable conflict set just before the join.

We model this prepayment by adding an additional term to the must concretization: an abstract state maps to all concrete states that honor the *maxage* bound *even if all the prepaid accesses are redeemed.*

$$\gamma(S) = \{age \ : \ \underbrace{\forall b \in \mathcal{B}.\, age(b) \leq S(b).maxage}_{\text{Must concretization}} - \underbrace{|S(b).\mathit{conflict} -_M \mathit{arrayof}(younger_{age}(b))|}_{\text{prepaid array accesses not yet redeemed}}\}$$

$$\text{where } younger_{age}(b) := \{b' \in \mathcal{B} \ : \ cacheset(b) = cacheset(b') \wedge age(b') < age(b)\}$$

The update function looks like the must update function if the program accesses a regular block. Accesses to arrays are handled differently. To simplify notation, we define a separate update function for each case. As a convention, we refer to concrete blocks by lowercase letters and to arrays by uppercase letters.

$$update(S, x) = \lambda b. \begin{cases} (0, \emptyset) & x = b \\ S(b) & S(x).age \leq S(b).age \\ (S(b).age + 1, S(b).conflict) & S(x).age > S(b).age \end{cases} \tag{5.7}$$

$$update(S, X) = \lambda b.\ (\min(k, S(b).age + \delta), S(b).conflicts \cup_M \{X\}) \tag{5.8}$$

$$\text{where } \delta \text{ is } 0 \text{ iff } saturated(S(b).conflicts, X) \text{ and } 1 \text{ otherwise}$$

Unfortunately, trying to formulate an abstraction function α (or equivalently, a join function $\sqcup$) gives rise to some ambiguities: Given a set of concrete states, it is not clear whether one should prioritize a large conflict set or a low age bound, or anything in between. Any arbitrary set of concrete states can be represented with any conflict set by simply adjusting the age bound.

As an example, assume a fully associative 4-way cache and a program with two arrays A and B, spanning 1 cache line each. Consider the following set of concrete cache states (where only blocks of interest are shown)

$$\{[A_0 \to 0, x \to 1, \ldots], [B_0 \to 0, x \to 1, \ldots]\}$$

Both, $x \to (2, \{A, B\}_M)$ and $x \to (1, \emptyset)$ are valid abstractions of this set. In the first case, we decided to prioritize the conflict set, while in the second case we decided to prioritize the age bound and simply dropped the conflict sets.

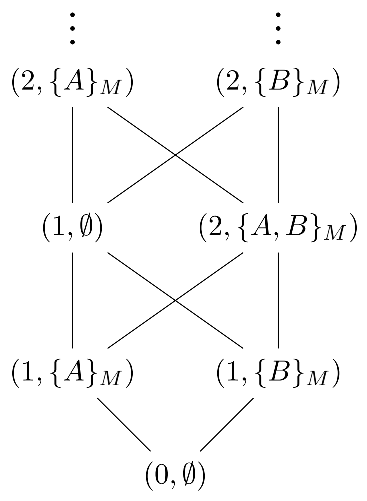


Figure 5.3: The array-aware partial order for two arrays of size 1.

This ambiguity would be harmless if one abstraction was inferior to the other, giving a unique best choice in the matter. However, if both, A and B, are accessed in the future the large conflict set is advantageous, while the increased age bound is a disadvantage in other future access patterns.

This issue is also apparent in the canonical partial order[3] on the set of abstract states depicted in Figure 5.3. The least upper bound of $(1, \{A\}_M)$ and $(1, \{B\}_M)$ is not unique, which means that the abstract domain is not a lattice. While it is technically possible to perform abstract interpretation on non-lattices, one loses the termination and precision guarantees of the framework in the process [22].

As a consequence of the ambiguous join, the analysis has to apply heuristics to choose the best join result. Since this leads to fickle analyses that are hard to reason about, we avoided any complex join heuristics in this thesis. We did, however, consider two constant heuristics, namely:

Conflict set union Prioritize the conflict set.

$$(a_1, c_1) \sqcup (a_2, c_2) = (\max((a_1 + \sum_{d \in c_2 -_M c_1} |d|), (a_2 + \sum_{d \in c_1 -_M c_2} |d|)), c_1 \cup_M c_2) \tag{5.9}$$

$$(a_1, c_1) \sqsubseteq (a_2, c_2) \Leftrightarrow a_1 + |c_2 -_M c_1| \leq a_2 \wedge c_1 \subseteq_M c_2 \tag{5.10}$$

[3] $\forall x, y \in \mathcal{D}. x \sqsubseteq y \Leftrightarrow \gamma(x) \subseteq \gamma(y)$

Conflict set intersection Prioritize the age bound.

$$(a_1, c_1) \sqcup (a_2, c_2) = (\max(a_1, a_2), c_1 \cap_M c_2) \tag{5.11}$$

$$(a_1, c_1) \sqsubseteq (a_2, c_2) \Leftrightarrow a_1 \leq a_2 \wedge c_1 \supseteq_M c_2 \tag{5.12}$$

For both definitions of $\sqcup$ we have explicitly specified the induced order

$$x \sqsubseteq y \Leftrightarrow x \sqcup y = y$$

Note that the simplicity of the solution comes at the price of precision. The union approach maximizes the conflict set at all costs, quickly going to $\top$ if the set becomes too large. On the other hand, the intersection approach leads to the ineffective analysis we tried to avoid with the prepayment mechanic. In particular, it only prevents overaging in array-traversing loops if the conflict set is already saturated at the beginning of the loop[4].

