Caches in WCET Analysis
Predictability, Competitiveness, Sensitivity

Dissertation

Zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

von
Jan Reineke

Saarbrücken
2008

Tag des Kolloquiums:	07.11.2008
Dekan:	Prof. Dr. Joachim Weickert
Prüfungsausschuss:	
Vorsitzender:	Prof. Dr. Raimund Seidel
Gutachter:	Prof. Dr. Dr. h.c. Reinhard Wilhelm Prof. Dr. Eljas Soisalon-Soininen Prof. Dr. Lothar Thiele
Akademischer Mitarbeiter:	Dr. Philipp Lucas

Impressum

Herstellung und Verlag: epubli GmbH, Berlin, www.epubli.de
Printed in Germany
ISBN: 978-3-941071-69-8

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Abstract

Embedded systems as they occur in application domains such as automotive, aeronautics, and industrial automation often have to satisfy hard real-time constraints. Safe and precise bounds on the worst-case execution time (WCET) of each task have to be derived. This thesis studies the influence of cache replacement policies on the precision and soundness of WCET analyses.

We define and evaluate *predictability metrics* that capture how quickly *may* and *must* information can be obtained under a particular replacement policy. The metrics mark a limit on the precision of *any* cache analysis.

We generalize the notion of competitiveness to that of *relative competitiveness*. Relative competitive ratios bound the performance of a policy relative to that of another policy. Constructing a quotient transition system enables us to automatically compute such competitive ratios. Competitive ratios of LRU relative to FIFO and MRU yield the first *may* cache analyses for the two policies. These analyses are optimal with respect to the predictability metrics.

Measurement has been proposed as an alternative to static analysis in WCET analysis. To evaluate the soundness of measurement-based WCET analysis, we investigate how *sensitive* replacement policies are to the state they are starting in. Analysis reveals that for FIFO, MRU, and PLRU, measurement may yield WCET estimates that are dramatically wrong.

Zusammenfassung

Eingebettete Systeme in der Luft- und Raumfahrt, der Fahrzeugtechnik und der industriellen Automation müssen oft harten Echtzeitanforderungen genügen. Sichere und enge Schranken für die maximale Laufzeit (WCET) eines Programms müssen hergeleitet werden. Diese Arbeit untersucht den Einfluss der Cache-Ersetzungsstrategie auf die Präzision und Zuverlässigkeit von WCET-Analysen.

Wir definieren und evaluieren *Vorhersagbarkeitsmetriken*, die erfassen wie schnell *May*- und *Must*-Information für eine Ersetzungsstrategie erlangt werden kann. Diese Metriken stellen Schranken für die Präzision *beliebiger* Cache-Analysen dar.

Wir verallgemeinern den Begriff der *Competitiveness* zu dem der *relativen Competitiveness*. *Relative-Competitive-Verhältnisse* begrenzen die Leistung einer Ersetzungsstrategie im Verhältnis zur Leistung einer anderen. Durch die Konstruktion von Quotienten-Transitions-Systemen können solche Verhältnisse automatisch berechnet werden. Die Verhältnisse von LRU relativ zu FIFO und MRU begründen die ersten *May*-Cache-Analysen für diese Strategien. Diese Analysen sind optimal in Bezug auf die Vorhersagbarkeitsmetriken.

Messung wurde als Alternative zu statischen WCET-Analysen vorgeschlagen. Um die Zuverlässigkeit messbasierter WCET-Analysen auszuwerten, analysieren wir wie stark die Leistung einer Ersetzungsstrategie von ihrem Anfangszustand abhängt. Die Analyse ergibt, dass Messung zu extrem falschen WCET-Schätzungen führen kann, falls FIFO, MRU oder PLRU verwendet wird.

Acknowledgements

First of all, I have to express my gratitude to my advisor Prof. Reinhard Wilhelm for giving me the opportunity to work in his group. He gave me the freedom and the support to pursue research in a variety of interesting areas but also the guidance to do relevant work. I learned a lot from him.

At the chair for programming languages and compiler construction I have had the chance to get to know and work with many great people: Much of the work presented in this thesis has been done in close collaboration with Daniel Grund. Thank you, I hope to continue our enjoyable cooperation! Christoph Berg inspired much of the early work on this thesis.

The following people deserve thanks for various subsets of {valuable discussions, instant hardware support, collaboration, administrative advice, entertainment, proofreading, basketball, football, defense preparations}: Mohamed Abdel Maksoud, Sebastian Altmeyer, Peter Backes, Jörg Bauer, Claire Burguière, Nico Fritz, Gernot Gebhard, Sebastian Hack, Jörg Herter, Philipp Lucas, Oleg Parshin, Markus Pister, Marc Schlickling, Rathijit Sen, Lili Tan, Björn Wachter. Thanks for planning and sometimes canceling trips to Ilina Bach, Rosy Faßbender, and Stefanie Haupert-Betz.

I would also like to thank Prof. Eljas Soisalon-Soininen and Prof. Lothar Thiele for serving as referees on my thesis committee.

Last but not least, I want to thank my parents for their love and support.

This work has been supported by the German Research Foundation (DFG) as part of the transregional research center SFB/TR 14 AVACS, by the German-Israeli Foundation for Scientific Research and Development (GIF), and by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement number 216008 (Predator). It has benefited from discussions within the European Networks of Excellence ARTIST2 and ARTIST DESIGN.

Contents

1 Introduction

Will my airbags save my life if I crash my car? The answer to this question critically depends on the airbag controller correctly and timely detecting the crash and firing the right airbags. The airbag controller is an example of an embedded system that has to satisfy hard real-time constraints. It is a computer system interacting with the physical environment: it reads sensor values and computes a response that will in turn affect its environment. Besides functional correctness, timeliness of reactions is absolutely necessary. Failure to react within a few milliseconds may be fatal. A schedulability analysis needs to prove that all of the tasks involved in critical decisions will meet their respective deadlines. As an input to such a schedulability analysis, safe and precise bounds on the worst-case execution time (WCET) of each task have to be derived. This thesis studies the influence of the cache on the precision, soundness, and construction of WCET analyses.

Execution times of a task vary depending on the task's inputs and the initial hardware state. While the functional behavior is usually input-deterministic, the timing behavior of a task under a particular input has become very dependent on the state of the hardware. Caches, deep pipelines, and all kinds of speculation are used in today's embedded systems to improve average-case performance. Such components increase the variability of execution times of instructions due to the possibility of timing accidents with high penalties: a cache miss may take 100 times longer than a cache hit. Therefore, the timing behavior strongly depends on the state of these components.

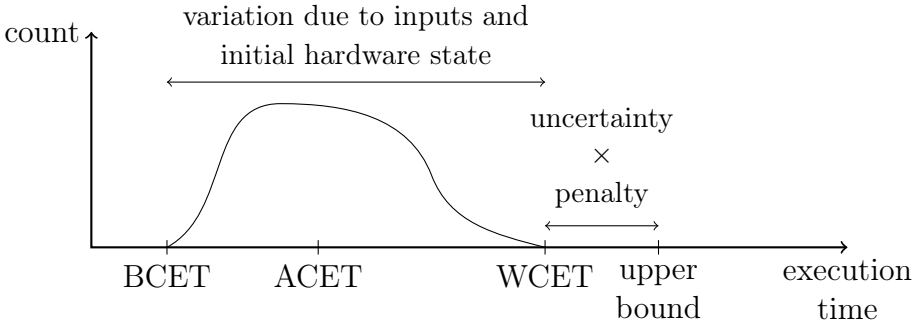


Figure 1.1: A distribution of execution times. The border cases are known as Best- and Worst-Case Execution Time (BCET and WCET). Uncertainty comprises timing accidents that cannot be excluded statically but never happen during execution.

WCET analyses need to compute an upper bound on the execution time for all possible initial hardware states and inputs. The precision of a WCET analysis greatly depends on its ability to statically exclude as much detrimental behavior to the timing of the program's instructions as possible: cache misses, mispredicted branches, pipeline stalls, etc. Exclusion of these so-called timing accidents tightens the upper bound by the associated timing penalty, e.g., the cache miss penalty or the time to refill the pipeline. Each timing accident that cannot be excluded statically but never happens during execution, degrades the precision of the computed upper bound on the WCET as illustrated in Figure 1.1.

Due to high miss penalties, caches have a particularly strong influence on both the variation of execution times due to the initial hardware state and on the precision of static WCET analyses. A cache's behavior is controlled by its replacement policy. We investigate the influence of the cache replacement policy on

- the amount of *inherent uncertainty in static cache analysis*, i.e., cache misses that cannot be excluded statically but never happen during execution,
- the *maximal variation in cache performance* due to the initial cache state, and
- the *construction of static cache analyses*, analyses that statically classify memory references as cache hits or misses.

The following sections explain the three problems in more detail and sketch our approaches and contributions.

Predictability Metrics – Limits on the Precision of Static Cache Analyses

In static analysis there is a concept of *may* and *must* information. In alias analysis, for instance, there are *may* and *must* aliases. Analogously, there is a concept of *may* and *must* cache information in static cache analysis: *may* and *must* caches are upper and lower approximations, respectively, to the contents of all concrete caches that will occur whenever program execution reaches a program point. The *must* cache at a program point is a set of memory blocks that are definitely in each concrete cache at that point. The *may* cache is a set of memory blocks that may be in a concrete cache whenever program execution reaches that program point. Corresponding to *may* and *must* cache information, there are *may* and *must* cache analyses.

Must cache information is used to derive safe information about cache hits; in other words it is used to exclude the timing accident "cache miss". The complement of the *may* cache information is used to safely predict cache misses. While some cache analyses explicitly maintain *may* and *must* cache information, others do so only implicitly. Whenever a cache analysis predicts a cache miss this prediction is based on *may* cache information, be it explicit or implicit. Similarly, a cache hit prediction is always based on *must* cache information.

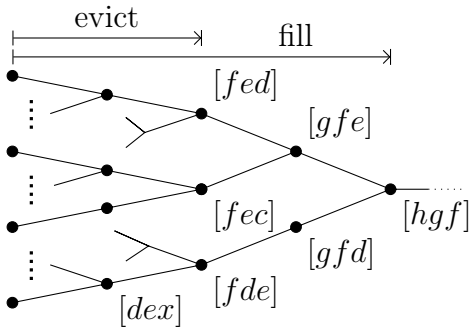


Figure 1.2: Initially different cache sets converge when accessing a sequence $\langle a, b, c, d, e, f, g, h, \ldots \rangle$ of pairwise different memory blocks. Selected cache sets are annotated with their contents.

Usually there is some uncertainty about the cache contents, i.e., the *may* and *must* caches do not coincide; there are memory blocks which can neither be guaranteed to be in the cache nor not to be in it. The greater the uncertainty in the *must* cache, the worse the upper bound on the worst-case execution time, as illustrated in Figure 1.1. Similarly, greater uncertainty in the *may* cache entails a less precise lower bound on the best-case execution time.

There are several reasons for uncertainty about cache contents:

- Static cache analyses usually cannot make any assumptions about the initial cache contents. Cache contents on entrance depend on previously executed tasks. Even assuming a completely empty cache may not be conservative [Berg, 2006].
- At control-flow joins, analysis information about different paths needs to be safely combined. Intuitively, one must take the intersection of the incoming *must* information and the union of the incoming *may* information. A memory block can only be in the *must* cache if it is in the *must* caches of all predecessor control-flow nodes, correspondingly for *may* caches.
- In data-cache analysis, the *address analysis* may not be able to exactly determine the address of a memory reference. Then the *cache analysis* must conservatively account for all possible addresses.
- Preempting tasks may change the cache state in an unpredictable way at preemption points [Gebhard and Altmeyer, 2007].

Since information about the cache state may thus be unknown or lost, it is important to recover information quickly to be able to classify memory references safely as cache hits or misses. This is possible for most caches. However, the *speed* of this recovery greatly depends on the cache replacement policy. It influences how much uncertainty about cache hits and misses remains. Thus, the *speed of recovery* is an indicator of timing predictability.

We introduce two metrics, *evict* and $fill$, that indicate how quickly knowledge about cache hits and misses can be (re-)obtained under a particular replacement policy. They

mark a limit on the precision that *any* cache analysis can achieve, be it by abstract interpretation or any other sound method. Figure 1.2 illustrates the two metrics. *evict* tells us at which point we can safely predict that some memory blocks are no more in the cache, i.e., they are in the complement of may information. Any memory block not contained in the last *evict* accesses cannot be in the cache set. The greater *evict*, the longer it takes to gain may information. $fill$ accesses are required to converge to one completely determined cache set. At this point, complete may and must information is obtained, which allows to precisely classify each memory access as a hit or a miss. The two metrics mark a limit on *any* cache analysis: no analysis can infer any may information (complete must information) given an unknown cache-set state and less than *evict* ($fill$) memory accesses.

A thorough analysis of the LRU, FIFO, MRU, and PLRU policies yields the respective values under these metrics. Under the two metrics, LRU is optimal, i.e., *may*- and *must*-information can be obtained in the least possible number of memory accesses. PLRU, MRU, and FIFO, perform considerably worse. Compared to an 8-way LRU, it takes more than twice as many accesses to regain complete *must*-information for equally-sized PLRU, MRU, and FIFO caches. As a consequence, it is *impossible* to construct cache analyses for PLRU, MRU, and FIFO that are as precise as known LRU analyses. Further analyses elaborate on these results and yield a more refined view on the limits of cache analyses: While *evict* and $fill$ constitute milestones in the recovery of information, supplementary results show how information evolves in between.

Relative Competitiveness of Replacement Policies

Developing cache analyses – analyses that statically determine whether a memory access associated with an instruction will always be a hit or a miss – is a difficult problem. Precise and efficient analyses have been developed for set-associative caches that employ the least-recently-used (LRU) replacement policy [Ferdinand et al., 1997, Ferdinand and Wilhelm, 1999, White et al., 1997, Ghosh et al., 1998, Chatterjee et al., 2001]. Other commonly used policies, like first-in-first-out (FIFO) or Pseudo-LRU (PLRU) are more difficult to analyze [Reineke et al., 2007]. We are not aware of any published analysis that may safely predict cache misses in the presence of FIFO, MRU, or PLRU replacement.

Relative competitive analyses yield upper (lower) bounds on the number of misses (hits) of a policy P relative to the number of misses (hits) of another policy Q. For example, a competitive analysis may find out that policy P will incur at most 30% more misses than policy Q and at most 20% less hits in the execution of any task.

We propose the following approach to determine safe bounds on the number of cache hits and misses by a task T under FIFO(k), PLRU(l)[1], or any another replacement policy:

1. Determine competitiveness of the desired policy P relative to a policy Q for which a cache analysis exists, like LRU.

[1] k and l denote the respective associativities of FIFO(k) and PLRU(l).

2. Perform cache analysis of task T for policy Q to obtain a cache-performance prediction, i.e., upper (lower) bounds on the number of misses (hits) by Q.
3. Calculate upper (lower) bounds on the number of misses (hits) for P using the cache analysis results for Q and the competitiveness results of P relative to Q.

Step 1 has to be performed only once for each pair of replacement policies.

A limitation of this approach is that it only produces upper (lower) bounds on the number of misses (hits) for the whole program execution. It does not reveal at which program points the misses (hits) will happen, something many timing analyses need. We will demonstrate that relative competitiveness results can also be used to obtain sound *may* and *must* cache analyses [Ferdinand and Wilhelm, 1999], i.e., analyses that can classify individual accesses as hits or misses.

We present a tool to automatically compute relative competitiveness results for a large class of replacement policies, including LRU, FIFO, MRU, PLRU, and OPT. We generalize some of the automatically computed results, which hold for fixed associativities, to arbitrary associativities. This is aided by the ability of our tool to generate example memory access sequences that exhibit the worst-case relative behavior. One of our results is that for any associativity k and any workload, FIFO(k) generates at least half the number of hits that LRU(k) generates. Another result is that *may* cache analyses for LRU can be safely used as *may* cache analyses for MRU and FIFO.

Sensitivity of Replacement Policies – On the Correctness of Measurement-based Timing Analysis

Different methods have been proposed for timing analysis [Wilhelm et al., 2008]; measurement[2] [Petters, 2002,Bernat et al., 2002,Wenzel, 2006] and static analysis [Ferdinand et al., 2001, Theiling et al., 2000] being the most prominent. Both methods compute estimates of the worst-case execution times for program fragments like basic blocks. If these estimates are correct, i.e., they are upper bounds on the worst-case execution time of the program fragment, they can be combined to obtain an upper bound on the worst-case execution time of the task.

While using similar methods in the combination of execution times of program fragments, the two methods take fundamentally different approaches to compute these estimates:

- Static analyses based on abstract models of the underlying hardware compute invariants about the set of all execution states at each program point under *all* possible initial states and inputs and derive upper bounds on the execution time of program fragments based on these invariants.
- Measurement executes each program fragment with a subset of the possible initial states and inputs. The maximum of the measured execution times is in general an underestimation of the worst-case execution time.

[2]Measurement-based timing analysis as discussed here is also referred to as hybrid measurement-based timing analysis as opposed to end-to-end measurement-based analysis.

If the abstract hardware models are correct, static analysis computes safe upper bounds on the WCETs of program fragments and thus also of tasks. However, creating abstract hardware models is an error-prone and laborious process, especially if no precise specification of the hardware is available.

The advantage of measurement over static analysis is that it is more easily portable to new architectures, as it does not rely on such abstract models of the architecture. In addition it may compute more precise estimates of the WCET. On the other hand, soundness of measurement-based approaches is often hard to guarantee. Measurement would trivially be sound if all initial states and inputs would be covered. Due to their huge number this is usually not feasible. Instead, only a subset of the initial states and inputs can be considered in the measurements. We study whether measurement-based timing analysis can be safely performed in the presence of unlocked caches. To this end, we introduce the notion of sensitivity of a cache replacement policy.

Sensitivity of a cache replacement policy expresses to what extent the initial state of the cache may influence the number of cache hits and misses during program execution. We first describe how to adapt the method to determine relative competitiveness relations to automatically compute sensitivity properties. However, our main contributions besides the introduction of sensitivity are the application of the analysis to relevant policies and the interpretation of the analysis results with respect to measurement-based timing analysis: Analysis results demonstrate that the initial state of the cache can have a strong impact on the number of cache hits and misses during program execution if FIFO, MRU, or PLRU replacement is used. A simple model of execution time is used to evaluate the impact of cache sensitivity on measured execution times. The model shows that underestimating the number of misses as strongly as is possible for FIFO, MRU, and PLRU may yield worst-case-execution-time estimates that are dramatically wrong. In a slightly modified analysis we show that the "empty cache is worst-case initial state" assumption [Petters, 2002] is wrong for FIFO, MRU, and PLRU. On the other hand, our analysis results show that LRU lends itself well to measurement- or simulation-based approaches as the influence of the initial cache state is minimal.

1.1 Structure of the Thesis

In Chapter 2, we introduce general cache notions and describe all replacement policies studied in this thesis. Chapter 3 describes abstract interpretation, a theory of sound approximation of semantics. Mathematical foundations of abstract interpretation and data-flow analysis required there can be found in Appendix C. Chapter 4 discusses challenges in cache analysis and presents state-of-the-art *may* and *must* analyses for LRU based on abstract interpretation.

Chapter 5 explores limits on the precision of static cache analyses by introducing and evaluating predictability metrics. Chapter 6 studies the relative competitiveness of cache replacement policies and how it can be used to obtain new cache analyses. Chapter 7 studies the sensitivity of cache replacement policies and discusses its impact on the correctness of measurement-based timing analysis. Finally, Chapter 8 concludes.