5.2 The Conflict Powerset Approach

There is another, radical solution to the issues with the array-aware must analysis: Instead of figuring out the optimal conflict set for a given situation, we try all of them. If the analysis follows all possible conflict sets it can pick the most advantageous choice whenever we need to classify an access.

Our abstract state is a maximal age *per conflict set*, i.e. changes from $\mathcal{B} \to \mathbb{N}_{\leq k} \times \mathcal{P}_M(\mathcal{A})$ to $\mathcal{B} \times \mathcal{P}(\mathcal{A}) \to \mathbb{N}_{\leq k}$. Note that the multiset has been replaced by a regular set. As we will see later, the powerset-based analysis already provides a smooth transition between the different conflict sets.

We interpret each mapping in this function as a bound on the maximal age, i.e. each mapping (ds, a) provides a bound a that remains valid after accesses to elements in ds. The concretization is therefore the set of states that fulfill all these bounds:

$$\gamma(a) = \{age | \forall b \in \mathcal{B}\, \forall C \in \mathcal{P}(\mathcal{A}) \;:\; age(b) \leq a(b, C) - |older_{age}(b) \cap \bigcup_{ds \in C} blocks(ds)|\}$$

[4]This can often be achieved by virtually unrolling loops sufficiently often.

The update function is similar to the single conflict set analysis:

$$update(a, x) = \lambda b.\lambda c. \begin{cases} \min(k, \sum_{ds \in c} |ds|) & x = b \\ a(b) & a(x, ds) \leq a(b, ds) \\ a(b) + 1 & \text{otherwise} \end{cases} \tag{5.13}$$

$$update(a, X) = \lambda b.\lambda c. \begin{cases} a(b, ds) & X \in c \\ \min(k, a(b, ds) + 1) & X \notin c \end{cases} \tag{5.14}$$

The classification function necessarily differs from the other analyses. The γ-function already prescribes how to bundle all these bounds: the concrete caches fulfill *all* of the constraints, therefore we classify a hit if *any* constraint is lower than k.

$$\textit{must-hit}(b, a) :\Leftrightarrow \exists c.\ a(b, c) < k$$

Similar to the other must analyses, joining consists of taking the point-wise maximum.

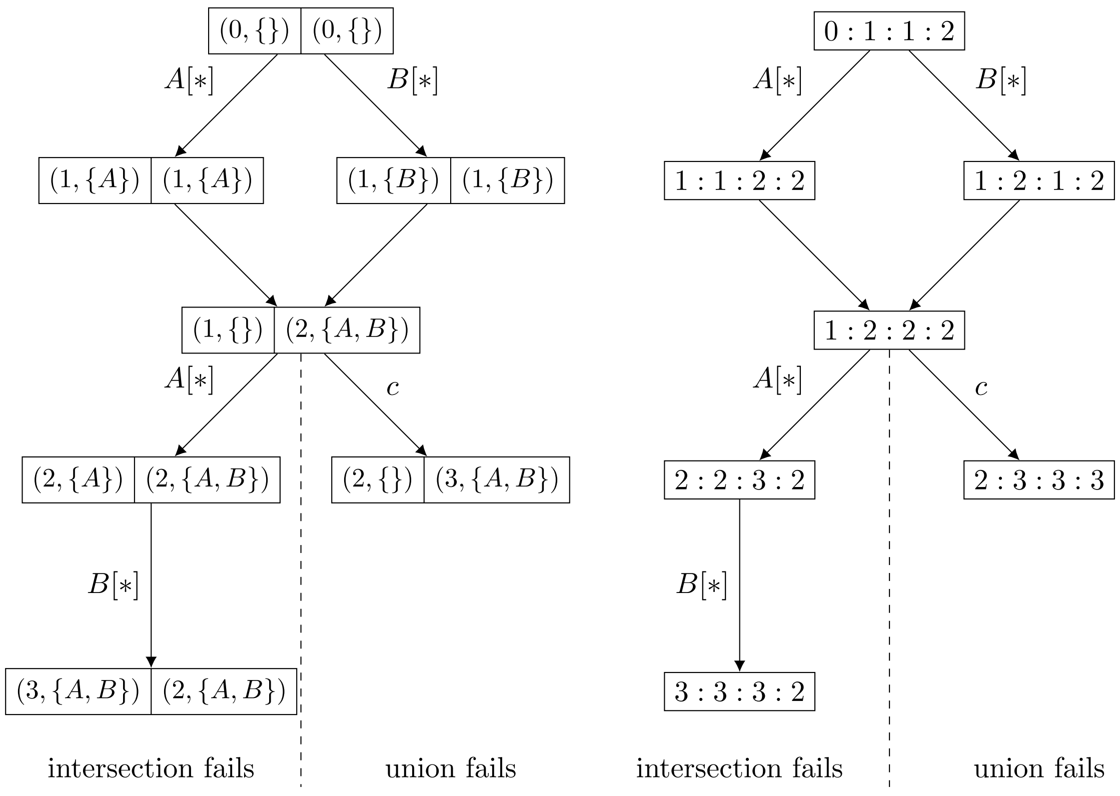
$$x \cup y = \lambda b.\lambda c. \max(x(b, c), y(b, c))$$

Example

The previous domains are no lattices, leading to ambiguous joins. Even though we gave two heuristics, both produce suboptimal results. Furthermore, they are somewhat arbitrary, lacking any formal justification. The powerset approach, on the other hand, is based on a true lattice. The join is uniquely determined and therefore provably optimal for this abstract domain. In particular, the analysis never makes a suboptimal compromise between the conflict set and the age bound. This can be observed in the following example. For simplicity, we assume a 3-way n-set associative cache.

```
/* A and B are of size n */
access(x)
if (...) access(A[i]);
else     access(B[i]);
if (...)  {
        access(A[i]);
        access(B[i]);
} else access(c);
```

The graph of age bounds for block x is depicted in Figure 5.4. The states in diagram (a) show the intersection heuristic on the left and the union heuristic on the right. Both heuristics fail to prove that x remains in the cache. The intersection heuristic



(a) Single conflict set **(b)** 2^n conflict sets

FIGURE 5.4: The microarchitectural state graph in the conflict powerset domain

erroneously ages x three times even though at most 2 blocks can be younger than x. The union heuristic on the other hand overeagerly ages x by two after the initial if, anticipating future array accesses. Accessing a non-array block then evicts x from the must cache. Similar examples can probably be constructed for other heuristics.

Diagram (b) shows the powerset-based analysis for the same program. The different age bounds are shown in the order $\emptyset : \{A\} : \{B\} : \{A, B\}$. The analysis never evicts x, proving its superiority over the previous analyses. Even though most bounds grow to k by the end, the state also contains the one conflict set configuration that yields a better result.

Unfortunately, this improved solution comes at a price: The domain grows exponentially with the number of arrays. However, we believe this is not a major hindrance: First, any conflict set that corresponds to more than $k-1$ blocks is inherently useless, as it will already start at $\top$. As the number of arrays grows, so does the number of inherently useless conflict sets. Second, the analysis can choose to ignore certain arrays: If the number of arrays becomes overwhelming some arrays can be ignored, their accesses turning unknown. We never assumed that $\mathcal{A}$ contains all arrays in the program. A clever analyzer could, for example, select the most promising arrays per program, per function or at even smaller granularity based on some heuristic. As simply forgetting a bound is always safe, it is even possible to change $\mathcal{A}$ during the analysis.

In addition, we believe that most programs will only encounter a small subset of this exponential lattice, meaning that a smart encoding might be able to work around the exponential growth in the common case. However, the encoding would need to allow efficient computation of the join, update and classification function, so a simple entropy coding does not work. Finding such an encoding remains future work.

5.3 Array-aware Persistence

As the must analysis, persistence analysis suffers from precision loss whenever dynamic addresses are used. Thus, we limit the effect of array accesses to the conflict sets of scalar variables. Moreover, we declare entire arrays persistent, which improves the WCET of tight array-sweeping loops that evaded the must analysis.

5.3.1 Handling Accesses to Arrays

Regular persistence analysis ca handle arrays, as the regular must analysis. However, the usual technique (updating the state with each potential target, then joining the result) gives unsatisfying results:

- Conflict set persistence immediately takes all the blocks of the array into the set, quickly exceeding the associativity. It does this even though only one of these accesses can actually happen
- Conditional must persistence, as the must analysis, ages blocks as often as the array is accessed, beyond the actual size of the array.

In this thesis, we decided to focus on conflict set persistence analysis. Adding array awareness to conditional must analysis remains future work.

To avoid the conflict set explosion, the adapted analysis does not remember which specific array element has been accessed. Instead, it counts how often each array has been accessed so far, stopping when the array's size has been reached. We implement this approach with bounded multisets, as in the array-aware must analysis.

$$\begin{tikzcd}
\boxed{x \rightarrow \{x, A[0], A[1], A[2]\}} \arrow[d, "\text{x}"] & & \boxed{x \rightarrow \{x, A\}} \arrow[d, "\text{x}"] \\
\boxed{x \rightarrow \{x\}} \arrow[d, "\text{A[*]}"] & & \boxed{x \rightarrow \{x\}} \arrow[d, "\text{A[*]}"] \\
\boxed{x \rightarrow \{x, A[0], A[1], A[2]\}} \arrow[uu, bend left=60] & & \boxed{x \rightarrow \{x, A\}} \arrow[uu, bend left=60]
\end{tikzcd}$$

FIGURE 5.5: Array accesses in elementwise conflict set analysis

At first, this approach seems unnecessarily complicated; adding the array n times to the conflict set instead of adding the n constituting blocks of the array looks like it ends up with the same result. However, the key difference is *when* these elements are added to the set. The individual blocks of the array all have to be inserted on the first unknown access into that array; adding the entire array can be done one at a time. While this usually does not make a difference in setwise conflict counting, it can be critical in elementwise conflict counting (Figure 5.5). Using special handling for unknown array accesses, we can exploit that between any two accesses to x only one access to A is possible.