Ideally one would desire an indefinitely large memory capacity such that any particular [...] word would be immediately available. [...] We are [...] forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

A. W. Burks, H.H. Goldstine, and J. von Neumann (1946)

2 Caches

Since the early days of the computer, architects are faced with the problem that different memory technologies are either fast but expensive (and thus small[1]) or cheap (and thus possibly large) but slow, but not both fast and cheap at the same time. As the above quotation demonstrates, it was recognized early on, that it is necessary to construct hierarchies of memory of different technologies to obtain a large and on the average fast memory.

Figure 2.1 depicts a typical memory hierarchy of a modern personal computer. Access latencies range from one cycle for accesses to the small and expensive register file to millions of cycles for accesses to the spacious hard disk. Observe that different parts of the memory hierarchy are managed by different entities. In contrast to the registers, the main memory, and the hard disk, which are managed by the compiler and the operating system, respectively, the caches are managed by the hardware. As a consequence their existence is transparent to software running on the system in terms of functional behavior. So how do caches work and why do memory hierarchies and in particular caches usually work?

[1] In fact, it is not just a matter of how much one is willing to pay. For a given guaranteed latency, technological constraints obviate the construction of an arbitrarily large memory.

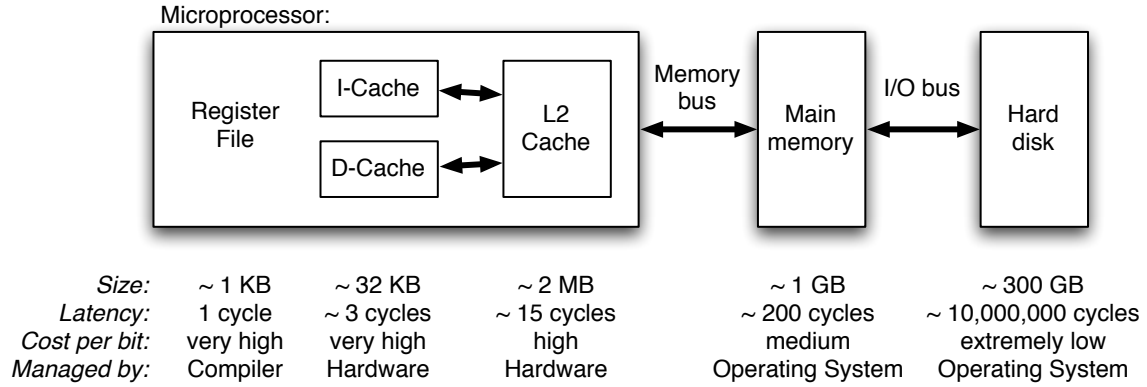


Figure 2.1: Memory hierarchy of a personal computer.

How they work.

Caches store a small subset of the data of the backing store, e.g. the main memory in the case of processor caches. Memory accesses are serviced from the cache. If it contains the referenced data, a *cache hit*, the data can be returned at a low latency. Upon a *cache miss*, i.e., the data is not contained in the cache, it first needs to be transferred from the backing store to the cache, replacing other data. Due to the employed technology, access latencies to the cache are much lower than those of the backing store. If most accesses are cache hits the average latency is close to that of the cache. However, due to the relatively small size of the cache, the average cost per bit of storage is closer to that of the backing store.

Why they work.

Why are caches and other building blocks of a memory hierarchy usually effective? They benefit from *locality of reference* (also called the *principle of locality*). Two types of locality are typically distinguished:

- Temporal locality: Resources that have been referenced recently are likely to be referenced again soon afterwards.
- Spatial locality: The likelihood of referencing a resource is greater if a resource near it has been referenced recently.

Caches exploit both types of locality:

- By caching resources that have been referenced recently.
- By not only caching the referenced resources but also its immediate neighborhood. In the case of processor caches, cache lines which are fetched upon a cache miss are larger blocks of data that contain the referenced instructions or data.

Other building blocks of memory hierarchies that we have seen are the register file and the main memory. The register file is managed by the compiler, which tries to keep the values of as many as possible variables in registers that are currently live. *Paging* is the mechanism usually implemented by the operating system to manage the contents of main memory. Many of the algorithms used for *paging* are similar to those used in processor caches.

Further examples of cache organizations include the Translation Lookaside Buffer (TLB), the Branch Target Buffer (BTB, also known as Branch Target Instruction Cache), web proxy caches, DNS caches, and database caches.

Overview.

In the remainder of this chapter we will concentrate on processor cache organizations occurring in embedded real-time systems. The following section discusses the organization of individual caches, i.e., the building blocks of cache hierarchies. The succeeding section is devoted solely to *replacement policies* as they are the main topic of this thesis.

2.1 Processor Caches

Processor caches are located on the die of the microprocessor. Depending on their position in the cache hierarchy, they are either connected to lower level caches or to the main memory via the memory bus.

This section shall discuss how an individual processor cache is logically organized, i.e., how it is structured internally, and how it manages memory accesses.

To reduce traffic and management overhead, the main memory is logically partitioned into a set of *memory blocks* M of size b bytes. Memory blocks are cached as a whole in cache lines of equal size. This approach is viable due to *spatial locality*. Usually, b is a power of two. This way the block offset is determined by the most significant bits of a memory address.

When accessing a memory block one has to determine whether the memory block is currently stored in the cache (cache hit) or not (cache miss). To enable an efficient look-up, each memory block can be stored in a small number of cache lines only. For this purpose, caches are partitioned into equally-sized cache sets. The size of a cache set is called the *associativity* k of the cache. The number of such equally sized cache sets s is usually a power of two, such that the set number is determined by the least significant bits of the block number, the *index*. There are two special instances of *set-associative* caches:

- Caches with associativity 1, i.e., the address uniquely determines the location of the data in the cache. This is called *direct-mapped*. Such caches are particularly cheap to implement, as the cache lookup is very simple: it is not necessary to compare several tags to determine whether an access is a hit or a miss.
- Caches with only one set, i.e., all addresses map to the same set and could be stored in any line of the cache. Such *fully-associative* caches are only implemented in hardware for very small caches, as the cache lookup and replacement policy is otherwise too expensive to implement. Examples of fully-associative caches are most translation-lookaside buffers and small branch target instruction caches.

Typical associativities are powers of two less than or equal to 32.

The remaining most-significant bits of the address, known as the *tag* are stored along with each cache line to finally decide, whether and where a memory block is cached within a set. Figure 2.2 illustrates the organization of a set-associative processor cache. In the illustration, the cache is vertically divided into its cache sets. It can be horizontally divided into its k *ways*. A *way* contains one line per cache set.

Let us go through the actions taken by the cache upon a reading memory access using Figure 2.2. The *index* of the address determines the cache set which might contain the requested data. To determine whether it is indeed contained, the *tags* of the cache set's lines are compared to the *tag* of the address. To reduce the latency on a hit, this can be done in parallel. If the referenced block is found, the *block offset* is used to return the appropriate part of the block. Upon a cache miss, the block containing the requested address is fetched from memory and inserted into the appropriate cache set.

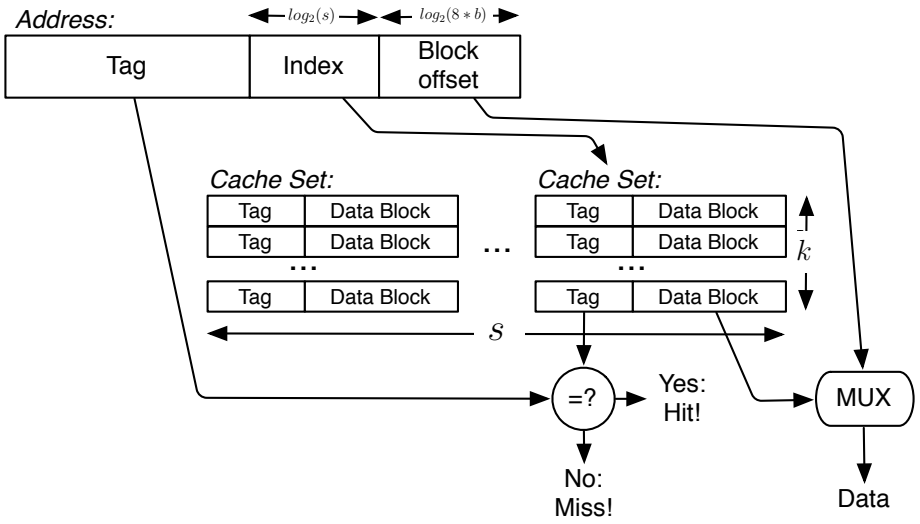


Figure 2.2: Logical organization of a k-way set-associative cache

Since the number of memory blocks that map to a set is usually far greater than the associativity of the cache, a so-called *replacement policy* must decide which memory block to replace upon a cache miss. To facilitate useful replacement decisions a number of status bits is maintained that store information about previous accesses. The following section is devoted to describing replacement policies in detail.

We have seen how reading memory access are handled. There are several choices in the design of a processor cache regarding writing accesses. Two questions arise:

1. When is the data written to the backing store?
   Either the data is written immediately to the backing store on every write or it is only written to the backing store when its cache line is replaced from the cache. The former *write policy* is called *write-through*, the latter *write-back*. *Write-through* may cause unnecessary data transfers to the backing store if a cache line is modified several times before being replaced. On the other hand *write-back* requires an additional status bit to indicate whether a cache line has been modified and must therefore be written back.

2. What happens upon a write-miss?
   If the memory block being modified is not contained in the cache, one can either bypass the cache and make the modification in the backing store (*no write-allocate*) or one can allocate a cache line and write to that line (*write-allocate*). *Write-back* caches usually use *write-allocate* hoping that subsequent writes to the same block will be captured by the cache. *Write-through* caches usually use *no write-allocate* as subsequent writes to the same block will have to be written through, too.

Another design choice in processor caches is whether to use physical or virtual addresses for tags and indices. Embedded systems currently do not employ virtual memory, so we will not elaborate on the various possibilities involving virtual addresses. An extensive discussion can be found in [Jacob et al., 2007].

2.2 Replacement Policies

Since caches can only store a small subset of the contents of the backing store, eventually its sets will fill up. Upon a cache miss to a "full" cache set, the *replacement policy* has to decide which memory block to replace. Most replacement policies base this decision solely on accesses to that particular cache set. We are aware of only one exception to this rule: the Pseudo Round-Robin policy employed in the Motorola Coldfire 5307 [Thesing, 2004].

The goal of the replacement policy is to minimize the number of cache misses. To this end it tries to replace the "least useful" memory block of a cache set.

We will now explain prominent replacement policies that follow different strategies to identify the "least useful" block:

- Belady's OPT – the optimal offline replacement policy
- Least Recently Used (LRU) used in Intel Pentium I and MIPS 24K/34K
- First-In First-Out (FIFO or Round-Robin) used in Motorola PowerPC 56x, Intel XScale, ARM9, ARM11
- Most Recently Used (MRU) as described in [Al-Zoubi et al., 2004, Malamy et al., 1994]
- Pseudo-LRU (PLRU) used in Intel Pentium II-IV and PowerPC 75x
- Pseudo Round-Robin (Pseudo-RR) used in Motorola Coldfire 5307, as described in [Thesing, 2004]

Except for Pseudo-RR, these policies treat each cache set independently of the other sets. Therefore, it usually suffices to present their behavior on individual cache sets. In addition to describing the way the policies operate, we will try to give an intuition of what makes it easy or difficult to statically predict their behavior. In Chapter 5 we will substantiate these intuitions.

2.2.1 Domains and Notations

Figure 2.3 introduces domains and metavariables for *memory blocks*, *cache-set states*, and *update*-functions that model the behavior of the replacement policies.

Each replacement policy P induces a set of *reachable cache-set states* C^P. After starting up the hardware, each cache-set is in its *initial state* i^P. Cache replacement policies change a cache set's state upon memory accesses. We represent cache sets by tuples

a, b, c	$\in$	M	the set of memory blocks
$\bot, a, b, c$	$\in$	$M_\bot = M \cup \{\bot\}$	the set of memory blocks of empty lines
P, Q	$\in$	$Policy$	the class of replacement policies
$[b, e, c, f]_P, i^P, p, q$	$\in$	$C^P \subseteq M_\bot^k \times \mathbb{B}^l$	the set of reachable cache-set states of policy P with i^P the initial state of P after starting up the hardware
$update_P$	$\in$	$C^P \times M \to C^P$	function modeling the effect of accessing a memory block under policy P

Figure 2.3: Domains and notations.

of memory blocks. For example, a set consisting of 4 lines containing memory blocks b, c, e, d may be denoted as $[b, c, e, d] \in C^P \subseteq M_\bot^4$. Invalid (empty) cache lines are denoted by $\bot$. Depending on the policy, the order of the memory blocks in the tuple will have different meanings. In the case of MRU and PLRU additional status bits will be necessary to represent cache states, as in $[c, d, e, b]_{0111} \in C^{\mathrm{MRU}(4)} \subseteq M_\bot^4 \times \mathbb{B}^4$. Their meaning will be explained in the description of the respective policies. In general, cache-set states of replacement policies that can be implemented in hardware use only a finite number of status bits. For any policy P, $C^P \subseteq M_\bot^k \times \mathbb{B}^l$, where k is the associativity of P and l is the number of status bits, which might be 0. We sometimes call memory blocks contained in cache sets simply elements. If it is not clear from the context which policies the states belong to, we will denote it by a subscript, as in $[a, b, c, d]_{\mathrm{LRU}(4)}$, where 4 denotes the associativity of the cache. To be able to precisely argue about the different replacement policies later, we provide formal definitions of their semantics in terms of functions $update_P : C^P \times M \to C^P$. $update_P(q, c)$ computes the cache-set state after accessing memory block c in cache-set state q in policy P.

2.2.2 OPT – Optimal Replacement

In 1966, [Belady, 1966] proposed OPT, an offline policy. It is also known as MIN and BEL. OPT replaces the memory block that will not be accessed for the longest time in the future. It was first proven to be optimal by [Mattson et al., 1970]. Later, [Roy, 2007] and [Vogler, 2008] gave short proofs of its optimality.

In the case of OPT, the order of elements in our representation of cache sets does not matter, because the policy bases its decisions solely on future accesses. This will not be the case for the online policies. $[a, b, c, d]$ represents the same cache set as $[d, c, a, b]$; they should be interpreted as sets.

Example. *Consider the sequence of accesses $e, f, a, g, a, d, e, c, b, f, \ldots$ and the initial state $[a, b, c, d]$. The first seven accesses of the sequence will result in the following behavior:*

$$[a,b,c,d] \xrightarrow[evict\ b]{e} [a,e,c,d] \xrightarrow[evict\ c]{f} [a,e,f,d] \xrightarrow[evict\ f]{g} [a,e,g,d]$$

The access to e evicts b, as b's next use is farther away than the next uses of a, c, and d. Then, f replaces c. The access to a results in a hit and does not change the state. The following access to g evicts f, although it has just been added to the cache set. The accesses to a, d, and e are hits. Given our limited knowledge of the access sequence it is not clear which element would be evicted on the following access to c.

OPT cannot actually be implemented, as knowledge of future accesses is necessary to make a replacement decision; it is an offline algorithm. Although OPT cannot be implemented it is very useful in evaluating the quality of implementable online policies, as in *competitive analysis* [Sleator and Tarjan, 1985] or empirical evaluations on full traces [Al-Zoubi et al., 2004].

2.2.3 Least Recently Used

Least Recently Used (LRU) replacement replaces the least-recently-used element on a cache miss. It conceptually maintains a queue of length k for each cache set, where k is the associativity of the cache. In the case of LRU, $[b,c,e,d]$ denotes a cache set, where elements are ordered from most- to least-recently-used, i.e., b is the most-recently-used element and d is the least-recently-used element. If an element is accessed that is not yet in the cache (a miss), it is placed at the front of the queue. The last element of the queue is then removed if the set is full. In our example, an access to f would thus result in $[f,b,c,e]$. The least-recently-used element d is replaced. Upon a cache hit, the accessed element is moved from its position in the queue to the front, in this respect treating hits and misses equally. Accessing c in $[f,b,c,e]$ results in $[c,f,b,e]$.
Formally, $C^{\mathrm{LRU}(k)} \subseteq M_{\perp}^{k}$, $i^{\mathrm{LRU}(k)} = [\perp,\ldots,\perp]$, and $update_{\mathrm{LRU}(k)}(q,c)$, where k denotes the associativity of the cache, is defined as follows:

$$update_{\mathrm{LRU}(k)}([a_1,\ldots,a_k]_{\mathrm{LRU}(k)},c) := \begin{cases} [a_i,a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_k]_{\mathrm{LRU}(k)} & \text{if } c = a_i \\ [c,a_1,\ldots,a_{k-1}]_{\mathrm{LRU}(k)} & \text{if } \forall i : c \neq a_i \end{cases}$$

$update_{\mathrm{LRU}(k)}$ models the logical behavior of LRU, not its hardware implementation. A hardware implementation of LRU would not shift the memory blocks around. Instead it would maintain status bits that capture the logical ordering of the memory blocks.

Notion of Age

One can associate an *age* with a memory block a by counting the number of different memory blocks accessed after the last access to a. With this notion of age, LRU orders elements by increasing age from age 0 (most-recently-used) to $k-1$ (least-recently-used). Upon a miss, the oldest element, i.e. the one with age $k-1$ is replaced.

Rationale behind LRU

LRU performs very well in practice. The rationale behind replacing the least-recently-used element is that the longer an element has not been accessed the less likely it is to access it again in the near future. Conversely, if an element has just been accessed it is likely to be accessed again soon. LRU may be considered the natural online counterpart of the offline policy OPT: Whereas OPT replaces the element that *will* not been used for the longest time, LRU replaces the element that *has* not been used for the longest time.

Predictability

LRU has been the target of most cache analyses [Ferdinand et al., 1997, Ferdinand and Wilhelm, 1999, White et al., 1997, Ghosh et al., 1998, Chatterjee et al., 2001]. It has some desirable properties that make it easier to analyze than other policies:

- It treats hits and misses similarly: Even if it is unknown whether an access is a hit or a miss, an analysis can gain valuable information about the state of the cache set after the access.
- The amount of information gained on each access is usually greater than the information gained in other policies even if the access can be classified.

Our studies of predictability metrics and the sensitivity of replacement policies shed more light on this.

2.2.4 First In, First Out

First In, First Out (FIFO, also known as Round-Robin) bases its replacement decisions on *when* an element entered the cache, *not* on the time of its most-recent use. It replaces the element which has been resident in the cache for the longest time.

FIFO cache sets can also be seen as a queue: new elements are inserted at the front evicting elements at the end of the queue. This resembles LRU. In contrast to LRU, hits do *not* change the queue. In our representation of FIFO cache sets, elements are ordered from last-in to first-in: Assume an access to f. In $[b, c, e, d]$, d will be replaced on a miss resulting in $[f, b, c, e]$.

Formally, $C^{\mathrm{FIFO}(k)} \subseteq M_{\perp}^{k}$, $i^{\mathrm{FIFO}(k)} = [\perp, \ldots, \perp]$, and $update_{\mathrm{FIFO}(k)}(q, c)$ is defined as follows:

$$update_{\mathrm{FIFO}(k)}([a_1, \ldots, a_k]_{\mathrm{FIFO}(k)}, c) := \begin{cases} [a_1, \ldots, a_k]_{\mathrm{FIFO}(k)} & \text{if } c = a_i \\ [c, a_1, \ldots, a_{k-1}]_{\mathrm{FIFO}(k)} & \text{if } \forall i : c \neq a_i \end{cases}$$

FIFO is a popular policy mainly because it can be implemented cheaply. In many benchmarks it performs only slightly worse than LRU [Al-Zoubi et al., 2004], although there are some exceptions, where FIFO incurs a considerable performance degradation compared with LRU.