5.3.2 Declaring Arrays Persistent

Conveniently, the classical setwise persistence analysis can also be used to declare entire arrays persistent. An array is persistent if the size of the global conflict set in the scope is $\leq k$ in all occupied cache sets. In this context, being persistent means that for each time the program enters the scope it can only incur $|A|$ misses for the array A.

It seems tempting to extend elementwise conflict counting to arrays. However, we can never rejuvenate the abstract cache after an access to an array, because we never know that all its blocks have been accessed. Consequently, the conflict set can never be reset and the element-wise conflict set becomes completely equivalent to the global conflict set.

Another improvement that quickly comes to mind is partial persistence, i.e. an array that is persistent in some cache sets but not in all. This allows the analysis to restrict the number of misses in the persistent cache sets, possibly improving the state-based cache analyses. However, this level of interaction between the path analysis and the microarchitectural analysis is hard to achieve. We would need to split up the state graph and select one of the paths later. This is prohibitively expensive. Finding a less expensive solution to partial persistence remains future work.

5.4 Evaluation

In this section, we first evaluate the different array-aware must analyses. Our measurements indicate that the array-aware must analysis does not affect the WCET bound in most cases. In the cases where it does, the intersection heuristic performs worse than the other two. The union heuristic performs exactly as well as the powerset-based analysis, although the sample set is too small to draw any definitive conclusions.

Subsequently, we evaluate our array-aware persistence analysis. We show that identifying arrays as persistent greatly reduces the WCET bound in most cases. It also subsumes most effects of the array-aware must analysis. It is worth noting, though, that many testcases are implementations of basic algorithms. In real programs these algorithms would be subroutines and the overall improvement probably smaller.

The evaluation also shows that array-aware cache analysis has negligible effects on analysis runtime and memory consumption.

Finally we evaluate the combination of the two contributions of this thesis, the write-back analysis and the array analysis. We show that array awareness has a greater impact on write-back systems than on write-through systems. We also show, that, compared to array-aware write-through, array-aware write-back analysis yields better result on all but two of our testcases

Preliminaries

The modelled machine has a line size of 16 bytes, 32 cache sets and either 2 or 8 ways. In all benchmarks we measure the number of cache misses, as this property only depends on the cache analysis. The cache follows the write-back policy, i.e. the cache does not differentiate between loads and stores.

Before we begin the evaluation, we first need to consider the address precision distribution in our benchmarks; an array-aware analysis obviously does not change anything if there are no array accesses in the program. The distribution for our testcases is depicted in Figure 5.6. Only the testcases containing array accesses are considered in this chapter.

So far, we never specified how we recognize array accesses. Since *llvmta* is integrated into the compiler, we still have access to high-level information. For example, we can read the annotations in LLVM's back-end representation. These annotations contain LLVM's knowledge about the target of the memory access, in particular the surrounding array.

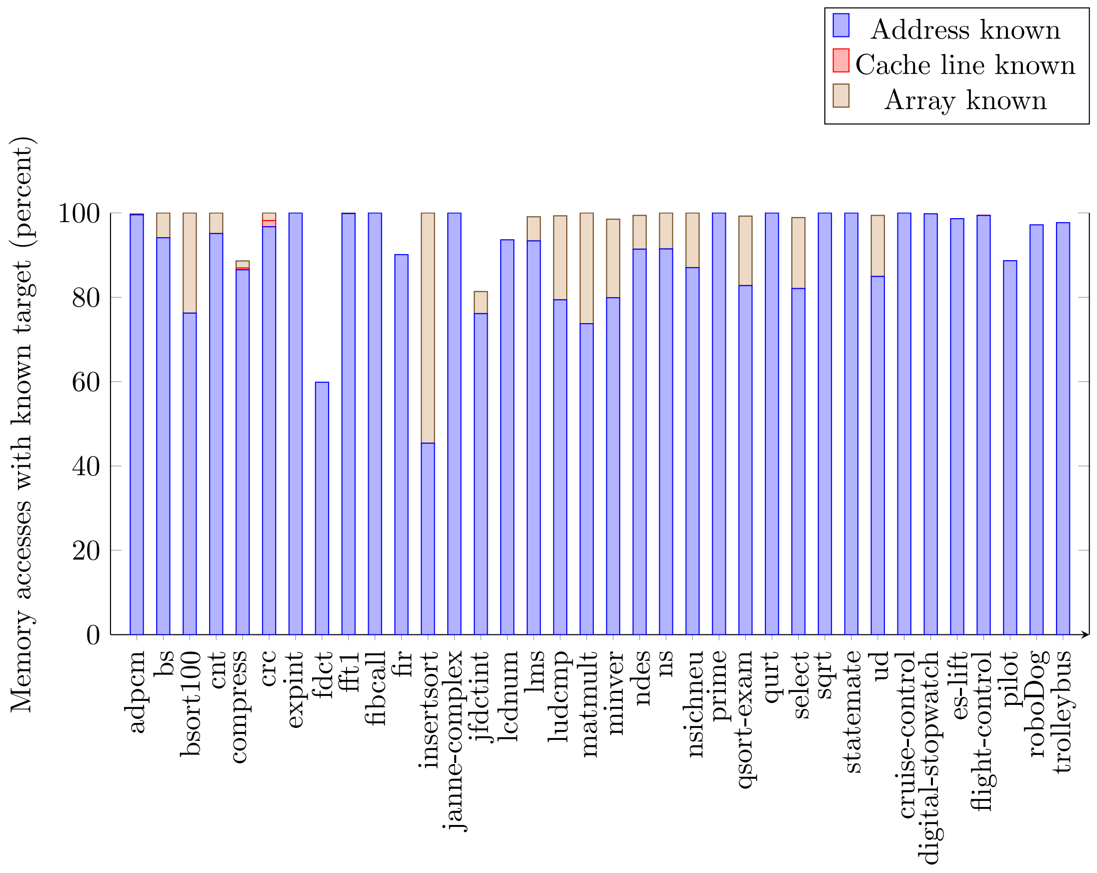


FIGURE 5.6: Diagram of the address precision distribution

5.4.1 Array-aware Must Analysis

In Figure 5.7 we compare our different array-aware must analyses. The diagram clearly shows that the array-aware must analysis has no effect on the bound in all but a few cases. The main reason is that it cannot prove cache hits for arrays; it can only prove that already cached elements cannot be evicted by arrays. The two cases where the must analysis achieves a bound reduction of more than 10% are *cnt* and *insertsort*.

A simplified version of the *cnt* core loop is depicted in Figure 5.8. Traditional must analysis assumes that the condition can age *pos* arbitrarily often. It concludes that *pos* can be evicted if the condition evaluates to *false* repeatedly. Our array-aware must analysis, though, recognizes that *array* has only one cache line in *pos*'s cache set and therefore cannot age it more than once. It thereby proves that all accesses to *pos* are hits.

The *insertsort* testcase initializes all the array elements individually before the loop. The array is therefore guaranteed to be cached and can therefore not be evicted by a loop that only accesses this array.

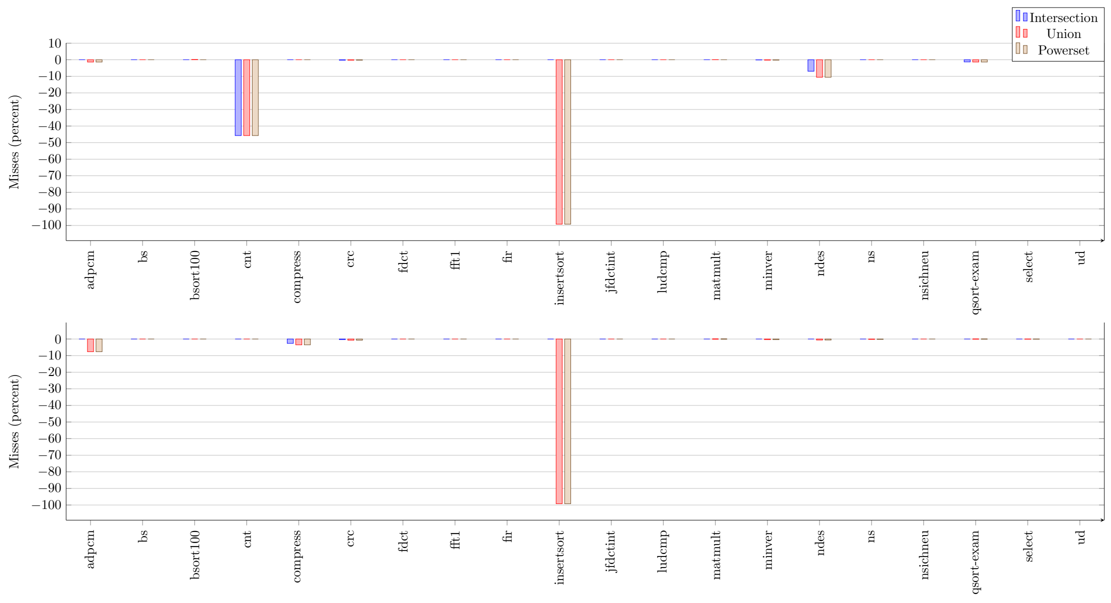


FIGURE 5.7: Comparison of array-aware must analyses. All bars are normalized to the number of cache misses under traditional cache analysis (i.e. traditional must and May analysis together with elementwise counting persistence)

```
/* array spans 1 cache line in each set,
   i, pos and neg are in different cache sets */
pos = 0;
for (i=0; i<N; i++)
        if (array[i] > 0) pos++;
```

FIGURE 5.8: The core loop of the *cnt* benchmark (simplified)

The analysis also shows that the intersection heuristic is inferior to the other two analyses. The union heuristic, however, performs equally well as the more complex and more expensive powerset-based analysis. This does not suffice to dismiss the powerset-based analysis as superfluous, though, as the sample size is way too small. These details are of little concern anyway unless the analysis can be made more effective.

5.4.2 Array-aware Persistence Analysis

Figure 5.9 presents the results of our array persistence analysis. We can see that it outperforms the array-aware must analysis in every single benchmark except *insertsort*. As predicted, its ability to prove entire arrays persistent tremendously reduces the number of misses. For the sorting benchmarks, for example, the number of misses shrinks from one miss per loop iteration to one miss per cache line in the array.