Predictability

Due to its non-uniform treatment of hits and misses, it is difficult to obtain information about FIFO cache contents in a static cache analysis: An element may be evicted right after being accessed, if it resides in the first-in position. Statically predicting cache misses for FIFO is hard, as we will later demonstrate.

2.2.5 Most Recently Used

Most Recently Used (MRU) does *not* replace the most-recently-used element. Instead it stores one status bit per cache line. In the following, we call these bits MRU-bits. Every access to a line sets its MRU-bit to 1, indicating that the line was recently used. Whenever the last remaining 0 bit of a set's status bits is set to 1, all other bits are reset to 0. Upon a cache miss, a line whose MRU-bit is 0 is replaced. At cache misses, the line with lowest index (in our representation the left-most) whose MRU-bit is 0 is replaced.

We represent a state of an MRU cache set as $[a, b, c, d]_{0101}$, where 0101 are the MRU-bits and $a, \ldots, d$ are the contents of the set. On this state an access to e would yield a cache miss and the new state $[e, b, c, d]_{1101}$, i.e., the left-most element with MRU-bit 0 has been replaced. Accessing d leaves the state unchanged. A hit on c forces a reset of the MRU-bits: $[e, b, c, d]_{0010}$.

Formally, $C^{\mathrm{MRU}(k)} \subseteq M_\perp^k \times \mathbb{B}^k$, $i^{\mathrm{MRU}(k)} = [\perp, \ldots, \perp]_{0\ldots0}$, and $update_{\mathrm{MRU}(k)}(q, c)$ is defined as follows:

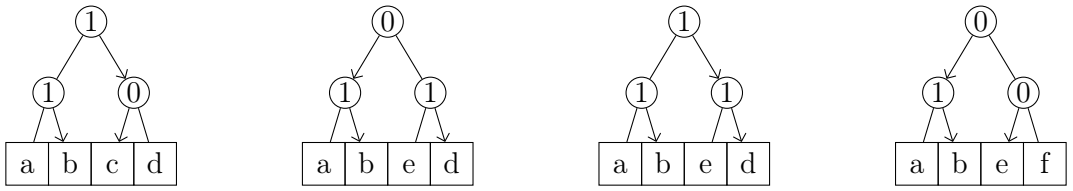
$$
update_{\mathrm{MRU}(k)}([a_1, \ldots, a_k]_{u_1 \ldots u_k}, c) := \begin{cases}
[a_1, \ldots, a_k]_{u_1 \ldots u_{i-1} 1 u_{i+1} \ldots u_k} & \text{if } c = a_i \wedge \exists j \neq i : u_j = 0 \\
[a_1, \ldots, a_k]_{\underbrace{0\ldots0}_{i-1} 1 \underbrace{0\ldots0}_{k-i}} & \text{if } c = a_i \wedge \forall j \neq i : u_j = 1 \\
[a_1, \ldots, a_{i-1}, c, a_{i+1}, \ldots, a_k]_{u_1 \ldots u_{i-1} 1 u_{i+1} \ldots u_k} & \text{if } \forall i : c \neq a_i \wedge \forall j < i : u_j = 1 \\
 & \quad \wedge\, u_i = 0 \wedge \exists j > i : u_j = 0 \\
[a_1, \ldots, a_{i-1}, c, a_{i+1}, \ldots, a_k]_{\underbrace{0\ldots0}_{i-1} 1 \underbrace{0\ldots0}_{k-i}} & \text{if } \forall i : c \neq a_i \wedge \forall j \neq i : u_j = 1
\end{cases}
$$

Predictability

Replacement behavior of MRU is heavily influenced by when the "last" bit is set to 1 and all other bits are reset to 0. This asymmetry in the last bit set to 1 makes MRU unpredictable. We will later see that it is impossible to ever infer the precise contents of an MRU cache set.

2.2.6 Pseudo-LRU

Pseudo-LRU (PLRU) is a tree-based approximation of the LRU policy. It arranges the cache lines at the leaves of a tree with $k-1$ "tree bits" pointing to the line to be replaced next; a 0 indicating the left subtree, a 1 indicating the right. After every access, all tree bits on the path from the accessed line to the root are set to point away from the line. Other tree bits are left untouched. Consider the following example of 3 consecutive accesses to a set of a 4-way set-associative PLRU cache:



Initial cache-set state $[a, b, c, d]_{110}$. After a miss on e. State: $[a, b, e, d]_{011}$. After a hit on a. State: $[a, b, e, d]_{111}$. After a miss on f. State: $[a, b, e, f]_{010}$.

In the initial state of the example, the tree bits point to the line containing memory block c. We textually represent a PLRU-state by the contents of its cache lines and the preorder traversal of its tree bits. The initial state in the example is thus written $[a, b, c, d]_{110}$. A miss on e evicts the memory block c which was pointed to by the tree bits. To protect e from eviction, all tree bits on the path to the root of the tree are made to point away from it. Similarly, upon the following hit on a, the bits on the path from a to the root of the tree are made to point away from a. Note that they are not necessarily flipped. Another access to a would not change the tree bits at all as they already point away from a. Finally, a miss on f eliminates d from the cache set. So, $C^{\mathrm{PLRU}(k)} \subseteq M_\bot^k \times \mathbb{B}^{k-1}$ and $i^{\mathrm{PLRU}(k)} = [\bot, \ldots, \bot]_{0\ldots0}$.

For associativity 1, $update_{\mathrm{PLRU}(k)}$ is trivial:

$$update_{\mathrm{PLRU}(1)}([a_1], c) := [c]$$

$update_{\mathrm{PLRU}(k)}$ can be defined recursively expressing the update of a $2k$-associative set in terms of the update of a k-associative one: Let $update_{\mathrm{PLRU}(k)}([a_1, \ldots, a_k]_{l_1 \ldots l_{k-1}}, c) = [a'_1, \ldots, a'_k]_{l'_1 \ldots l'_{k-1}}$ and $update_{\mathrm{PLRU}(k)}([b_1, \ldots, b_k]_{r_1 \ldots r_{k-1}}, c) = [b'_1, \ldots, b'_k]_{r'_1 \ldots r'_{k-1}}$. Then:

$$update_{\mathrm{PLRU}(2k)}([a_1, \ldots, a_k, b_1, \ldots, b_k]_{u l_1 \ldots l_{k-1} r_1 \ldots r_{k-1}}, c) :=$$

$$\begin{cases} [a'_1, \ldots, a'_k, b_1, \ldots, b_k]_{1 l'_1 \ldots l'_{k-1} r_1 \ldots r_{k-1}} & \text{if } \exists i : c = a_i \\ [a_1, \ldots, a_k, b'_1, \ldots, b'_k]_{0 l_1 \ldots l_{k-1} r'_1 \ldots r'_{k-1}} & \text{else if } \exists i : c = b_i \end{cases} \text{“hit”}$$

$$\begin{cases} [a'_1, \ldots, a'_k, b_1, \ldots, b_k]_{1 l'_1 \ldots l'_{k-1} r_1 \ldots r_{k-1}} & \text{else if } \exists i : a_i = \bot \\ [a_1, \ldots, a_k, b'_1, \ldots, b'_k]_{0 l_1 \ldots l_{k-1} r'_1 \ldots r'_{k-1}} & \text{else if } \exists i : b_i = \bot \end{cases} \text{“miss” invalid}$$

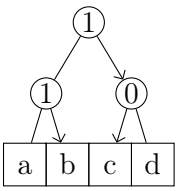
$$\begin{cases} [a'_1, \ldots, a'_k, b_1, \ldots, b_k]_{1 l'_1 \ldots l'_{k-1} r_1 \ldots r_{k-1}} & \text{else if } u = 0 \\ [a_1, \ldots, a_k, b'_1, \ldots, b'_k]_{0 l_1 \ldots l_{k-1} r'_1 \ldots r'_{k-1}} & \text{else if } u = 1 \end{cases} \text{“miss” valid}$$

In the "hit" case, we need to flip the u bit away[2] from the subtree that c is contained in and update the subtree similarly. In the "miss" case, we need to insert c into the subtree pointed to by the u bit unless there is an invalid/empty line:

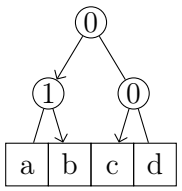
PLRU as described in [Freescale Semiconductor Inc., 2002] has a special treatment of invalid lines: On a cache miss, invalid lines are filled from left to right, ignoring the tree bits. However, the tree bits are still updated.

Predictability or PLRU vs LRU

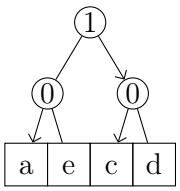
PLRU is much cheaper to implement than true LRU in terms of storage requirements and update logic. In practice PLRU performs well [Al-Zoubi et al., 2004]. However, in rare cases, its replacement decisions differ substantially from those of LRU, i.e., it does not replace the least-recently-used element. The reason is that accesses do not only protect themselves from eviction; they also "rejuvenate" their neighborhood. Consider the following example:



Initial cache-set state. State: $[a, b, c, d]_{110}$.



After accessing d. State: $[a, b, c, d]_{010}$.



After a miss on e. State: $[a, e, c, d]_{100}$.

In the initial state a miss would evict c. However, accessing its neighbor d flips the tree bit at the root of the tree. The access to d protects c as well. The following miss evicts b instead of c. In this way, elements may survive indefinitely without ever being accessed.

This property is detrimental to its predictability. Our predictability metrics will precisely quantify this. Furthermore, our relative competitiveness study shed more light on the relation of PLRU and LRU.

2.2.7 Pseudo Round-Robin

Pseudo Round-Robin (Pseudo-RR) is the replacement policy used in the Motorola Coldfire 5307 [Thesing, 2004]. In contrast to the policies described previously, cache sets are not managed independently by Pseudo-RR: Replacement decisions for a cache set are not only based on accesses to that particular cache set, but also on other accesses.

Pseudo-RR maintains *one* modulo-k counter for the entire cache, where k is the associativity of the cache. This counter points to one of the k ways of the cache. In other words, the counter points to one cache line in each cache set.

[2] If it already points away from the accessed element, it will be left unchanged.

Upon a cache miss, the cache line in the referenced set pointed to by the counter is replaced. Then, the counter is incremented. On a cache hit the counter is left untouched. Consider the following example.

Example. *The subscripts in the symbolic addresses denote the index part of the address, e.g.* a_3 *is an address whose index is* 3. a_3 *thus maps to cache set* 3. *The global modulo-4 counter is illustrated by an arrow pointing to the respective way:*

Way \ *Set*	*0*	*1*	*2*	*3*	⋯	*127*
0	b_0	c_1	f_2	d_3	…	b_{127}
1→	–	b_1	a_2	–	…	g_{127}
2	e_0	f_1	d_2	f_3	…	d_{127}
3	f_0	a_1	–	b_3	…	c_{127}

accessing $\langle d_1 a_2 e_3 \rangle$ *yields*

Way \ *Set*	*0*	*1*	*2*	*3*	⋯	*127*
0	b_0	c_1	f_2	d_3	…	b_{127}
1	–	$\mathbf{d_1}$	a_2	–	…	g_{127}
2	e_0	f_1	d_2	$\mathbf{e_3}$	…	d_{127}
3→	f_0	a_1	–	b_3	…	c_{127}

In the example, the counter initially points to way 1. *Accessing* d_1 *incurs a cache miss, replacing* b_1 *and incrementing the counter to* 2. *The access to* a_2 *is a hit leaving the counter unchanged. Finally,* e_3 *incurs another miss, replacing* f_3 *in set* 3 *and way* 2. *The counter is incremented to way* 3.

Predictability

PSEUDO-RR can be seen as a cheaper to implement variant of FIFO, which is also known as Round-Robin. FIFO is already quite unpredictable in itself. The difference is that FIFO maintains one counter per set, whereas PSEUDO-RR shares one counter among all sets. By sharing one counter among all sets, uncertainty about memory accesses in one set spreads to uncertainty in other sets. Assume an analysis has inferred that the global counter currently points to way 1 of the cache. It takes only 3 accesses that cannot be classified as hits or misses to completely lose this information. If one does not know the position of the counter, it is unclear which elements are evicted on cache misses. This yields further uncertainty about memory accesses later.

2.2.8 Implementation Issues

Although LRU achieves higher cache hit ratios than FIFO or PLRU on the average, the latter policies are more widely used in hardware caches. Why is that? The choice of which policy to implement is governed by two properties, *performance* and the *cost of implementation.*

Performance

The cache hit ratio has a strong impact on the cache performance. However, it also depends on the miss and the hit latencies. Upon a miss, the operations performed by the replacement policy are usually not on the critical path, as the requested memory

block needs to fetched from the backing store, a slower device in the memory hierarchy. However, in the case of a hit, the replacement policies' operations are on the critical path, as the requested memory block is immediately available. This is one of the reasons for implementing FIFO instead of LRU. Upon a cache hit, LRU needs to "move" the cache line that was accessed to the most-recently-used position and shift other lines down[3]. FIFO simply ignores the access and can thus provide a better hit latency.

Cost of Implementation

Different policies consume different amounts of die area. Area consumption can be divided into two parts: storage of status bits, like the PLRU tree bits, and circuits implementing the actual update and replacement logic.

The following table shows the minimal number of status bits required to implement the replacement policies per cache set in terms of the associativity k of the cache.

Policy	# Status bits in terms of k	$k = 4$	$k = 8$	$k = 16$	$k = 32$
LRU	$\lceil log_2(k!) \rceil \in \mathcal{O}(k \cdot log_2 k)$	5	16	45	118
FIFO	$\lceil log_2 k \rceil$	2	3	4	5
MRU	k	4	8	16	32
PLRU	$k - 1$	3	7	15	31

LRU needs $\lceil log_2(k!) \rceil$ status bits to encode the $k!$ different orderings of its cache lines regarding recent accesses. In contrast, FIFO needs to maintain a simple replacement counter, pointing to the cache line to replace next. The number of status bits for MRU and PLRU follow directly from the description of the two policies. In contrast to the other policies, PSEUDO-RR does not maintain independent status bits for each set. Instead it maintains only $\lceil log_2 k \rceil$ status bits for the entire cache.

At higher associativities it becomes quite expensive to implement LRU in terms of status bits. In addition, many implementations use more than the minimal number of status bits. This way it is possible to reduce the length of the critical path upon cache hits. In general, there are often several alternative implementations of a given policy that represent different tradeoffs between implementation cost and hit latency [Peir et al., 1998, Sudarshan et al., 2004]. [Ackland et al., 2000] have shown that in contrast to popular belief among computer architects, LRU replacement can be implemented with a 1-cycle update up to associativity 16.

Studies of average-case cache hit ratios [Al-Zoubi et al., 2004] show only a small advantage of LRU over FIFO, MRU, and PLRU. In some cases, it is even outperformed by the latter. Regarding its disadvantages in terms of implementation cost and hit latency, it is hardly surprising that LRU is rarely implemented. However, when it comes to *predictable* performance, LRU outperforms its competitors, as we will demonstrate. Even smaller, and thus cheaper to implement LRU-controlled caches might provide better performance guarantees, than their larger, FIFO, MRU, or PLRU-controlled cousins.

[3]Note, that the contents of the cache lines are not actually moved around. Instead, the status bits record information about the order of the cache lines.

$\langle b, b\rangle, \langle b, c, d\rangle, s, t$	$\in$	$S = M^*$	the set of finite access sequences
$\langle b, c, d\rangle, s, t$	$\in$	$S^{\neq} \subset S$;	the set of finite access sequences with pairwise different accesses
$\circ$	:	$S \times S \to S$	concatenation of access sequences

Figure 2.4: Access sequences.

Invalidation of Cache Lines

In a multiprocessor or multicore environment, processes running on different processors or cores may share memory. Several versions of a shared datum may reside in the private caches of different processors. To ensure that changes made in one process will become visible to other processes, a cache coherence protocol [Jacob et al., 2007] is employed. There are different coherence protocols, but in all of them a cache line may be invalidated in one private cache if the datum stored in the cache line is modified by another process on another processor/core. This way arbitrary lines may be invalidated. Even the most-recently-used line may be invalid. In the following, we assume that invalid/empty[4] lines are uniformly treated like valid lines in LRU, FIFO, and MRU, as we did not find documentation of other treatments. It depends on the particular implementation of a replacement policy how such lines are dealt with. It would be reasonable to replace a more-recently-used but invalid line before replacing a less-recently-used but valid line. The PLRU implementation found in the PowerPC 75x series [Freescale Semiconductor Inc., 2002], exhibits a documented non-uniform treatment of invalid lines, which we take into account. On a cache miss, invalid lines are chosen before regarding the tree bits.

Current hard real-time embedded systems do not use multicore processors, so the treatment of invalid lines is irrelevant. However, future embedded systems are expected to do so. Then, the treatment of invalid lines will matter, and some of our analyses in the following chapters might have to be adapted to the particular implementations of the policies with respect to invalid lines.

2.2.9 Additional Domains and Notations

Earlier, we have introduced the tuple representation of cache sets, which was used in the explanation of the replacement policies. In the following chapters, we will need additional domains and notations to reason about access sequences and their effect on replacement policies.

Figure 2.4 introduces domains and metavariables for *access sequences*. We denote the *set of access sequences* by S. Individual access sequences are written like this: $\langle b, c, d\rangle$, where accesses occur from left to right. Sometimes we restrict our attention to access sequences that contain pairwise different accesses only. The set of such sequences is denoted by $S^{\neq}$. Access sequences can be concatenated by $\circ$, as in $\langle b, c, d\rangle \circ \langle f, d, a\rangle = \langle b, c, d, f, d, a\rangle \notin S^{\neq}$.

[4] We use the terms interchangeably.

CC_P	:	$C^P \to \mathcal{P}(M)$	function computing the contents of cache-set state
SC	:	$S \to \mathcal{P}(M)$	function computing the contents of an access sequence

Figure 2.5: Extracting the contents of cache-set states and access sequences.

$update_P$	:	$C^P \times S \to C^P$	function computing the effect of an access sequence on a cache-set state under policy P
m_P	:	$C^P \times S \to \mathbb{N}$	the number of misses incurred by policy P on a given access sequence and cache-set state.
h_P	:	$C^P \times S \to \mathbb{N}$	the number of hits of policy P on a given access sequence and cache-set state.

Figure 2.6: Functions modeling cache replacement policies

Sometimes we are only interested in which memory blocks are contained in a cache set or an access sequence but not in their order. Figure 2.5 introduces functions to extract the contents of cache-set states and sequences. The function $CC_P : C^P \to \mathcal{P}(M)$ returns the *"cache contents"* of a cache set. For example, $CC_{\text{LRU(4)}}([b, e, c, f]_{\text{LRU(4)}}) = \{b, c, e, f\}$. The initial cache-set state i^P of a policy, that the cache assumes at startup of the hardware, is empty:

$$CC_P(i^P) = \varnothing$$

Similarly, $SC : S \to \mathcal{P}(M)$ computes the contents of a given access sequence. For example, $SC(\langle d, b, a, f, a\rangle) = \{a, b, d, f\}$.