It might be surprising at first that *bs* (which performs binary search on an array) does not profit from array-aware cache analysis. The reason is that binary search avoids loading the entire array into the cache. The whole point of binary search is to skip large

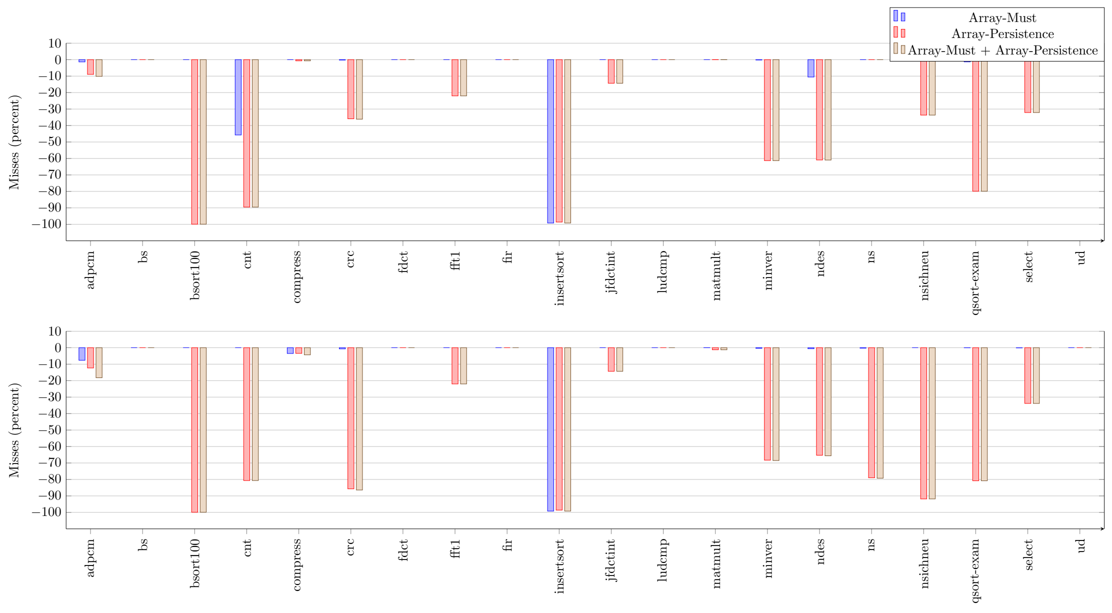


FIGURE 5.9: Relative WCET increase under array-aware persistence analysis. The bars are normalized to the array-unaware analysis. For comparison, we also show the result of the conflict powerset array-aware must analysis. Third, we present the effect of combining both analyses.
Top: associativity 2, Bottom: associativity 8

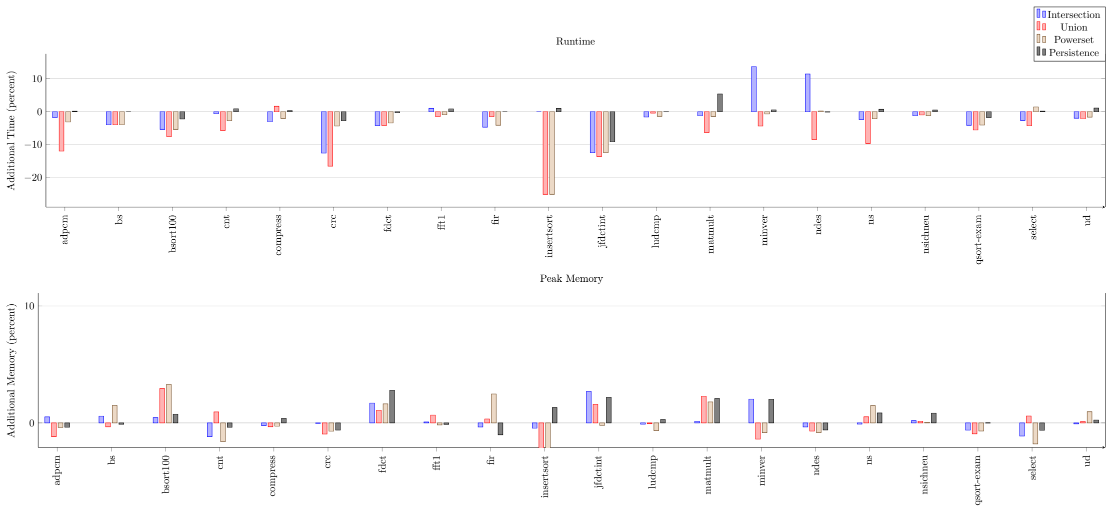


FIGURE 5.10: Effects of the array-aware analyses on analysis runtime and memory consumption. The bars are normalized to the array-oblivious analysis

parts of the array. Bounding the misses in a $\mathcal{O}(\log n)$ loop by a value in the order of $\mathcal{O}(n)$ obviously does not improve the WCET bound.