Cache replacement policies change a cache set's state upon memory accesses. We have previously introduced $update_P$ which models the behavior of policy P. $update_P$ can be lifted from individual memory blocks to sequences of memory blocks:

$$\begin{aligned} update_P(q, \langle\rangle) &:= q \\ update_P(q, \langle b_1, \ldots, b_n\rangle) &:= update_P(update_P(q, b_1), \langle b_2, \ldots, b_n\rangle) \end{aligned}$$

For instance, $update_{\text{FIFO(4)}}([b, e, d, f]_{\text{FIFO(4)}}, \langle f, c\rangle) = update_{\text{FIFO(4)}}([b, e, d, f]_{\text{FIFO(4)}}, \langle c\rangle)$ $= [c, b, e, d]_{\text{FIFO(4)}}$. In contrast, $update_{\text{LRU(4)}}([b, e, d, f]_{\text{LRU(4)}}, \langle f, c\rangle)$ $= update_{\text{LRU(4)}}([f, b, e, d]_{\text{LRU(4)}}, \langle c\rangle) = [c, f, b, e]_{\text{LRU(4)}}$. The lifted $update_P$-functions allow us to express that cache-set states are reachable from the initial state:

$$\forall q \in C^P : \exists s \in S : q = update_P(i^P, s)$$

Functions modeling cache replacement policies are introduced in Figure 2.6. $update_P(q, s)$ has been introduced before. $m_P(q, s)$ and $h_P(q, s)$ compute *the number of of misses and hits*, respectively, of policy P starting in state q processing access sequence s. For instance, $m_{\text{LRU(4)}}([b, e, d, f]_{\text{LRU(4)}}, \langle f, c, f\rangle) = 1$. In contrast, as the second access to f results in a miss, $m_{\text{FIFO(4)}}([b, e, d, f]_{\text{FIFO(4)}}, \langle f, c, f\rangle) = 2$.

m_P can be defined in terms of $update_P$ and CC_P:

$$\begin{aligned} m_P(q, \langle\rangle) &:= 0 \\ m_P(q, \langle b \rangle) &:= \begin{cases} 1 & \text{if } b \notin CC_P(q) \quad \text{“miss”} \\ 0 & \text{if } b \in CC_P(q) \quad \text{“hit”} \end{cases} \\ m_P(q, \langle b_1, \ldots, b_n \rangle) &:= m_P(q, \langle b_1 \rangle) + m_P(update_P(q, \langle b_1 \rangle), \langle b_2, \ldots, b_n \rangle) \end{aligned}$$

As each access is either a hit or a miss, $h_P(q, s) = |s| - m_P(q, s)$, where $|s|$ denotes the length of the access sequence s.

3 Abstract Interpretation

Abstract interpretation [Cousot and Cousot, 1976, Cousot and Cousot, 1977] is a theory of sound semantics-based program analyses. Properties of the concrete semantics of a computer system are usually undecidable. For instance, it is in general undecidable whether a memory access will cause a cache hit or a cache miss. Therefore, one is forced to approximate the system's semantics. Such an approximation should be sound such that only true properties can be derived from it. Abstract interpretation provides a theory of deriving such sound approximations.

In this chapter, we describe some of the basics of abstract interpretation and data-flow analysis, which predates abstract interpretation. Mathematical foundations, like partial orders, complete lattices, fixed points, and chains, are introduced in Appendix C. In Chapter 4, we present a state-of-the-art cache analysis based on abstract interpretation.

3.1 Collecting Semantics

In order to develop a notion of sound approximation we first need to introduce a concrete semantics. Programs may start their execution in a set of initial states. For instance, the initial valuation of variables and the state of the cache may vary. To be able to take all possible initial states into account, the concrete semantics which operates on individual concrete states is lifted to a collecting semantics which operates on sets of concrete states. Based on a collecting path semantics we will go on to define the sticky collecting semantics, which maps each program point to the set of concrete states that may arise at that program point during program execution. We define these different semantics for programs represented by control-flow graphs:

Definition 3.1 (Control-flow graph, path)**.**
A program P may be represented by a control-flow graph $G = (V, E, i)$*, where*

- *the nodes V represent program statements,*
- *the edges $E \subseteq V \times V$ represent possible control-flow, and*
- *$i \in V$ represents the start node, which has no incoming edges: $\neg\exists v \in V : (v, i) \in E$.*

A sequence $(v_1, \ldots, v_n) \in V^*$ is a path *through a control-flow graph $G = (V, E, i)$ iff $v_1 = i$ and $\forall j \in \{1, \ldots, n-1\} : (v_j, v_{j+1}) \in E$. A sequence $(v_1, \ldots, v_n) \in V^*$ is a* path to v *if it is a path and $(v_n, v) \in E$.*

Given a concrete transformer $f : V \to \mathcal{D}_{conc} \to \mathcal{D}_{conc}$ that computes the effect of a program statement $v \in V$ on a concrete state $s \in \mathcal{D}_{conc}$, one can define the semantics $[\![\pi]\!]_{conc} : \mathcal{D}_{conc} \to \mathcal{D}_{conc}$ of a path $\pi = (v_1, \ldots, v_n) \in V^*$ through the control-flow graph:

Definition 3.2 (Path semantics)**.**

$$[\![\pi]\!]_{conc} := \begin{cases} \lambda x.x & \text{if } \pi \text{ is the empty path} \\ [\![(v_2, \ldots, v_n)]\!]_{conc} \circ f(v_1) & \text{if } \pi = (v_1, \ldots, v_n) \end{cases}$$

Program analyses shall usually determine properties for sets of initial states. Path semantics can be lifted from individual concrete states $s \in \mathcal{D}_{conc}$ to sets of concrete states $S_{coll} \in \mathcal{D}_{coll} = \mathcal{P}(\mathcal{D}_{conc})$ by first lifting the concrete transformer f to a collecting transformer $f_{coll} : V \to \mathcal{D}_{coll} \to \mathcal{D}_{coll}$ on sets of concrete states:

$$f_{coll}(v)(S_{coll}) := \{f(v)(s) \mid s \in S_{coll}\}$$

Note that the partially ordered set $(\mathcal{P}(M), \subseteq)$ is a *complete lattice* for any set M, see Appendix C. In particular, $(\mathcal{D}_{coll} = \mathcal{P}(\mathcal{D}_{conc}), \subseteq, \bigcup, \bigcap, \varnothing, \mathcal{D}_{conc})$ is a complete lattice independently of the concrete domain $\mathcal{D}_{conc}$. The partial order $\subseteq$ orders states with respect to their precision: if $A \subseteq B \in \mathcal{D}_{coll}$, then A is more precise than B, it represents fewer concrete states. Also, the collecting transformer f_{coll} is monotone by construction: given more precise information as input it computes more precise output.

$[\![\pi]\!]_{conc}$ can similarly be lifted to sets of concrete states using f_{coll}:

Definition 3.3 (Collecting path semantics)**.**

$$[\![\pi]\!]_{coll} := \begin{cases} \lambda x.x & \text{if } \pi \text{ is the empty path} \\ [\![(v_2, \ldots, v_n)]\!]_{coll} \circ f_{coll}(v_1) & \text{if } \pi = (v_1, \ldots, v_n) \end{cases}$$

Program analyses often determine properties of the *sticky collecting semantics*. Does a variable assume a constant value at a particular program point? Is the memory access associated with a particular program point always a cache hit? The sticky collecting semantics $Coll_P : V \to \mathcal{D}_{coll}$ maps a program point to the set of concrete states that may arise at that point in the program:

Definition 3.4 (Sticky collecting semantics)**.**
The sticky collecting semantics $Coll_P : V \to \mathcal{D}_{coll}$ *of a program* P*, represented by control-flow graph* G_P*, on a set of concrete initial states* $Init \subseteq \mathcal{D}_{conc}$ *is*

$$Coll_P(v) := \bigcup \{[\![\pi]\!]_{coll}(Init) \mid \pi \text{ is a path to } v \text{ in } G_P\}.$$

Other properties may need different concrete semantics. The liveness of a variable cannot be expressed in terms of the sticky collecting semantics. It needs a trace semantics. We do not go into detail about other semantics as the sticky collecting semantics suffices for cache analysis.

The sticky collecting semantics is usually not computable. The set of concrete initial states is often infinitely large or finite but prohibitively large. Furthermore, the definition quantifies over all paths through the control-flow graph which may as well be infinitely many due to loops. To address the former problem, one can replace the concrete domain by a more abstract one, that makes analysis feasible, usually at the cost of precision.

3.2 Abstract Semantics

Abstract interpretation is a general framework for soundly approximating semantics [Cousot and Cousot, 1976]. In this section, we discuss the abstraction of a sticky collecting semantics and argue about correctness conditions.

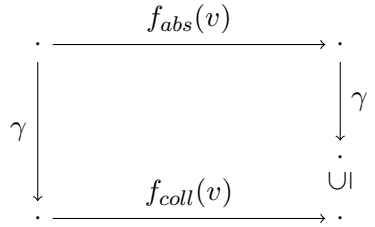
Sets of concrete states in $\mathcal{D}_{coll} = \mathcal{P}(\mathcal{D}_{conc})$ can be described by elements of an abstract domain $\mathcal{D}_{abs} = (\mathcal{D}_{abs}, \sqsubseteq)$. As in the collecting semantics, the partial order $\sqsubseteq$ can be interpreted as "more precise than", i.e., if $a \sqsubseteq b$, than a denotes more precise analysis information than b. The abstract domain $\mathcal{D}_{abs}$ should be a complete lattice so that least upper bounds exist for all subsets of $\mathcal{D}_{abs}$ such that we can always safely and uniquely combine analysis information. Analogously to the concrete semantics, we need a monotone abstract transformer $f_{abs} : V \to \mathcal{D}_{abs} \to \mathcal{D}_{abs}$ to compute the effect of program statements.

The abstract analogue to the collecting path semantics can be defined using this abstract transformer f_{abs}:

Definition 3.5 (Abstract collecting path semantics)**.**

$$[\![\pi]\!]_{abs} := \begin{cases} \lambda x.x & \text{if } \pi \text{ is the empty path} \\ [\![(v_2, \ldots, v_n)]\!]_{abs} \circ f_{abs}(v_1) & \text{if } \pi = (v_1, \ldots, v_n) \end{cases}$$

We would like the abstract collecting path semantics to be sound with respect to the collecting path semantics. To argue about the correctness of the abstract transformer and the induced abstract semantics, we need to relate the abstract with the concrete domain. One of several possibilities of doing so, is to define a monotone *concretization function* $\gamma : \mathcal{D}_{abs} \to \mathcal{D}_{coll}$ that maps each abstract state to a set of concrete states which it represents. The monotonicity of γ ensures that the partial order $\sqsubseteq$ on $\mathcal{D}_{abs}$ indeed orders abstract states by their precision. The concretization of an abstract state will be a superset of the concretization of a more precise abstract state, i.e., $a \sqsubseteq b$ implies $\gamma(a) \subseteq \gamma(b)$. The following diagram illustrates what it means for f_{abs} to be *locally consistent* with f_{coll}:



Definition 3.6 (Local consistency)**.**
An abstract transformer f_{abs} is locally consistent *with a collecting transformer f_{coll}, if*

$$\forall v \in V, S_{abs} \in \mathcal{D}_{abs} : f_{coll}(v)(\gamma(S_{abs})) \subseteq \gamma(f_{abs}(v)(S_{abs}))$$

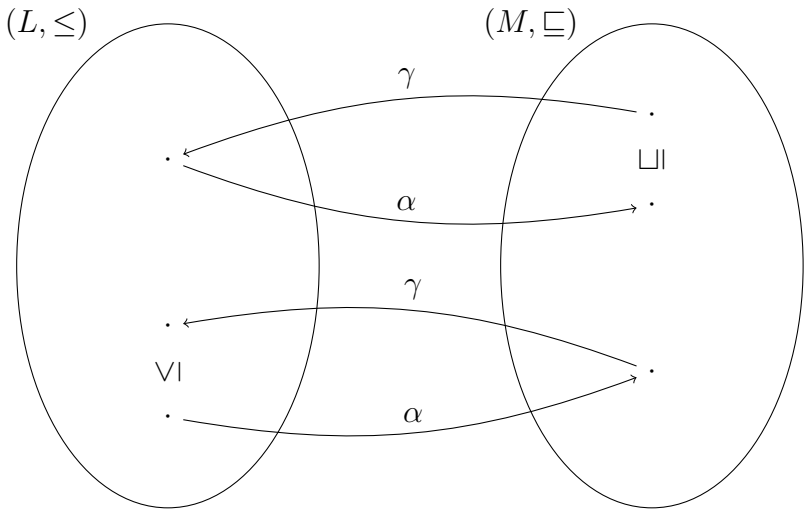


Figure 3.1: Galois connection $(L, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (M, \sqsubseteq)$.

Thus, f_{abs} should overapproximate the behavior of f_{coll}. Applying f_{abs} to an abstract state S_{abs} that represents the set of states $\gamma(S_{abs})$ in the concrete domain should yield an abstract state that includes $f_{coll}(\gamma(S_{abs}))$.

Often one can also define a monotone *abstraction function* $\alpha : \mathcal{D}_{coll} \to \mathcal{D}_{abs}$ that computes the *best* abstract description of a set of concrete states. The abstraction function α and the concretization function γ should form a *Galois connection*:

Definition 3.7 (Galois connection)**.**
Let $(L, \leq)$ *and* $(M, \sqsubseteq)$ *be partially ordered sets and* $\alpha \in L \to M$, $\gamma \in M \to L$. *We call* $(L, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (M, \sqsubseteq)$ *a* Galois connection *if* α *and* γ *are monotone functions and*

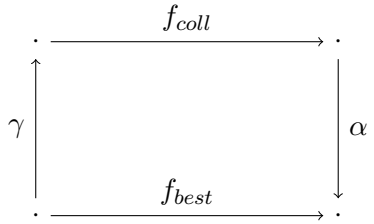
$$\begin{array}{rcl} l & \leq & \gamma(\alpha(l)) \\ \alpha(\gamma(m)) & \sqsubseteq & m \end{array}$$

for all $l \in L$ *and* $m \in M$.

This relation is also illustrated in Figure 3.1.

Intuitively, the first condition ensures that by abstracting and concretizing one can only lose precision, which is necessary for soundness. Losing some precision is usually also unavoidable, as the abstract domain is smaller than the concrete domain and thus cannot distinguish certain concrete states. The second condition ensures that the abstraction function α computes precise approximations of concrete states. However, if $\alpha(\gamma(x)) \sqsubseteq x$, then there are two abstract states that represent the same set of concrete states which is undesirable. Therefore, the condition is often strengthened to $\alpha(\gamma(m)) = m$. In that case α and γ form a *Galois insertion*.

Given a Galois connection $(\mathcal{D}_{coll}, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{D}_{abs}, \sqsubseteq)$ one can define the *best abstract transformer* $f_{best} = \alpha \circ f_{coll} \circ \gamma$:



The *local consistency* of f_{best} is easily verified. It is also the *best* abstract transformer in the sense that it will compute the most precise results in the given abstract domain. Usually it is not feasible to compute f_{best} by actually concretizing, transforming, and abstracting: concretizing may be too costly or impossible depending on the size of the concrete domain. However, it is often possible to compute f_{best} without explicitly leaving the abstract domain.

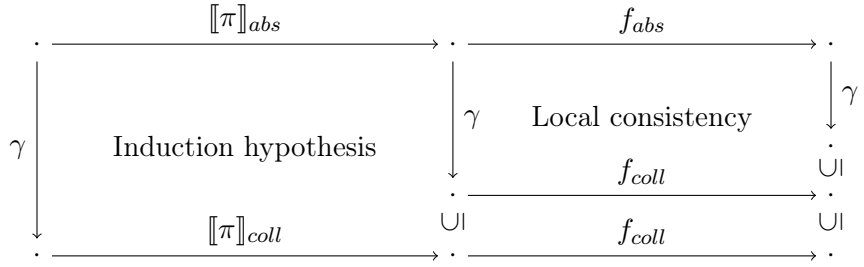
Local consistency guarantees soundness:

Lemma 3.8 (Soundness of abstract collecting path semantics)**.**
The abstract collecting path semantics is a sound overapproximation of the collecting path semantics, i.e., $\forall x \in \mathcal{D}_{abs} : (\llbracket \pi \rrbracket_{coll} \circ \gamma)(x) \subseteq (\gamma \circ \llbracket \pi \rrbracket_{abs})(x)$ *if* f_{abs} *is locally consistent.*

Proof. Proof by induction over the length of π: For the empty path, the theorem is trivially true. For the induction step, we need to show that $\forall x \in \mathcal{D}_{abs} : \llbracket (v_1) \circ \pi \rrbracket_{coll} \circ \gamma(x) \subseteq \gamma \circ \llbracket (v_1) \circ \pi \rrbracket_{abs}(x)$ provided $\forall x \in \mathcal{D}_{abs} : \llbracket \pi \rrbracket_{coll} \circ \gamma(x) \subseteq \gamma \circ \llbracket \pi \rrbracket_{abs}(x)$:

$$
\begin{array}{ll}
 & \llbracket (v_1) \circ \pi \rrbracket_{coll} \circ \gamma(x) \\
\overset{\text{Def.}}{=} & \llbracket \pi \rrbracket_{coll} \circ f_{coll}(v_1) \circ \gamma(x) \\
\overset{\text{Local consistency and monotonicity}}{\subseteq} & \llbracket \pi \rrbracket_{coll} \circ \gamma \circ f_{abs}(v_1)(x) \\
\overset{\text{Induction hypothesis}}{\subseteq} & \gamma \circ \llbracket \pi \rrbracket_{abs} \circ f_{abs}(v_1)(x) \\
\overset{\text{Def.}}{=} & \gamma \circ \llbracket (v_1) \circ \pi \rrbracket_{abs}(x)
\end{array}
$$

The proof is illustrated by the following diagram:



□

Using the abstract collecting path semantics we can also define the abstract version of the sticky collecting semantics:

Definition 3.9 (Abstract sticky collecting semantics)**.**
The abstract sticky collecting semantics *of a program P, represented by control-flow graph G_P, on an abstract initial state $Init_{abs}$ is*

$$Abs_P(v) := \bigsqcup \{[\![\pi]\!]_{abs}(Init_{abs}) \mid \pi \text{ is a path from } i \text{ to } v \text{ in } G_P\}.$$

In order to use the abstract sticky collecting semantics or an approximation of it to infer properties of the collecting semantics, the abstract sticky collecting semantics should be sound:

Theorem 3.10 (Soundness of abstract sticky collecting semantics)**.**
The abstract sticky collecting semantics is a sound overapproximation of the sticky collecting semantics, i.e., $\forall v \in V : Coll_P(v) \subseteq \gamma(Abs_P(v))$ if f_{abs} is locally consistent and the abstract initial state is the abstraction of the set of concrete initial states, i.e., $Init_{abs} = \alpha(Init)$.

Proof. Due to Lemma 3.8, $\forall \pi : \gamma([\![\pi]\!]_{abs}(Init_{abs})) \supseteq [\![\pi]\!]_{coll}(\gamma(Init_{abs}))$. Since $[\![\cdot]\!]_{coll}$ is monotone, $\gamma(Init_{abs}) = \gamma(\alpha(Init)) \supseteq Init$, and $\supseteq$ is transitive, also $\forall \pi : \gamma([\![\pi]\!]_{abs}(\alpha(Init)))$ $\supseteq [\![\pi]\!]_{coll}(Init)$. Given this and the monotonicity of γ we can prove:

$$\begin{array}{ll}
 & Coll_P(v) \\
\overset{\text{Def.}}{=} & \bigcup \{[\![\pi]\!]_{coll}(Init) \mid \pi \text{ is a path from } i \text{ to } v \text{ in } G_P\} \\
\subseteq & \bigcup \{\gamma([\![\pi]\!]_{abs}(\alpha(Init))) \mid \pi \text{ is a path from } i \text{ to } v \text{ in } G_P\} \\
\overset{\text{Monotonicity of } \gamma}{\subseteq} & \gamma(\bigsqcup \{[\![\pi]\!]_{abs}(\alpha(Init)) \mid \pi \text{ is a path from } i \text{ to } v \text{ in } G_P\}) \\
\overset{\text{Def.}}{=} & \gamma(Abs_P(v))
\end{array}$$

□

3.3 Data-Flow Analysis, *MFP* vs *MOP*

Data-flow analysis [Kildall, 1973, Kam and Ullman, 1977] predates abstract interpretation. It does not have an inherent notion of correctness with respect to a concrete semantics. Given abstract transformers over a complete lattice it allows to approximate the abstract sticky collecting semantics Abs_P of a program P. Abstract interpretation provides semantics-based soundness to data-flow analyses; however, the applicability of abstract interpretation is not limited to data-flow analyses.