5.4.3 Analysis Cost

Figure 5.10 shows the impact of array-aware cache analyses on the runtime and the memory consumption of *llvmta* (at associativity 8). We observe that the exponential

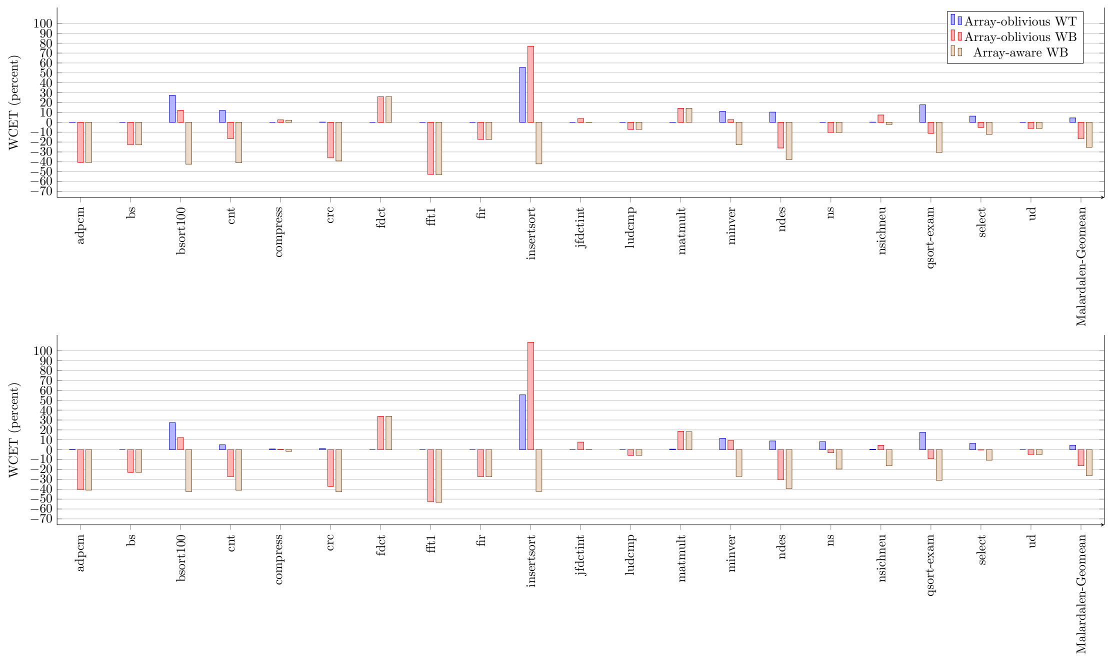


Figure 5.11: Combining write-back and array-aware analysis. All bars are normalized to the WCET bound of an array-aware write-through analysis.
Top: associativity 2, Bottom: associativity 8

nature of the conflict-powerset domain has little effect on runtime or memory consumption. On the contrary, the analysis becomes even *faster* in most cases. The reduction of the state graph size apparently makes up for the additional analysis effort.

Overall, the change in analysis runtime is small. Ignoring the intersection analysis (which is inferior anyway), the worst observed runtime increase is at about 5%. The additional memory consumption is negligible, never exceeding 3%.

5.4.4 Array-aware Write-back analysis

Finally, Figure 5.11 shows the combination of our two contributions. We can see that, depending on the cache size, write-back caches perform 25% better on average than equivalent write-through caches. Array-awareness apparently has greater impact on write-back systems. This phenomenon is best explained by example. Consider Figure 5.12, which contains the inner loop of the *insertsort* benchmark. This loop performs four loads from a and two stores to a. Under write-through, the four loads are covered by the persistence constraint while the two stores access memory anyway. Under write-back, though, all six accesses are covered by the persistence constraint, yielding two memory accesses less *per loop iteration* as soon as the array is loaded into the cache.

```
while (a[j] < a[j-1])
{
        temp = a[j];
        a[j] = a[j-1];
        a[j-1] = temp;
        j--;
}
```

FIGURE 5.12: The inner loop of the *insertsort* benchmark

There still are two testcases that do not perform better under write-back: *fdct* and *matmult*. The former cannot be analyzed properly since it explicitly uses pointers to walk through the array. Our analysis is not sophisticated enough to detect this. However, this problem can be easily handled in practice by programmer annotations. Unlike loop bounds, identifying the array a pointer points into is trivial for the programmer. The *matmult* testcase, however, hints at a fundamental weakness of our analysis. The cache behaviour of matrix multiplication is highly dependent on the iteration order. Since our analysis is oblivious to this order, it cannot prove anything unless all three arrays fit into the cache.

Chapter 6

Conclusion and Future Work

In this thesis we developed a new cache analysis for write-back caches. We identified dirtifying stores as the primary means to bound the number of write backs in a program. Evaluation showed, that write-back caches together with our analysis often achieve lower WCET bounds than write-through systems. However, unknown accesses impact the write-back analysis more than the write-through analysis. Write-through systems therefore achieve better WCET bounds on programs containing many unknown accesses. Motivated by this shortcoming, we developed array-aware must and persistence analyses. While the array-aware must analysis mostly has negligible impact, array-aware persistence significantly reduces the WCET bound. Our final evaluation shows that this addresses the shortcomings of the write-back analysis.

Future Work

Exploiting persistence in write-back analysis: Another potential improvement to the dirtifying store analysis is to take persistence information into account. If a block is persistent, we can also infer that there can only be one dirtifying store to this block. Likewise, there can only be $|A|$ dirtifying stores to a persistent array.