Abs_P is known as the "meet over all paths" solution (MOP_P) in data-flow analysis. Following this custom we will denote it by MOP_P from now on. This nomenclature is somewhat confusing as our definition joins the semantics of all paths. According to [Nielson et al., 1999], this is for historic reasons: classical literature tends to focus on analyses where $\bigsqcup$ is $\bigcap$.

The definition of the *MOP* solution does not provide an obvious way of computing it. In the presence of loops, enumerating all paths and computing their semantics is impossible. It turns out that the *MOP* solution is not computable in general [Kam and Ullman, 1977]. However, under certain circumstances it is possible to compute the maximal fixed point solution (*MFP*). The *MFP* solution is always a safe approximation of the *MOP* solution. For an important class of problems it even coincides with *MOP*:

Definition 3.11 (Maximal fixed point solution, *MFP*)**.**
The maximal fixed point solution $MFP_P(G_P, Init_{abs}, f_{abs}) : V_P \to \mathcal{D}_{abs}$ *of a data-flow problem defined by a control-flow graph* $G_P = (V_P, E_P, i_P)$*, a complete lattice* $\mathcal{D}_{abs}$*, an abstract initial state* $Init_{abs}$*, and monotone abstract transformer* $f_{abs} : V_P \to \mathcal{D}_{abs} \to \mathcal{D}_{abs}$ *is the least fixed point* $lfp(F_P)$ *of the functional* $F_P : (V_P \to \mathcal{D}_{abs}) \to (V_P \to \mathcal{D}_{abs})$

$$F_P(f)(v) := \begin{cases} \alpha(Init_{abs}) & \text{if } v = i_P \\ \bigsqcup\{f_{abs}(v')(f(v')) \mid (v', v) \in E_P\} & \text{otherwise} \end{cases}$$

The *MFP* solution is not called minimal fixed point solution for the same historic reasons as in the case of the *MOP* solution. The *MFP* solution safely approximates the *MOP* solution:

Theorem 3.12 (*MFP* vs *MOP*)**.**
The maximal fixed point solution $MFP_P \equiv lfp(F_P)$ *is a safe approximation of the "meet over all paths" solution* $MOP_P \equiv Abs_P$*:*

$$\forall v \in V_P : MOP(v) \sqsubseteq MFP_F(v)$$

If all transfer functions $f_{abs}(v)$ *are distributive, the two solutions coincide:*

$$\forall v \in V_P : MOP(v) = MFP_F(v)$$

Proof. See [Nielson et al., 1999]. Intuitively, distributivity allows to join intermediate results in the *MFP* solution without losing any precision. □

Least fixed points of a function $f : A \to A$ can be computed by Kleene's fixed point iteration if the underlying domain A satisfies the *ascending chain condition*, see Theorem C.11. According to Lemma C.10, the *total function space* between S and L, $S \to L$ satisfies the ascending chain condition if S is a finite set and L satisfies the ascending chain condition. The $lfp(F_P)$ can thus be computed by Kleene's fixed point iteration if the abstract domain $\mathcal{D}_{abs}$ satisfies the ascending chain condition as V_P is always finite.

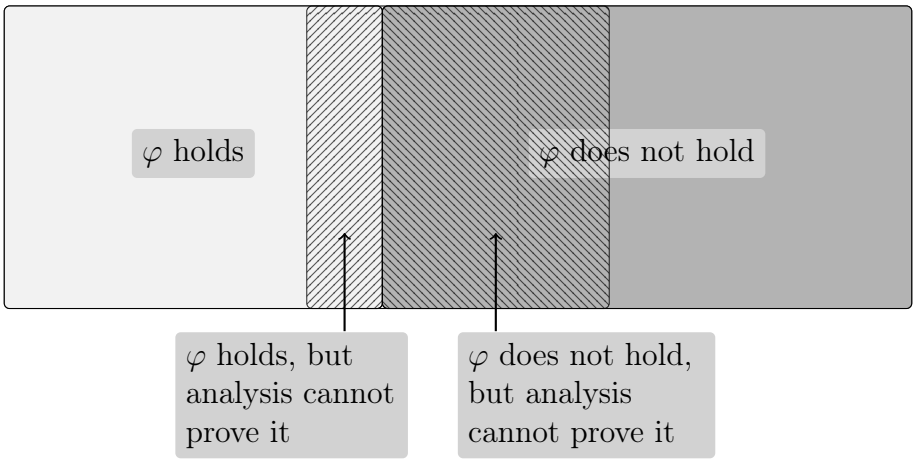
3.4 Properties and Uncertainty

Ultimately, we are interested in computing properties of the sticky collecting semantics. We would like to know whether some predicate $\varphi \subseteq \mathcal{D}_{conc}$ holds in *all* concrete states $Coll_P(v)$ that reach program point v. As we can only compute the *MFP* solution on

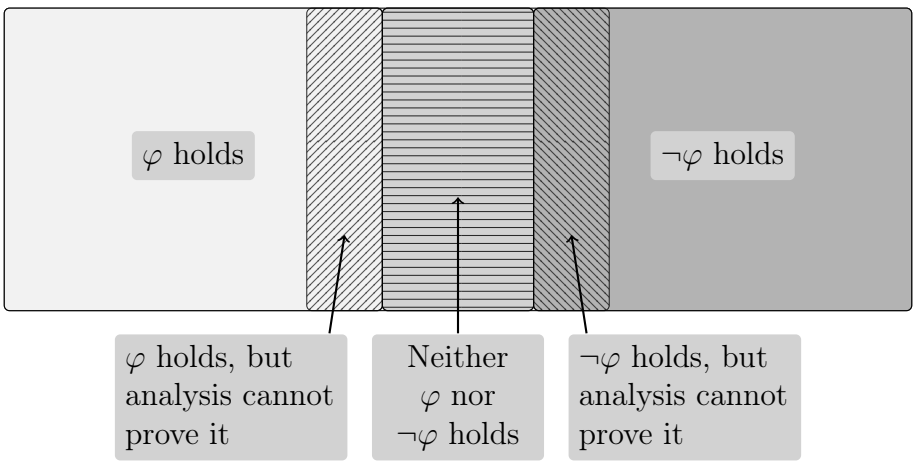
an abstract domain, we cannot always decide whether φ is satisfied by all reachable concrete states or not. The relation between $Coll_P(v)$ and $MFP_P(v)$ is as follows:

$$Coll_P(v) \subseteq \gamma(MOP_P(v)) \subseteq \gamma(MFP_P(v))$$

If we find out that $\gamma(MFP_P(v)) \subseteq \varphi$, we also know that the property holds in all concrete states: $Coll_P(v) \subseteq \varphi$. However, the converse does not hold $\gamma(MFP_P(v)) \not\subseteq \varphi$ does not imply $Coll_P(v) \not\subseteq \varphi$. To prove that φ does not hold we can check whether $\neg\varphi$ holds, i.e., whether $\gamma(MFP_P(v)) \subseteq \mathcal{D}_{conc} \setminus \varphi = \neg\varphi$. For some properties neither $\gamma(MFP_P(v)) \subseteq \varphi$ nor $\gamma(MFP_P(v)) \subseteq \mathcal{D}_{conc} \setminus \varphi$ hold and we do not know whether the property holds or not. This is the standard view of the situation:



In many application areas the goal is to either prove that a property is satisfied by all concrete states or to prove that it is satisfied by no concrete state at a particular program point. An example in our application area is that of classifying memory accesses associated with instructions in the program as cache hits or cache misses. One can distinguish three cases if the analysis can neither prove φ (classify the memory access as a cache hit) nor $\neg\varphi$ (classify it as a cache miss):



Either this is because neither φ nor $\neg\varphi$ holds or because of the imprecision of the analysis. In our cache example, some of the states in $Coll_P(v)$ result in a cache miss whereas others result in a cache hit: the memory access can neither be classified as a definite hit nor as a definite miss. Such uncertainty is inherent to the sticky collecting semantics.

Imprecision due to abstraction

Imprecision due to the abstraction can have two reasons:

1. The abstract domain is inadequate:
   - The domain should be able to express properties of interest: If one is for instance interested in whether a program variable is even or odd, an analysis that records the sign of the variable is inadequate. Even if one obtains precise information in this domain, one cannot decide whether a variable is even or odd.
   - The domain should be strong enough to maintain or infer interesting invariants: A cache domain that simply remembers whether a memory block is contained in the cache or not but does not maintain additional information to maintain such information will be weak. Upon a single memory access it will lose all information about its contents. Any of the memory blocks could be evicted by that access.
2. The abstract transformer is not precise. Provided a Galois connection between the concrete and the abstract domain, the best abstract transformer can be defined. However, sometimes it is not efficiently computable, and one has to settle with a sound but not the best abstract transformer.

Uncertainty inherent to the sticky collecting semantics

As an example of uncertainty inherent to the collecting semantics consider the following example program:

Example.

```
while (100/x > 5)
    x := x + 1;
```

If variable `x` can initially take any value, evaluating `100/x` may cause a division-by-zero exception. The collecting semantics at that program point reflects this, the predicate $\varphi := \texttt{x} \neq 0$ does not hold, neither does $\neg\varphi$. A compiler would have to introduce code that checks whether `x` is zero or not to generate safe code. However, from the second iteration of the loop on, evaluating `100/x` is safe, since `x` must be greater than zero. For all iterations but the first, the safety check is thus superfluous. Given this knowledge a compiler might decide to unroll the loop once to save the superfluous checks.

To distinguish the two cases in a preceding analysis, one needs to introduce context, i.e., information about how control reached this point. Formally, the *sticky collecting semantics* can be made context-sensitive in the following way: The *context-sensitive sticky collecting semantics* refines the *sticky collecting semantics* by mapping a program point *and* a context $c \in C$ to the set of concrete states that may arise at that point in the program under that context:

Definition 3.13 (Context-sensitive sticky collecting semantics)**.** *The* context-sensitive sticky collecting semantics $CSColl_P : V \times C \to \mathcal{D}_{coll}$ *of a program* P*, represented by control-flow graph* G_P*, on a set of concrete initial states* $Init \subseteq \mathcal{D}_{conc}$ *is*

$$CSColl_P(v, c) := \bigcup \{ [\![\pi]\!]_{coll}(Init) \mid \pi \text{ is a path from } i \text{ to } v \text{ in } G_P \text{ in context } c \}$$

Abstract context-sensitive semantics can be defined analogously.

In this particular example, virtually unrolling the loop once allows the analysis to determine that only the first safety check is necessary. As we will see in the following chapter, virtual loop unrolling is also important in cache analysis.

4
Cache Analysis

The goal of a cache analysis is to statically predict the cache behavior of a program on a set of inputs with a possibly unknown initial cache state. As the cache behavior may vary from input to input and from one initial state to another, it may not be possible to safely classify each memory access in the program as a hit or a miss. A cache analysis is therefore forced to approximate the cache behavior in a conservative way if it shall be used to provide guarantees on the execution time of a task.

To obtain tight bounds on the execution time of a program it is essential to use a precise cache analysis. Each excluded cache miss improves the provable upper bound on the worst-case execution time roughly by the cache miss penalty[1]. Conversely, each guaranteed cache miss improves the provable lower bound on the best-case execution time. Due to the high cache miss penalty, it is not an option to simply assume cache misses everywhere:

> At the Ada Deutschland conference, Alfred Roßkopf of EADS presented results showing that a PowerPC 604 running at 300 MHz with caches disabled [...] delivers a similar performance on some benchmarks as a Motorola 68020 running at 20 MHz, while the PowerPC outperforms the 68020 by a factor of 20 with caching enabled. [Langenbach et al., 2002]

By now, typical cache miss penalty are even larger than they were in 2002. A timing analysis without any cache analysis would thus yield extremely pessimistic guarantees for a processor with caches. Many task sets would be rendered unschedulable by the analysis, although, in reality they would be perfectly schedulable.

What should a cache analysis determine?

WCET and BCET analyses need a classification of individual memory accesses in the program as hits or misses. For most architectures it is not sufficient to determine upper or lower bounds on the number of misses, respectively, for the execution of the entire program. This is because caches interact with other microarchitectural features like pipelines. For instance, the cache miss penalty may overlap with pipeline stalls. To precisely take such effects into account, a timing analysis needs to know where and when

[1] Part of the penalty may overlap with pipeline stalls.

the cache misses happen. Cache analyses for BCET and WCET have been proposed in [Li et al., 1996, White et al., 1997, Ferdinand, 1997, Ferdinand et al., 1997, Ferdinand and Wilhelm, 1999, Ramaprasad and Mueller, 2005, Sen and Srikant, 2007].

Another area of application for cache analyses are compiler optimizations. Loops that operate on large arrays have a large impact on the execution time. There is great potential of performance improvement by changing the data layout of the arrays and by changing the structure of the loops accordingly. Popular optimizations include inter-/intra-array padding, blocking, and loop fusion. Cache analyses that provide tight bounds on the number of misses of loop nests can aid the compiler in choosing appropriate parameters for these optimizations. Work in this area has been described in [Ghosh et al., 1997, Ghosh et al., 1998, Fraguela et al., 1999, Ghosh et al., 1999, Bermudo et al., 2000, Chatterjee et al., 2001].

How do analyses for these two applications differ? Cache analyses used in compiler optimizations are not aimed at providing precise guarantees on the execution time. They solely provide upper bounds on the number of cache misses. This is a good enough indicator of the resulting execution time for compiler optimization purposes. In contrast, as explained earlier, WCET analyses usually need a classification of individual memory references in the program as hits or misses.

The focus of this work is on cache analysis for WCET and BCET analysis, i.e., analyses that classify individual memory accesses as cache hits or cache misses. In the following we will therefore focus on such analyses and the difficulties that arise in that area. However, some of our results in the following chapters also apply to cache analyses that compute global bounds on the number of cache hits and misses.

What are the constituents of a cache analysis?

In order to answer the question of whether a certain memory reference in the program will cause a hit or a miss, an analysis needs to answer the following subquestions:

- **Which memory block is being accessed?** For instruction fetches this question is easy to answer. For data accesses, however, the answer to this question is in general not computable. In addition, static memory references often result in accesses to different memory addresses dynamically. As an example, consider array accesses within loops. An analysis has to compute safe approximations of the set of memory addresses that are accessed by a memory reference. Such analyses are called *value* or *address analysis* and can be formalized by abstract interpretation. Abstract domains used for *address analysis* include interval domains [Ferdinand et al., 1999] and circular linear progressions [Sen and Srikant, 2007].

- **Is the accessed memory block contained in the cache?** Once the accessed memory address or some overapproximation of the possible memory addresses is calculated, it remains to determine whether this address or set of addresses is cached. Often, this part alone is called *cache analysis*. Our focus will be solely on this problem. To answer the question, the cache analysis needs to compute a safe

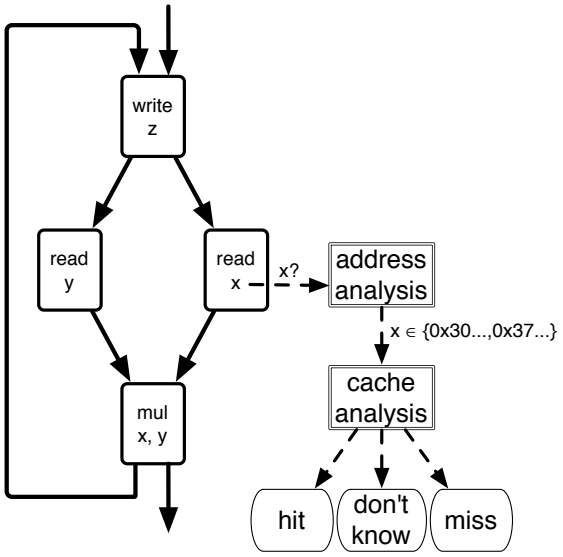


Figure 4.1: Classification of memory references.

approximation of the cache contents before the memory access, i.e., the "collecting cache semantics". This can then be queried to classify the access.

Figure 4.1 illustrates the two steps involved in classifying a memory reference as a hit or a miss. The results of the *address analysis* are used to compute a set of memory addresses that overapproximates the set of memory addresses that may be accessed by the reference. Then, the results of the *cache analysis* are used to classify the reference as a hit or a miss. Due to the inaccuracy of the analysis or the fact that the reference sometimes results in hits and sometimes in misses, i.e., uncertainty inherent to the collecting semantics, it is not always possible to make a definite classification. Hence, "don't know" is a possible answer.

Of course, the *address analysis* and *cache analysis* cannot answer these questions by looking at individual instructions in isolation. The cache contents at a program point depend on the cache contents and memory accesses at preceding program points. The same holds for variable values that are tracked by the address analysis. The whole program needs to be taken into account. Different *data-flow analyses* [Kildall, 1973] have been proposed for address [Ferdinand et al., 1999, Sen and Srikant, 2007] and cache analysis [White et al., 1997,Ferdinand, 1997,Ferdinand et al., 1997,Ferdinand and Wilhelm, 1999, Sen and Srikant, 2007]. Some [Ferdinand, 1997, Ferdinand et al., 1997, Ferdinand and Wilhelm, 1999,Sen and Srikant, 2007] are based on *abstract interpretation*.

Cache and address analyses cannot be performed on the source level; they *must* operate on executables. The precise data layout and scheduling of instructions is usually unknown at the source level. In that case, it is impossible to perform a cache analysis at this level. In order to still perform a static analysis over the control-flow graph of the program, it needs to be recovered from the binary [Theiling, 2003].

4.1 May and Must Information

In static analysis there is a concept of *may* and *must* information. In alias analysis, for instance, there are *may* and *must* aliases. Two access paths are in a *must* alias relation at a program point if they are guaranteed to alias at that point in the program. Two access paths are in a *may* alias relation at a program point if they cannot be guaranteed not to alias at that point in the program. Corresponding to *may* and *must* alias information, there are *may* and *must* alias analyses.

Analogously, there is a concept of *may* and *must* cache information in static cache analysis: *may* and *must* caches are upper and lower approximations, respectively, to the contents of all concrete caches that will occur whenever program execution reaches a program point. So, the *must* cache at a program point is a set of memory blocks that are definitely in each concrete cache at that point. The *may* cache is a set of memory blocks that may be in a concrete cache whenever program execution reaches that program point. In Section 4.2, we are presenting *may* and *must* analyses for LRU based on abstract interpretation.

Must cache information is used to derive safe information about cache hits; in other words it is used to exclude the "timing accident" cache miss. The complement of the *may* cache information is used to safely predict cache misses. The more cache hits can be predicted, the better the upper bound on the worst-case execution time will be. Vice versa, predicting more cache misses will result in a better lower bound on the best-case execution time. Observe the asymmetry between *may* and *must*: while a greater *must* cache means more precise information, a greater *may* cache means less precise information.

While some cache analyses explicitly maintain *may* and *must* cache information, others do so only implicitly. Whenever a cache analysis predicts a cache miss this prediction is based on *may* cache information, be it explicit or implicit. Similarly, a cache hit prediction is always based on *must* cache information.

4.2 Cache Analysis by Abstract Interpretation: Ferdinand's LRU Analysis

As an example of a state-of-the-art abstract interpretation-based cache analysis, we describe the LRU analysis of Christian Ferdinand [Ferdinand, 1997]. Later, in Chapter 6, we will show how to derive cache analyses for other policies, like FIFO and PLRU, from this or other LRU analyses. In the course of presenting the analysis we will try to point out properties of LRU that make such an analysis precise, and discuss challenges to cache analysis that are independent of the replacement policy.

For the sake of simplicity, we will restrict ourselves to fully-associative caches, although processor caches are usually set-associative. Set-associative caches can be seen as collec-

tions of many independent[2] fully-associative caches. Therefore, cache analyses for the fully-associative case trivially translate to analyses for set-associative caches.

Concrete Cache Semantics

In sections 2.2.1 and 2.2.9, we introduced cache domains and functions modeling the replacement policies' semantics. Recall that we denote the set of memory blocks by M. The set of access sequences $\langle b, b\rangle, \langle b, c, d\rangle, \ldots$ is denoted by $S = M^*$. We denote the set of reachable cache-set states of policy P by C^P. We use k-tuples of memory blocks to represent cache-sets of associativity k, i.e., $C^{\mathrm{LRU}(k)} \subseteq M_\perp^k$. For LRU, memory blocks are ordered from most- to least-recently-used from left to right. In $[b, e, d, f]_{\mathrm{LRU}}$, b is the most-recently-used element and f the least-recently-used one. The function $update_P(q, s) : C^P \times S \to C^P$ computes the cache-set state after accessing a sequence s in state q under policy P. For instance, $update_{\mathrm{LRU}}([b, e, d, f]_{\mathrm{LRU}}, \langle f, c\rangle) = [c, f, b, e]_{\mathrm{LRU}}$. To define the concrete transformer $f^{\mathrm{LRU}} : V \to C^{\mathrm{LRU}} \to C^{\mathrm{LRU}}$, which computes the effect of program statements in V on the state of the cache, we need a function $\iota : V \to S$ that maps a program statement $v \in V$ to the sequence of memory blocks $s \in S = M^*$ that is accessed by v. In the case of an instruction cache analysis, ι is easily obtained. For data caches, the memory blocks that are being accessed depend on the state of the system. We only consider instruction caches here, as we do not want to consider the *address analysis* problem. Given ι, the concrete transformer is defined as follows:

$$f^{\mathrm{LRU}}(v)([a_1, \ldots, a_k]_{\mathrm{LRU}}) := update_{\mathrm{LRU}}([a_1, \ldots, a_k]_{\mathrm{LRU}}, \iota(v))$$

The concrete transformer can be lifted to a collecting transformer f_{coll}^{LRU} as described in Chapter 3. f_{coll}^{LRU} induces a *collecting path semantics* and finally the *sticky collecting semantics*, which can be queried to classify memory accesses as cache hits or cache misses.

Abstract Cache Semantics

The *sticky collecting semantics* is not computable. It has to be approximated by the maximal fixed point solution in an abstract domain as described in Chapter 3.

Christian Ferdinand introduces two analyses for LRU:

- A *may* analysis, which computes a set of memory blocks that *may* be in the cache at a program point. *May* analysis results can be used to predict cache misses, namely for those memory blocks that *may not* be in the cache.
- A *must* analysis, which computes a set of memory blocks that *must* be in the cache at a program point. *Must* analysis results can be used to predict cache hits.

[2] All of the policies presented in this dissertation except PSEUDO-RR treat each cache set independently.

Recall the *notion of age* introduced in Section 2.2.3. Both analyses approximate the age of memory blocks: The *may* analysis approximates ages from below, while the *must* analysis approximates the memory blocks' ages from above.

Both abstract domains $\mathcal{A}^{\text{LRU}}_{must}$, $\mathcal{A}^{\text{LRU}}_{may}$ can be modeled as function spaces in which memory blocks are mapped to approximations of their ages:

$$\mathcal{A}^{\text{LRU}}_{must} = \mathcal{A}^{\text{LRU}}_{may} = M \to \{0, \ldots, k-1, \infty\}$$

Abstract states can be compactly represented by only explicitly storing a memory block's age if it is not ∞. In the *must* analysis at most k elements satisfy this constraint at any point in time. In the *may* analysis more complicated representations are required for efficiency, which we do not want to elaborate on here. To compactly represent abstract states textually, we rely on the isomorphic representation by k-tuples of sets of memory blocks, as in $\widehat{p} = [\{a, b\}, \{f\}, \{d\}, \{e, g\}]$, where $k = 4$. Memory blocks that map to i in the function representation are contained in the $(i+1)$th component of the tuple. Memory blocks that map to ∞ are not contained in any of the tuple's sets. In our example state, $\widehat{p}(a) = \widehat{p}(b) = 0, \widehat{p}(f) = 1, \widehat{p}(d) = 2, \widehat{p}(e) = \widehat{p}(g) = 3$, and $\widehat{p}(x) = \infty$ for all other memory blocks.

While the domains are the same for *may* and *must*, the interpretations of abstract states are different. To concisely define both concretization functions $\gamma^{\text{LRU}}_{must} : \mathcal{A}^{\text{LRU}}_{must} \to \mathcal{P}(C^{\text{LRU}})$ and $\gamma^{\text{LRU}}_{may} : \mathcal{A}^{\text{LRU}}_{may} \to \mathcal{P}(C^{\text{LRU}})$, we need to define the helper function $age : C^{\text{LRU}} \times M \to \mathbb{N}$ that computes the age of a memory block in a cache set:

$$age([a_0, \ldots, a_{k-1}], m) := \begin{cases} i & \text{if } a_i = m \\ \infty & \text{otherwise} \end{cases}$$

Given age, $\gamma^{\text{LRU}}_{must}$ is easily defined:

$$\gamma^{\text{LRU}}_{must}(\widehat{q}) = \{q \in C^{\text{LRU}} \mid \forall m \in M : age(q, m) \leq \widehat{q}(m)\},$$

where $0 < 1 < \ldots < k-1 < \infty$. I.e., a concrete state c is represented by an abstract state $\widehat{q}$ if memory blocks are *at most* as "old" in c as they are in $\widehat{q}$. In particular, if $\widehat{q}(b) \leq k-1$, b *must* be contained in a concretization of $\widehat{q}$. In the *may* domain it is exactly the other way around:

$$\gamma^{\text{LRU}}_{may}(\widehat{q}) = \{q \in C^{\text{LRU}} \mid \forall m \in M : \widehat{q}(m) \leq age(q, m)\},$$

i.e., a concrete state c is represented by an abstract state $\widehat{q}$ if memory blocks are *at least* as "old" in c as they are in $\widehat{q}$. In particular, if $\widehat{q}(b) = \infty$, b *may not* be contained in any of the concretizations of $\widehat{q}$.

Example (Concretization of abstract states)**.**
Consider the following three abstract states $\widehat{p}, \widehat{q}, \widehat{s} \in \mathcal{A}^{\text{LRU}}_{must}$*:*

$$\begin{aligned} \widehat{p} &= [\{b\}, \{\}, \{a, c\}, \{f\}] \in \mathcal{A}^{\text{LRU}}_{must} \\ \widehat{q} &= [\{\}, \{d\}, \{b, c\}, \{a\}] \in \mathcal{A}^{\text{LRU}}_{must} \\ \widehat{s} &= [\{\}, \{d, c\}, \{\}, \{b\}] \in \mathcal{A}^{\text{LRU}}_{must} \end{aligned}$$

In $\widehat{p}$ and $\widehat{q}$, the contents of the cache sets are fully determined. Due to the uncertainty about the precise ages of their contents, a finite set of concrete states is represented by $\widehat{p}$ and $\widehat{q}$:

$$\begin{aligned}\gamma^{\mathrm{LRU}}_{must}(\widehat{p}) &= \{[b,a,c,f],[b,c,a,f]\}\\ \gamma^{\mathrm{LRU}}_{must}(\widehat{q}) &= \{[d,b,c,a],[d,c,b,a],[b,d,c,a],[c,d,b,a]\}\end{aligned}$$

In $\widehat{s}$, only three of the four memory blocks that can be stored in a cache set are known. Therefore, any other memory block might occupy the remaining cache line:

$$\gamma^{\mathrm{LRU}}_{must}(\widehat{s}) = \{[d,c,b,x],[d,c,x,b],[c,d,b,x],[c,d,x,b] \mid x \in M_{\perp}\backslash\{b,c,d\}\}$$

In $\mathcal{A}^{\mathrm{LRU}}_{may}$, ages of memory blocks are bound from below. Consider the following two abstract states $\widehat{t},\widehat{u} \in \mathcal{A}^{\mathrm{LRU}}_{may}$:

$$\begin{aligned}\widehat{t} &= [\{\},\{\},\{\},\{a\}] \in \mathcal{A}^{\mathrm{LRU}}_{may}\\ \widehat{u} &= [\{\},\{\},\{a\},\{b\}] \in \mathcal{A}^{\mathrm{LRU}}_{may}\end{aligned}$$

In a cache analysis $\widehat{t}$ and $\widehat{u}$ are not reachable. However, we chose these two abstract states as they have very small concretizations that are easily understandable. The concretization of $\widehat{t}$ contains only two elements:

$$\gamma^{\mathrm{LRU}}_{may}(\widehat{t}) = \{[\perp,\perp,\perp,\perp],[\perp,\perp,\perp,a]\}$$

The empty concrete state $[\perp,\perp,\perp,\perp]$ is contained in the concretization of all abstract states. $\widehat{u}$'s concretization subsumes $\widehat{t}$'s concretization:

$$\gamma^{\mathrm{LRU}}_{may}(\widehat{u}) = \{[\perp,\perp,\perp,\perp],[\perp,\perp,\perp,a],[\perp,\perp,\perp,b],[\perp,\perp,a,\perp],[\perp,\perp,a,b]\}.$$

Abstraction functions compute the best abstract description of a set of concrete states in an abstract domain. Using *age* they can also be easily defined:

$$\begin{aligned}\alpha^{\mathrm{LRU}}_{must} &: \mathcal{P}(C^{\mathrm{LRU}}) \to \mathcal{A}^{\mathrm{LRU}}_{must}\\ \alpha^{\mathrm{LRU}}_{must}(C) &= \lambda m.\max_{c\in C} age(c,m)\\ \\ \alpha^{\mathrm{LRU}}_{may} &: \mathcal{P}(C^{\mathrm{LRU}}) \to \mathcal{A}^{\mathrm{LRU}}_{may}\\ \alpha^{\mathrm{LRU}}_{may}(C) &= \lambda m.\min_{c\in C} age(c,m)\end{aligned}$$

With the following orders $\sqsubseteq^{\mathrm{LRU}}_{must}$ and $\sqsubseteq^{\mathrm{LRU}}_{may}$, $(\mathcal{P}(C^{\mathrm{LRU}}),\subseteq) \underset{\alpha^{\mathrm{LRU}}_{must}}{\overset{\gamma^{\mathrm{LRU}}_{must}}{\leftrightarrows}} (\mathcal{A}^{\mathrm{LRU}}_{must},\sqsubseteq^{\mathrm{LRU}}_{must})$ and $(\mathcal{P}(C^{\mathrm{LRU}}),\subseteq) \underset{\alpha^{\mathrm{LRU}}_{may}}{\overset{\gamma^{\mathrm{LRU}}_{may}}{\leftrightarrows}} (\mathcal{A}^{\mathrm{LRU}}_{may},\sqsubseteq^{\mathrm{LRU}}_{may})$ form *Galois insertions*. See [Ferdinand, 1997] for a proof.

The orders $\sqsubseteq^{\mathrm{LRU}}_{must}$ and $\sqsubseteq^{\mathrm{LRU}}_{may}$ are defined such that $\gamma^{\mathrm{LRU}}_{must}$ and $\gamma^{\mathrm{LRU}}_{may}$ are monotone:

$$\begin{aligned}\widehat{p} \sqsubseteq^{\mathrm{LRU}}_{must} \widehat{q} &:\Leftrightarrow \gamma^{\mathrm{LRU}}_{must}(\widehat{p}) \subseteq \gamma^{\mathrm{LRU}}_{must}(\widehat{q})\\ \widehat{p} \sqsubseteq^{\mathrm{LRU}}_{may} \widehat{q} &:\Leftrightarrow \gamma^{\mathrm{LRU}}_{may}(\widehat{p}) \subseteq \gamma^{\mathrm{LRU}}_{may}(\widehat{q})\end{aligned}$$

$\sqsubseteq^{\mathrm{LRU}}_{must}$ and $\sqsubseteq^{\mathrm{LRU}}_{may}$ can be easily expressed without referring to the concretization functions:

$$\begin{aligned}\widehat{p} \sqsubseteq^{\mathrm{LRU}}_{must} \widehat{q} &:\Leftrightarrow \forall m \in M : \widehat{p}(m) \leq \widehat{q}(m)\\ \widehat{p} \sqsubseteq^{\mathrm{LRU}}_{may} \widehat{q} &:\Leftrightarrow \forall m \in M : \widehat{p}(m) \geq \widehat{q}(m)\end{aligned}$$

Therefore, in our example on concretizations, $\widehat{t} \sqsubseteq^{\mathrm{LRU}}_{may} \widehat{u}$ and $\gamma^{\mathrm{LRU}}_{may}(\widehat{t}) \subseteq \gamma^{\mathrm{LRU}}_{may}(\widehat{u})$. Both $\mathcal{A}^{\mathrm{LRU}}_{must}$ and $\mathcal{A}^{\mathrm{LRU}}_{may}$ satisfy the *ascending chain condition* [Ferdinand, 1997].
The partial orders $(\mathcal{A}^{\mathrm{LRU}}_{must}, \sqsubseteq^{\mathrm{LRU}}_{must})$ and $(\mathcal{A}^{\mathrm{LRU}}_{may}, \sqsubseteq^{\mathrm{LRU}}_{may})$ are join-semilattices so that least upper bounds always exist. The binary join operators $\sqcup^{\mathrm{LRU}}_{must}$ and $\sqcup^{\mathrm{LRU}}_{may}$ induced by $\sqsubseteq^{\mathrm{LRU}}_{must}$ and $\sqsubseteq^{\mathrm{LRU}}_{may}$ are:

$$\begin{aligned}\widehat{p} \sqcup^{\mathrm{LRU}}_{must} \widehat{q} = \widehat{s}, \quad \text{where } \forall m \in M : \quad \widehat{s}(m) = max\{\widehat{p}(m), \widehat{q}(m)\},\\ \widehat{p} \sqcup^{\mathrm{LRU}}_{may} \widehat{q} = \widehat{s}, \quad \text{where } \forall m \in M : \quad \widehat{s}(m) = min\{\widehat{p}(m), \widehat{q}(m)\},\end{aligned}$$

where *max* and *min* are based on the order given before: $0 < 1 < \ldots < k-1 < \infty$.

The two semilattices are no complete lattices, because they lack least elements. However, they can be augmented with least elements $\bot_{must}$ and $\bot_{may}$, such that $\forall \widehat{q} \in \mathcal{A}^{\mathrm{LRU}}_{must} : \bot_{must} \sqsubseteq^{\mathrm{LRU}}_{must} \widehat{q}$ and $\forall \widehat{q} \in \mathcal{A}^{\mathrm{LRU}}_{may} : \bot_{may} \sqsubseteq^{\mathrm{LRU}}_{may} \widehat{q}$, to form complete lattices.
In the *must* analysis, the abstract states maintain upper bounds on the ages of memory blocks. To maintain this invariant, the maximum of the ages has to be taken at joins. If an element might not be contained in one of the two states that are being joined, i.e., $\widehat{p}(m) = \infty$, it might not be contained in the join either. Therefore $\widehat{s}(m) = \infty$. Conversely, in the *may* analysis, the abstract states maintain lower bounds on the ages of memory blocks. Therefore, the minimum of the ages has to be taken at joins.

Example (Loss of information at joins)**.** *In the computation of the MFP-solution, abstract states have to be joined where control-flow merges. Often, this incurs a loss of information as the abstract domain is not strong enough to precisely represent the join of the states that are to be merged, i.e.,* $\gamma(\widehat{x}) \cup \gamma(\widehat{y}) \subsetneq \gamma(\widehat{x} \sqcup \widehat{y})$*. Consider* $\widehat{p}, \widehat{q} \in \mathcal{A}^{\mathrm{LRU}}_{must}$ *as in the previous example.*

$$\begin{aligned}\widehat{p} \sqcup^{\mathrm{LRU}}_{must} \widehat{q} = \widehat{pq} &= [\{b\}, \{\}, \{a, c\}, \{f\}] \sqcup^{\mathrm{LRU}}_{must} [\{\}, \{d\}, \{b, c\}, \{a\}]\\ &= [\{\}, \{\}, \{b, c\}, \{a\}]\end{aligned}$$

We lose the information that either f *or* d *must be contained in the cache by joining the two abstract states.*

$$\begin{aligned}\gamma^{\mathrm{LRU}}_{must}(\widehat{pq}) = \{&[b, c, a, x], [b, c, x, a], [c, b, a, x], [c, b, x, a],\\ &[b, a, c, x], [c, a, b, x], [b, x, c, a], [c, x, b, a],\\ &[a, b, c, x], [a, c, b, x], [x, b, c, a], [x, c, b, a] \mid x \in M_{\bot} \backslash \{a, b, c\}\}\end{aligned}$$

Together, the concretizations of $\widehat{p}$ *and* $\widehat{q}$ *contain* 6 *states. The concretization of* $\widehat{pq}$ *contains* $12 \cdot (|M_\perp| - 3)$ *states.*
Joining the may *states* $\widehat{t}$ *and* $\widehat{u}$ *of our previous example yields* $\widehat{u} = \widehat{t} \sqcup^{\mathrm{LRU}}_{may} \widehat{u}$ *as* $\widehat{t} \sqsubseteq^{\mathrm{LRU}}_{may} \widehat{u}$.

In designing abstract domains there is a tradeoff between precision and efficiency. Increasing the size of the domain may reduce the information loss through joins. There are two extremal abstract domains:

- The power set of the concrete states. Joins simply become set unions and no loss of information is incurred. Due to its size and the height of its chains, analyses with such a domain are usually infeasible.
- A flat domain in which abstract states are either concrete states or $\top$. Whenever, two different states are joined, all information is lost. Due to the limited height of the lattice, namely 1, analyses on this domain are very efficient. However, they are usually also very imprecise.