Partial array persistence: In this thesis we did not consider partially persistent arrays, i.e. arrays that are persistent in some but not all cache sets. The main reason is that we are unable to identify the cache set of an unknown access. For unknown accesses depending on the loop iteration variable, we can do better. Using an analysis similar to [20], it might be possible to formulate persistence constraints for a subset of the loop iterations; in our case, for loop iterations that only access cache sets where the array is persistent.

Overlapping arrays: We assumed that no cache line belongs to two arrays at once. Arrays are not always aligned to cache lines, though, and a cache line might contain the end of one and the beginning of another array. In this case, one knows that the two arrays together occupy one cache line less than their sizes indicate. Formalizing and implementing this behaviour remains future work.

Speculative execution: Our array-aware analysis assumes that array accesses are always restricted to the interval $[\mathit{base}, \mathit{base} + \mathit{size}]$. If the underlying processor contains an out-of-order pipeline, it might speculatively load data *after* the array. For those processors, it might be possible to assume a larger *size* that covers all data loaded as the result of speculative execution as well.

Store buffers: Besides write-back caches, *store buffers* are another common processor component that may delay stores. Store buffers are small waiting queues for store requests. When the processor performs a store it hands the request to the buffer and continues executing without delay. As long as there is space in the buffer, stores appear to complete within a single cycle. The *unblocked stores* mechanism we used in the evaluation can be seen as a store buffer of size 1. However, real store buffers are often more sophisticated. They combine multiple stores into one if possible. They forward pending stores to the processor if it loads the same address. They might even reorder or delay pending stores on their own. It would be interesting whether a suitable abstract domain can be found and, if so, how it performs compared to the write-back analysis.

Relational write-back analysis: Our write-back analysis, as the must and may analyses it is based on, operate on memory addresses. Relational analyses, on the other hand, are based on symbolic names. This allows them to prove that two accesses target the same block without knowing the address of the block (Hahn et al., 2012 [23]). This same-block relation is also useful for write-back caches: if we know two subsequent stores target the same unknown block, we can infer that the second store is not dirtifying.

Preemptive multitasking: This thesis focused on WCET estimation of single programs. For multi-tasked systems this is insufficient. Even if a task provably meets its deadline in isolation, interference by other tasks might further delay it. For example, other tasks can preempt the task and place dirty data in the cache. The original task then has to write back data that was not anticipated by the WCET analysis. Therefore, WCET estimation has to be embedded into a larger *response time analysis* that takes this interference into account. Davis et al. [24] developed such a response time analysis for write-back caches. However, they lacked a suitable WCET analysis and therefore only evaluated their results using simulated execution traces. Combining the two approaches allows a more rigorous evaluation.

Bibliography

[1] Honglu Zhang, Deren Ma, and Srini V. Raman. CAE-based side curtain airbag design. SAE technical paper, Delphi Corporation, 2004.

[2] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 5th edition, 2012.

[3] *AGC4 Memo #9 - Block II instructions*. MIT Instrumentation Laboratory, July 1966.

[4] *IBM System/360 Model 85 – Functional Characteristics*, 2nd edition, June 1968.

[5] Andreas E Dalsgaard, Mads Chr Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. Metamoc: Modular execution time analysis using model checking. In *OASIcs-OpenAccess Series in Informatics*, volume 15. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2010.

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[7] Wayne Wolf. *Computers as components: principles of embedded computing system design*. Academic Press, 2001.

[8] Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems*, 3(1):05–1–05:48, 2016.

[9] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In *International Static Analysis Symposium*, pages 52–66. Springer, 1996.

[10] Patrick Cousot. Abstract interpretation based formal methods and future challenges. In *Informatics: 10 Years Back, 10 Years Ahead*, pages 138–156. Springer, 2001.

[11] Christoph Cullmann. Cache persistence analysis: a novel approachtheory and practice. *ACM SIGPLAN Notices*, 46(5):121–130, 2011.

[12] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.

[13] Claus Michael Faymonville. Evaluating compositional timing analyses. Master's thesis, Saarland University, 2015.

[14] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):26, 2007.

[15] Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[16] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. In Björn Lisper, editor, *WCET2010*, pages 137–147, Brussels, Belgium, jul 2010. OCG.

[17] SCADE suite. http://www.esterel-technologies.com/products/scade-suite/.

[18] Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Towards formally verified optimizing compilation in flight control software. In *PPES 2011: Predictability and Performance in Embedded Systems*, volume 18, pages 59–68. Schloss Dagstuhl, Leibniz-Zentrum fuer Informatik, 2011.

[19] Simon Wegener. Computing same block relations for relational cache analysis. In *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23 of *OpenAccess Series in Informatics (OASIcs)*, pages 25–37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012.

[20] Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-aware data cache analysis for wcet estimation. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

[21] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 11th international conference on Supercomputing*, pages 317–324. ACM, 1997.

[22] Graeme Gange, Jorge A Navas, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. Abstract interpretation over non-lattice abstract domains. In *International Static Analysis Symposium*, pages 6–24. Springer, 2013.

[23] Sebastian Hahn and Daniel Grund. Relational cache analysis for static timing analysis. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, pages 102–111, 2012.

[24] Robert I. Davis, Sebastian Altmeyer, and Jan Reineke. Analysis of write-back caches under fixed-priority preemptive and non-preemptive scheduling. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, pages 309–318, 2016.