The challenge is thus to find a domain somewhere in between the two extremes, that represents a good tradeoff between the two conflicting goals of precision and efficiency. For LRU, the presented domain has proved to be a good compromise.
It remains to define the abstract transformers $f^{\mathrm{LRU}}_{must} : V \to \mathcal{A}^{\mathrm{LRU}}_{must} \to \mathcal{A}^{\mathrm{LRU}}_{must}$ and $f^{\mathrm{LRU}}_{may} : V \to \mathcal{A}^{\mathrm{LRU}}_{may} \to \mathcal{A}^{\mathrm{LRU}}_{may}$. As in the concrete case, we define these transformers in terms of update functions that take sequences of memory accesses as arguments:

$$\begin{aligned} f^{\mathrm{LRU}}_{must}(v)(\widehat{q}) &:= update^{\mathrm{LRU}}_{must}(\widehat{q}, \iota(v)), \\ f^{\mathrm{LRU}}_{may}(v)(\widehat{q}) &:= update^{\mathrm{LRU}}_{may}(\widehat{q}, \iota(v)), \end{aligned}$$

where, as in the case of the concrete transformer, ι maps a program statement $v \in V$ to the memory blocks $s \in S = M^*$ that are accessed by v. Again, as in the concrete case, update functions on sequences of memory accesses can be defined in terms of update functions on individual memory accesses:

$$\begin{aligned} update^{\mathrm{LRU}}_{must}(\widehat{q}, ()) &:= \widehat{q} \\ update^{\mathrm{LRU}}_{must}(\widehat{q}, (b_1, \ldots, b_n)) &:= update^{\mathrm{LRU}}_{must}(update^{\mathrm{LRU}}_{must}(\widehat{q}, b_1), (b_2, \ldots, b_n)) \end{aligned}$$

Similarly for $update^{\mathrm{LRU}}_{may}$. Finally, $update^{\mathrm{LRU}}_{must} : \mathcal{A}^{\mathrm{LRU}}_{must} \times M \to \mathcal{A}^{\mathrm{LRU}}_{must}$ and $update^{\mathrm{LRU}}_{may} : \mathcal{A}^{\mathrm{LRU}}_{may} \times M \to \mathcal{A}^{\mathrm{LRU}}_{may}$, the *best* abstract updates, are defined as follows:

$$update^{\mathrm{LRU}}_{must}(\widehat{q}, a) := \lambda b. \begin{cases} 0 & \text{if } b = a & (1) \\ \widehat{q}(b) & \text{if } \widehat{q}(a) \leq \widehat{q}(b) & (2) \\ \widehat{q}(b) + 1 & \text{if } \widehat{q}(a) > \widehat{q}(b) \wedge \widehat{q}(b) < k - 1 & (3) \\ \infty & \text{if } \widehat{q}(a) > \widehat{q}(b) \wedge \widehat{q}(b) \geq k - 1 & (4) \end{cases}$$

Case (1): The memory block a that is accessed assumes age 0 independently of its previous position. This is a nice property of LRU that allows to regain information

about cache states quickly. In other policies, it often depends on whether the access is a hit or a miss, which might be unknown to the analysis, how the position of the element is updated.

Case (2): If $\widehat{q}(a) \leq \widehat{q}(b)$, there are two possibilities in $\widehat{q}$'s concretization:

1. a is younger than b. Then b's age is not affected by the memory access and $\widehat{q}(b)$ remains a sound approximation of its age.
2. b is younger than a. Then, b must be younger than $\widehat{q}(a)$ and thus also younger than $\widehat{q}(b)$. Accessing a ages b but not beyond $\widehat{q}(b)$.

Cases (3+4): If $\widehat{q}(a) > \widehat{q}(b)$, then there is a state in the concretization of $\widehat{q}$ such that a's age is greater than b's age and b's age is $\widehat{q}(b)$. In that state, accessing a ages b by 1. This evicts b from the cache if its previous age was $k-1$ (Case 4). For a formal proof of the *local consistency* of the abstract updates consult [Ferdinand, 1997].

Example (LRU-*must*-update)**.** *Assume abstract state* $\widehat{p} \in \mathcal{A}^{\mathrm{LRU}}_{must}$ *as before:*

$$\widehat{p} \;=\; [\{b\},\{\},\{a,c\},\{f\}] \in \mathcal{A}^{\mathrm{LRU}}_{must}$$

Its concretization is $\{[b,a,c,f],[b,c,a,f]\}$*. In both states accessing* c *yields* $[c,b,a,f]$ *whose abstraction is* $[\{c\},\{b\},\{a\},\{f\}]$*. Since* $update^{\mathrm{LRU}}_{must}$ *is the best abstract update,*

$$update^{\mathrm{LRU}}_{must}(\widehat{p},c) \;=\; [\{c\},\{b\},\{a\},\{f\}] \in \mathcal{A}^{\mathrm{LRU}}_{must}.$$

The most interesting part of the abstract update is the handling of a*. Its update follows case (2) which can be explained by the case distinction discussed above.*

The ability to quickly eliminate uncertainty about cache states, as in the previous example, is a nice property of Ferdinand's LRU analysis. Other replacement policies do *not* allow quick recovery from information loss in *any* abstract domain. We will study this in depth in Chapter 5.

The update of *may* states is slightly different:

$$update^{\mathrm{LRU}}_{may}(\widehat{q},a) := \lambda b.\begin{cases} 0 & \text{if } b = a & (1)\\ \widehat{q}(b) & \text{if } \widehat{q}(a) < \widehat{q}(b) & (2)\\ \widehat{q}(b)+1 & \text{if } \widehat{q}(a) \geq \widehat{q}(b) \wedge \widehat{q}(b) < k-1 & (3)\\ \infty & \text{if } \widehat{q}(a) \geq \widehat{q}(b) \wedge \widehat{q}(b) \geq k-1 & (4)\end{cases}$$

Case (1): The memory block a that is accessed assumes age 0 independently of its previous position.

Case (2): If $\widehat{q}(a) < \widehat{q}(b)$, then b's age may be $\widehat{q}(b)$ and a may be younger. In that case, accessing a does not age b and $\widehat{q}(b)$ may not be incremented.

Case (3): If $\widehat{q}(a) \geq \widehat{q}(b)$, there are two possibilities in $\widehat{q}$'s concretization:

1. a is younger than b. Then b does not age by accessing a, but its age must have already been greater than $\widehat{q}(b)$.
2. b is younger than a. Then b ages by accessing a and $\widehat{q}(b)+1$ is a valid lower bound on its new age.

Case (4): As in case (3). However, increasing a's age evicts it from the *may* state.

Example (LRU-*may* update)**.** *Assume abstract state*

$$\widehat{v} \;=\; [\{\},\{\},\{c\},\{a,b,d\}] \in \mathcal{A}^{\mathrm{LRU}}_{may}.$$

Accessing d in this state yields

$$update^{\mathrm{LRU}}_{may}(\widehat{v},d) \;=\; [\{d\},\{\},\{\},\{c\}] \in \mathcal{A}^{\mathrm{LRU}}_{may}.$$

The updates of a and b follow case (4) *which can be explained by the case distinction discussed above. c is updated according to case* (2)*.*

Since the two lattices $\mathcal{A}^{\mathrm{LRU}}_{must}$ and $\mathcal{A}^{\mathrm{LRU}}_{may}$ satisfy the ascending chain condition, the *MFP* solution can be computed by Kleene's fixed point iteration. It safely approximates the *MOP* solution and – due to the correctness of the update functions – transitively the *sticky collecting semantics*. It has been an open question whether the analysis is distributive. In Appendix B, we give two small examples that demonstrate its non-distributivity.

Example Analysis

To illustrate the LRU-*must*-analysis consider the following simple example program:

```
A:   while (x < 10)
     {
B:     if (x % 2 == 0)
C:       y++;
       else
D:       y--;
E:     x++;
     }
```

For simplicity of exposition each of the five instructions at program points A, B, C, D, E is stored in its own separate memory block, also denoted A, B, C, D, or E, respectively. Furthermore, we assume a fully-associative cache with five cache lines; so all instructions map to the same single cache set.

Figure 4.2 shows the control-flow-graph of the program and the *must* information associated with each node the graph. At program entry we have no knowledge of the state of the cache. Therefore, we conservatively start with the $\top$-element of the lattice, $[\{\},\{\},\{\},\{\},\{\}]$ at entry.

In this example, the LRU-*must*-analysis infers as much *must* information for each program point as is possible, i.e., the abstract cache state associated with each program point is precisely the abstraction of the collecting semantics. However, no cache hits can be predicted using the analysis results: In the first iteration of the loop, all memory

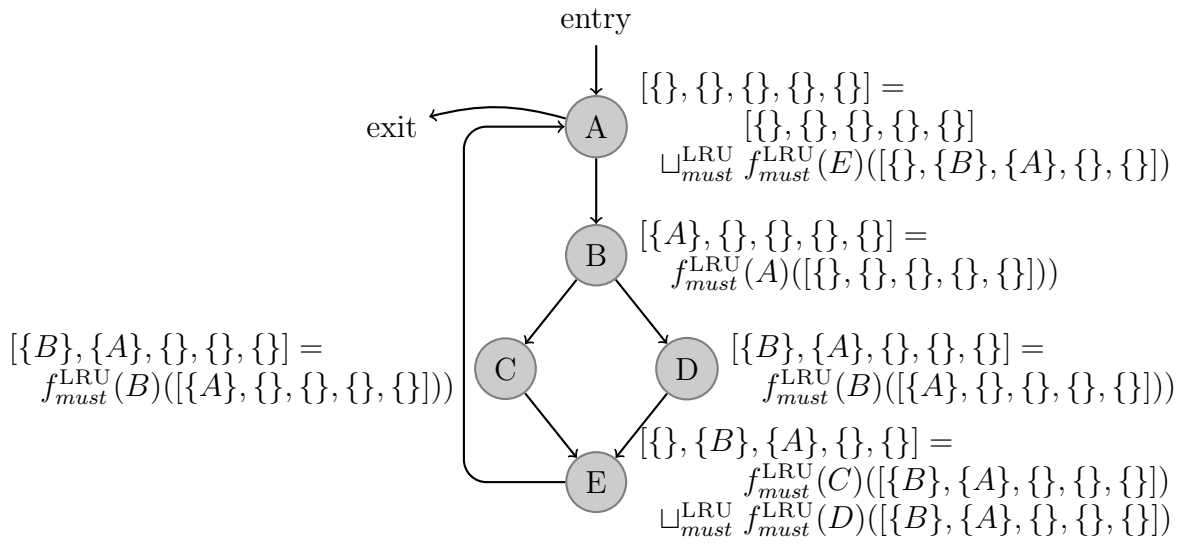


Figure 4.2: Analysis results on example program.

accesses might miss the cache. This is an example of *uncertainty inherent to the collecting semantics* as discussed in Section 3.4. As discussed there, one needs to introduce context, i.e., information about how control reached a program point, to overcome this problem. In this particular example, virtually unrolling the loop once, i.e., distinguishing the first iteration from subsequent iterations, allows the analysis to determine that only the first memory accesses to A, B, and E might miss the cache.

Figure 4.3 illustrates the effect of virtually unrolling the loop once. Nodes with thick borders reflect that the associated instruction access can be predicted to be a cache hit.

Starting with the second iteration of the loop, three of the four instruction accesses of an iteration are predicted to hit the cache. On the other hand, the analysis is still not able to predict hits for accesses to C and D. In any concrete execution of the loop, only the first accesses to C and D will result in misses. By distinguishing the first executions of C and D from following executions, the analysis could thus become even more precise. However, there is a price to pay: increasing the context-sensitivity of the analysis also increases its runtime and space consumption.

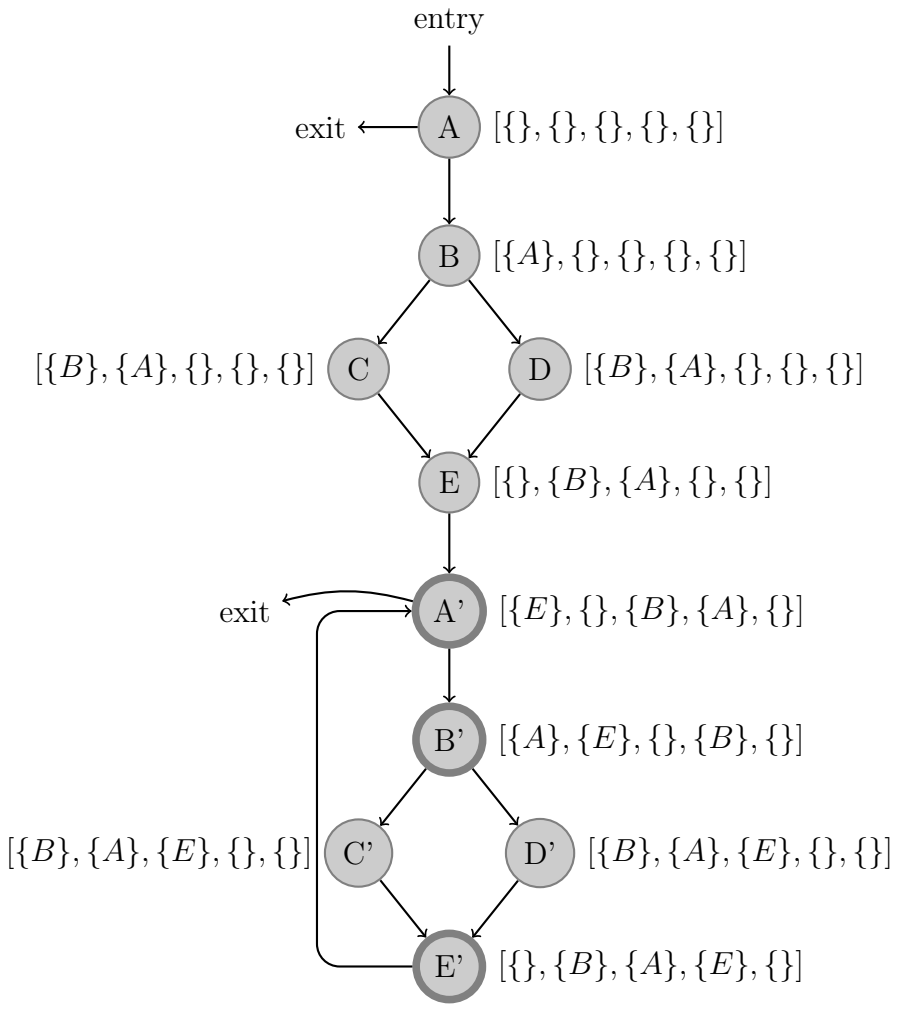


Figure 4.3: Analysis results after virtually unrolling the loop once.

4.3 Other Approaches

There have been a number of other approaches to cache analysis. As noted in the beginning of this chapter, one can distinguish between two types of cache analyses: Cache analyses directed at WCET analysis classify individual references in the program as hits or misses, whereas cache analyses directed at compiler optimizations compute bounds on the number of misses for larger program fragments, like loop nests. The following two sections describe work in the two areas, respectively.

4.3.1 Local Classification

The work closest to Ferdinand's is that of Mueller et al. [Mueller et al., 1994, White et al., 1997]. [Mueller et al., 1994] presents a *static cache simulation* for direct-mapped instruction caches. It classifies instructions as *always-miss*, *always-hit*, *first-miss*, or *conflict*. *Always-hit*, *always-miss*, and *conflict* correspond to the classifications of Ferdinand's analysis. *First-miss* expresses that the first instruction fetch may miss but all further accesses will be hits. This is a typical situation in the execution of loops, where only the first iteration results in cache misses. Similar classifications are obtained in Ferdinand's analysis by virtual loop unrolling. Mueller et al. obtain these classifications by data-flow analyses that are not semantics-based. In subsequent work [White et al., 1997], the analysis is extended to data caches, where the main challenges lie in the *address analysis*. To address shortcomings of the classification of individual references the analysis also computes upper bounds on the number of misses in loops. Such a classification is, however, difficult to use in most timing analyses. Furthermore, an instruction cache analysis for set-associative caches with LRU replacement is sketched. As the direct-mapped analysis, it is not semantics-based and its correctness is difficult to judge.

In [Li et al., 1996], Li, Malik, and Wolfe present an extension of their work on timing analysis using ILP formulations, that can handle set-associative caches, data caches, and unified caches. This work integrates pipeline, cache, and path analysis using an ILP formulation. To address set-associative caches, their concrete semantics is encoded using linear constraints. In theory this approach can handle any possible replacement policy. However, due to the ILP formulation that encodes the concrete cache behavior and all other timing-relevant aspects of a system, this approach suffers from severe complexity problems. In practice it is limited to direct-mapped caches and simple pipelines.

In [Heckmann et al., 2003], different threats to timing predictability are discussed, among others the use of non-LRU replacement policies. The paper introduces a *must* analysis for PLRU based on Ferdinand's *must* analysis for LRU. This analysis can be concisely justified by the competitiveness of PLRU relative to LRU (see Chapter 6).

4.3.2 Global Bounds

Ghosh, Martonosi, and Malik [Ghosh et al., 1997] introduce *Cache Miss Equations* (CMEs). The data cache behavior of loop nests is characterized by Diophantine equations. Each solution of these equations corresponds to a possible cache miss. The equations provide a framework to guide code optimizations for improving cache performance. Examples include the determination of array padding and offset amounts, and optimal blocking factors for tiled code. The approach is more precise than previous heuristics and potentially faster than simulation. It only applies to perfectly nested loops such that array references are contained within the innermost loops. Conditional statements within loops are not supported. In subsequent work [Ghosh et al., 1998, Ghosh et al., 1999], the approach is generalized to set-associative caches with LRU replacement and new algorithms to solve CMEs are introduced that allow to trade precision off against analysis complexity.

A new strategy to develop probabilistic analytical models of the cache behavior is described in [Fraguela et al., 1999]. Again, this work applies to set-associative caches with LRU replacement and perfectly nested loops. Non-perfectly nested loops may also be analyzed but at a possible loss of precision. Due to the probabilistic nature of the approach no guarantees on the cache behavior can be derived.

An exact model of the cache behavior of loops nests is developed in [Chatterjee et al., 2001] by using a non-standard classification of misses and by taking into account the state of the cache at the end of a loop nest. It can handle imperfect loop nests by converting them to perfect loop nests and various flavors of non-linear array layouts. Modest levels of associativity with LRU replacement can be handled. In contrast to previous approaches, the complexity solely relates to the static structure of the loop nests. In particular it is independent of the loop bounds. Furthermore, exact numbers of misses are determined instead of upper bounds as in the CME approach. Misses are classified in a non-standard fashion into *interior misses*, misses that occur independently of the context of the program fragment, and *potential boundary misses*, data references that may either hit or miss, depending on the initial cache state. Due to the way potential boundary misses are handled, the approach cannot be easily extended to other policies.

The approach of [Ramaprasad and Mueller, 2005] is directed at WCET analysis and is based on the CMEs approach [Ghosh et al., 1999]. In the context of WCET analysis, it improves on the original approach in two ways: Loops are virtually fused to enable the analysis of whole programs. To facilitate WCET analysis, the precise location of cache misses is extracted. For each memory reference that cannot be classified by CMEs to always hit or always miss additional analysis is performed by considering all competing references and their positions in the iteration space. Being based on CMEs it is limited to LRU.

4.4 Challenges and Outlook

In the design of a static cache analysis for a particular replacement policy, one aims to find an *efficient* and *precise* abstract domain, such that:

- The abstract cache states *precisely* and *compactly* represent sets of concrete cache states.
- The best abstract transformer is *cheaply computable* and *quickly eliminates uncertainty* from abstract states.
- The join operator does *not lose much information*.

Obviously, there is a tradeoff between precision and efficiency in all three points. A more precise domain usually comes at the precise of a loss of efficiency.

In the following three chapters, we will present three contributions in the context of cache analysis:

1. Even if one is willing to give up efficiency, there are limits to the attainable precision that are inherent to the underlying replacement policy. In Chapter 5, we introduce *predictability metrics* that capture the influence of the replacement policy on attainable precision.
2. It has proven difficult to develop *efficient* cache analyses for policies other than LRU. For instance, no *may* analysis for FIFO or MRU has been known previously. Except for [Heckmann et al., 2003], all approaches in literature either deal with direct-mapped caches or set-associative ones with LRU replacement. In Chapter 6, we introduce the concept of *relative competitiveness*, which allows to derive sound cache analyses for a policy A from existing analyses for another policy B. One result of studying *relative competitiveness* is that the LRU-*may*-analysis presented in this chapter can be used as a *may* analysis for smaller FIFO- and MRU-controlled caches. Other results allow to translate global bounds on the number of hits or misses to bounds for other policies.
3. Due to the large effort involved in devising sound and precise abstract domains for WCET analysis, recently, the use of *measurement* has been advocated. In Chapter 7, we study the influence of the initial cache state on the cache performance. Our analysis results suggest that measurement-based timing analysis *may* yield BCET and WCET estimates that are dramatically wrong.

5

Predictability Metrics – Limits on the Precision of Static Cache Analyses

This chapter presents results about the predictability of common cache replacement policies and mainly stems from [Reineke et al., 2007]. We introduce three metrics, *evict*, $fill$, and mls that capture aspects of cache-state predictability. A thorough analysis of the LRU, FIFO, MRU, and PLRU policies yields the respective values under these metrics. To the best of our knowledge, this work presents the first quantitative, analytical results for the predictability of replacement policies. Our results support empirical evidence in static cache analysis.

5.1 Introduction

Embedded systems as they occur in application domains such as automotive, aeronautics, and industrial automation often have to satisfy hard real-time constraints. Timeliness of reactions is absolutely necessary. Offline guarantees on the worst-case execution time of each task have to be derived using safe methods. Execution times of a task vary depending on the task's inputs and the initial hardware state. The vast number of cases prohibits exhaustive testing to exactly determine the worst-case execution time. Instead approximative methods have to be applied. Such methods must be conservative, i.e., they must never underestimate the worst-case execution time, they must statically overapproximate the dynamic behavior of a task on all possible inputs and hardware states.

Caches, deep pipelines, and all kinds of speculation are increasingly used in today's embedded systems to improve average-case performance. A designer that introduces such a component may find himself in the paradoxical situation that he has successfully improved the average-case performance of the system, but fails to derive sufficient timing guarantees despite his best efforts. This may be for two reasons: although the system's average-case behavior has improved, its worst-case performance has deteriorated. Even if the worst-case performance is sufficient, the *provable* bound may be too imprecise due

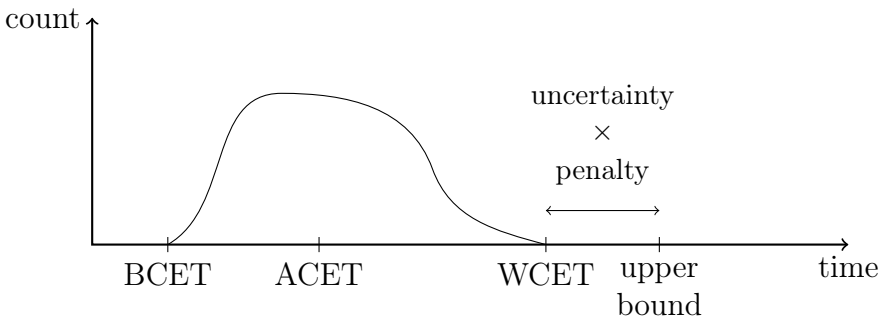


Figure 5.1: Execution times of tasks vary depending on inputs and the initial state of the hardware they are executed on. The figure depicts a distribution of execution times. The border cases are known as Best- and Worst-Case Execution Time (BCET and WCET). A correct timing analysis obtains a safe upper bound on all possible execution times.

to low predictability of the new components. Hence, a system with good average-case, but with poor worst-case performance or low predictability will not be certifiable. [Thiele and Wilhelm, 2004] describes threats to the predictability of systems and proposes design principles that support timing predictability.

The timing predictability of a system is a measure for the possibility of determining tight bounds on execution times. As depicted in Figure 5.1, timing predictability is composed of uncertainty and associated penalties. Uncertainty comprises timing accidents that cannot be excluded statically but never happen during execution. High penalties do not automatically make a system unpredictable: if there is no uncertainty this is not a problem. On the other hand, high levels of uncertainty only become harmful to timing predictability if the associated penalties are large.

As noted before, the processor caches have a strong influence on both the average-case and the worst-case performance. Due to the high cache-miss penalties they have a potentially strong impact on the predictability of a system. Several properties of the processor caches influence predictability: replacement policy, write policy, allocation policy, etc., see [Heckmann et al., 2003]. Of these, the replacement policy has by far the strongest influence on the predictability of the cache behavior. We will investigate the following previously introduced replacement policies regarding their timing predictability:

- Least Recently Used (LRU) used in Intel Pentium I and MIPS 24K/34K
- First-In First-Out (FIFO or Round-Robin) as used in ARM9, ARM11, Intel XScale
- Most Recently Used (MRU) as described in [Al-Zoubi et al., 2004, Malamy et al., 1994]
- Pseudo-LRU (PLRU) used in PowerPC 75x and Intel Pentium II-IV

The cache miss penalty is the same for all of the considered replacement policies. Timing predictability of cache replacement policies therefore only depends on the amount of uncertainty.

Contributions

We introduce two metrics, *evict* and $fill$, that capture our notion of the predictability of cache replacement policies.

Every cache analysis has to cope with a certain amount of uncertainty resulting from various sources explained in Section 5.2. The two metrics, *evict* and $fill$ indicate how quickly knowledge about cache hits and misses can be (re-)obtained. They mark a limit on the precision that *any* cache analysis can achieve, be it by abstract interpretation or any other sound method. A thorough analysis of the LRU, FIFO, MRU, and PLRU policies yields the respective values under these metrics.

Further analyses elaborate on these results and yield a more refined view on the limits of cache analyses: While *evict* and $fill$ constitute milestones in the recovery of information, supplementary results show how information evolves in between.

5.2 Uncertainty in Cache Analysis

In Section 4.1, we have introduced the concept of *may* and *must* cache information. *may* and *must* caches are upper and lower approximations, respectively, to the contents of all concrete caches that will occur whenever program execution reaches a program point.

Usually there is some uncertainty about the cache contents, i.e., the *may* and *must* caches do not coincide; there are memory blocks which can neither be guaranteed to be in the cache nor not to be in it.

The greater the uncertainty in the *must* cache, the worse the upper bound on the worst-case execution time will be. Similarly, greater uncertainty in the *may* cache entails a less precise lower bound on the best-case execution time. In addition, more cases have to be considered in the timing analysis making it more expensive, if a memory access cannot be classified as a hit or a miss.

There are several reasons for uncertainty about cache contents:

- Static cache analyses usually cannot make any assumptions about the initial cache contents. Cache contents on entrance depend on previously executed tasks. Due to timing anomalies [Lundqvist and Stenström, 1999, Reineke et al., 2006], even assuming a completely empty cache may not be conservative as shown in [Berg, 2006]. The only safe initial *must* cache is the empty set, whereas the only safe initial *may* cache must contain every memory block that may be mapped to the particular cache set.

- At control-flow joins, analysis information about different paths needs to be safely combined. Intuitively, one must take the intersection of the incoming *must* information and the union of the incoming *may* information. A memory block can only be in the *must* cache if it is in the *must* caches of all predecessor control-flow nodes, correspondingly for *may* caches.

- In data-cache analysis, the *address analysis* may not be able to exactly determine the address of a memory access. Then the *cache analysis* must conservatively account for all possible addresses. This especially deteriorates *may* information.
- Statically undetermined preempting tasks may change the cache state at preemption points [Gebhard and Altmeyer, 2007].

Since information about the cache state may thus be unknown or lost, it is important to recover information quickly to be able to classify memory accesses safely as cache hits or misses. This is possible for most caches. However, the *speed* of information recovery greatly depends on the cache replacement policy. It strongly influences how much uncertainty about cache hits and misses remains. Thus, the *speed of recovery* is an indicator of timing predictability.

5.3 Cache Predictability Metrics

We investigate how quickly cache contents become known when accessing a sequence of memory blocks starting from an unknown cache state. For the replacement policies we consider, an access to a cache set does not affect the state of other sets. Thus, we consider the recovery of information about single cache sets.

We assume all memory accesses in the regarded sequences to be *pairwise different*. This is sensible because recurring accesses do not contribute additional information about the cache contents. Another reason is that *arbitrarily* long access sequences can be constructed for two of the considered replacement policies, namely PLRU and MRU, that never recover complete information about the cache contents if repetitive accesses are allowed. In other words, there are access sequences such that different initial states result in different states for an arbitrary number of accesses; they never converge.

From sections 2.2.1 and 2.2.9 recall $update_{P(k)}$, $CC_{P(k)}$, $C^{P(k)}$, and $S^{\neq}$, which denotes the set of access sequences with pairwise different accesses. May and must information available after observing an access sequence s without knowing the initial set state can be defined as follows:

$$
\begin{aligned}
May^{P(k)}(s) &:= \bigcup_{q \in C^{P(k)}} CC_{P(k)}(update_{P(k)}(q, s)) \\
Must^{P(k)}(s) &:= \bigcap_{q \in C^{P(k)}} CC_{P(k)}(update_{P(k)}(q, s))
\end{aligned}
$$

$May^{P(k)}(s)$ is the set of cache contents that may still be in the cache set after accessing the sequence s, regardless of the initial cache state. Analogously, $Must^{P(k)}(s)$ is the set of cache contents that must be in the cache set after accessing the sequence s. Since we take into account every initial state, $Must^{P(k)}(s)$ is always a subset of $CC_{P(k)}(s)$.

The following two definitions show *how much* may and must information is available after observing any access sequence s of length n:

$$
\begin{aligned}
may^{P(k)}(n) &:= |May^{P(k)}(s)|, \text{where } s \in S^{\neq}, |s| = n \\
must^{P(k)}(n) &:= |Must^{P(k)}(s)|, \text{where } s \in S^{\neq}, |s| = n
\end{aligned}
$$

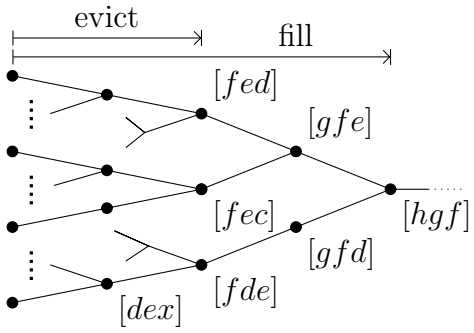


Figure 5.2: Initially different cache sets converge when accessing a sequence $\langle a, b, c, d, e, f, g, h, \ldots \rangle$ of pairwise different memory blocks. After *evict* accesses, any set contains only memory blocks from the access sequence. $fill$ accesses are required to converge to one completely known cache set. Selected cache sets are annotated with their respective contents.

Note that $may^{P(k)}(n)$ and $must^{P(k)}(n)$ are well-defined: For all sequences s of length n, $|May^{P(k)}(s)|$ is equal (the same goes for $|Must^{P(k)}(s)|$). The sequences contain pairwise different accesses only and are thus equal up to renaming. Thus, $May^{P(k)}(s_1)$ equals $May^{P(k)}(s_2)$ up to renaming, too, if $|s_1| = |s_2|$. In the following proofs we may therefore always restrict our attention to one representative access sequence.

Metrics

Based on $may^{P(k)}(n)$ and $must^{P(k)}(n)$ we are ready to define *evict* and $fill$ that indicate how quickly may and must information can be recovered:

Definition 5.1 (*evict* and $fill$)**.**

$$\begin{aligned} evict^P(k) &:= \min\left\{n \mid may^{P(k)}(n) \leq n\right\} \\ fill^P(k) &:= \min\left\{n \mid must^{P(k)}(n) = k\right\} \end{aligned}$$

Figure 5.2 illustrates the two metrics. $evict^P(k)$ tells us at which point we can safely predict that some memory blocks are no more in the cache, i.e., they are in the complement of may information. Any memory block not contained in the last $evict^P(k)$ accesses cannot be in the cache set: If some memory block not contained in the sequence could have "survived" then any other memory block not contained in the sequence could have "survived" as well. Then $may^{P(k)}(n) = c \gg n$ where c is the number of blocks that map to the cache set[1]. Less than $evict^P(k)$ accesses do not allow to predict any misses. The greater $evict^P(k)$, the longer it takes to gain may information, and furthermore, the

[1]The definition of *evict* and the following analyses are based on the reasonable assumption that c is larger than *evict*.

obtained may information is *less* precise. The obtained may information is less precise, because any of the greater number of $evict^P(k)$ memory blocks may still be in the cache set.

After $fill^P(k)$ pairwise different memory accesses we know exactly what is contained in the cache set, namely the last k accesses, i.e., we obtain complete may and must information. This allows us to precisely predict cache hits and misses. In contrast to may information, some must information is directly obtained with the first memory accesses. At least the most recently accessed memory block is in the cache set. Thus, it is pointless to define a counterpart to *evict* for must information, since $\min\{n \mid must^{P(k)}(n) \geq 1\} = 1$ for all policies.

Consider the implications of these metrics on *any* cache analysis. They mark a limit on achievable precision: no analysis can infer any may information (complete must information) given an unknown cache-set state and less than $evict(k)$ ($fill(k)$) pairwise different memory accesses. At the same time the metrics allow us to investigate the quality of different analyses. Does an analysis need longer access sequences to derive safe information about the cache set contents, or is it optimal with respect to the metrics?

Another application of these metrics is to determine the minimal effort to establish a desired cache-set state, assuming that no explicit instructions are available to do so. This may be used to eliminate initial uncertainty in cache analyses by prepending load instructions. Or simply to create uniform conditions for performance measurements. For this special purpose, it is interesting to investigate access sequences resulting in cache misses only. In such a case, a desired cache-set state can be obtained faster. We therefore distinguish M- and HM-access sequences: if we assume all accesses in the regarded sequences to be cache misses we denote this by the subscript M, otherwise by HM. Thus $fill^{\mathrm{LRU}}_{\mathrm{HM}}(8)$ is the number of pairwise different accesses (hits or misses) needed to know the exact contents of an 8-way cache set using LRU replacement. For brevity, we will also use $e(k)$ and $f(k)$ for $evict(k)$ and $fill(k)$.

As we have noted above, *some* must information can be immediately obtained with one individual access. The following definition of the *minimal life-span* captures how this generalizes:

Definition 5.2 (Minimal life-span)**.**

$$mls^P(k) := \max\left\{n \mid must^{P(k)}(n) = n\right\}$$

The minimal life-span is the minimal number of distinct accesses necessary to evict a memory block that has just been accessed out of a cache set (not counting the access that possibly brought the memory block into the set). In other words, it tells us how many of the most-recently used memory blocks are always in the cache.

Based on the minimal life-span of a policy, it is easy to determine some must information: the last $mls(k)$ accessed memory blocks are always in the cache set if they were pairwise different.

Equalities

The definitions given in the previous section were chosen to be as uniform as possible: they all relate $must^{P(k)}(n)$ and $may^{P(k)}(n)$ with k and n. However, for the following proofs we need to establish some equalities to ease argumentation about $evict$, $fill$, and mls.

Lemma 5.3. *$evict^P(k)$ is the minimal length of access sequences such that only memory blocks of the sequence may be contained in the cache set.*

$$evict^P(k) = \min\left\{n \mid \forall s \in S^{\neq}, |s| = n : May^{P(k)}(s) \subseteq SC(s)\right\}$$

Proof. We need to show $may^{P(k)}(n) \leq n \Leftrightarrow \forall s \in S^{\neq}, |s| = n : May^{P(k)}(s) \subseteq SC(s)$.

$\Leftarrow$ is clear since $|SC(s)| = |s| = n$ and therefore $|May^{P(k)}(s)| \leq n$.

$\Rightarrow$: Assume $May^{P(k)}(s) \not\subseteq SC(s)$ for some s. Then at least one memory block a not contained in s must have survived. Upon an access, the update process of the status bits is independent of the tag bits of all non-accessed memory blocks. Thus, the tag bits of a can be chosen arbitrarily. I.e., any other memory block $b \notin SC(s)$ could have survived as well. Then, $may^{P(k)}(n) = c \gg n$ where c is the number of blocks that map to the cache set. □

Lemma 5.4. *This following equation makes explicit that the cache set is filled with the last k accesses of the access sequence s, once its state is known.*

$$fill^P(k) = \min\left\{n \mid \forall s \in S^{\neq}, |s| = n, s = s_1 \circ s_2, |s_2| = k : Must^{P(k)}(s) = SC(s_2)\right\}$$

Proof. One needs to show $must^{P(k)}(n) = k \Leftrightarrow \forall s \in S^{\neq}, |s| = n, s = s_1 \circ s_2, |s_2| = k : Must^{P(k)}(s) = SC(s_2)$.

The $\Leftarrow$ direction of the equivalence is obvious. For $\Rightarrow$ one needs to show that whenever $must^{P(k)}(n) = k$ then for any sequence $s = s_1 \circ s_2$ of length $n \geq k$, where $|s_2| = k$, $Must^{P(k)}(s) = SC(s_2)$. This holds because $SC(s_2) \supseteq Must^{P(k)}(s)$: For any access a in s_2 there is an initial state, such that the access to a misses and the following less than k accesses in s_2 do not evict it. Therefore, each a in s_2 is contained in one of the intersected sets. In addition $|SC(s_2)| = k$. As $|Must^{P(k)}(s)| = k$, $SC(s_2)$ and $Must^{P(k)}(s)$ must be equal. □

Lemma 5.5. *An address a that has just been accessed will at least remain in the cache set for the $mls^P(k)$ subsequent accesses.*

$$mls^P(k) = \max\left\{n \mid \forall s \in S^{\neq}, |s| < n : a \in Must_k(\langle a \rangle \circ s)\right\}$$

Proof. We need to show $must^{P(k)}(n) = n \Leftrightarrow \forall s \in S^{\neq}, |s| < n : a \in Must_k(\langle a \rangle \circ s)$.

$\Rightarrow$: For all access sequences $s \in S^{\neq}$, $Must^{P(k)}(s) \subseteq SC(s)$. Since $must^{P(k)}(n) = n$, for all access sequences s of length n: $|Must^{P(k)}(s)| = n = |SC(s)|$. Therefore $Must^{P(k)}(s) = SC(s)$.

$\Leftarrow$: $must^{P(k)}(n) \geq n$ since the last n memory blocks are always contained in the cache set. Obviously $must^{P(k)}(n) \leq n$. □

5.4 LRU Caches

LRU replacement conceptually maintains a queue of length k for each cache set, where k is the associativity of the cache. If an element is accessed that is not yet in the cache (a miss), it is placed at the front of the queue. The last element of the queue is then removed if the set is full. It is the least-recently-used element of those in the queue. At a cache hit, the element is moved from its position in the queue to the front, in this respect treating hits and misses equally.

The contents of LRU caches are very easy to predict. For memory access sequences with pairwise different accesses and a strict least-recently-used replacement, we obtain the following tight bounds.

Theorem 5.6 ($evict^{\mathrm{LRU}}, fill^{\mathrm{LRU}}, mls^{\mathrm{LRU}}$)**.**
After k pairwise different memory accesses the contents of a k-way LRU*-set are precisely determined:*

$$evict^{\mathrm{LRU}}_{\mathrm{HM}}(k) = evict^{\mathrm{LRU}}_{\mathrm{M}}(k) = fill^{\mathrm{LRU}}_{\mathrm{HM}}(k) = fill^{\mathrm{LRU}}_{\mathrm{M}}(k) = mls^{\mathrm{LRU}}(k) = k$$

Proof. We show that $fill^{\mathrm{LRU}}_{\mathrm{HM}}(k) \leq k$, which entails the rest of the statements in the theorem. For any sequence $s = \langle a_1, \ldots, a_k \rangle \in S^{\neq}$ of length k and any state $q = [b_1, \ldots, b_n]_{\mathrm{LRU}(k)}$, $update_{\mathrm{LRU}(k)}(q, \langle a_1, \ldots, a_k \rangle) = [a_k, \ldots, a_1]_{\mathrm{LRU}(k)}$, because $a_1, \ldots, a_k$ are the k most-recently-used memory blocks. Therefore, $must^{\mathrm{LRU}(k)}(k) = k$. □

$evict(k)$ and $fill(k)$ tell us at which point any may and complete must information can be determined. However, the metrics do not tell us how may and must information evolves before and after these points. For the common case of an 8-way associative cache, we have precisely determined how much may and must information is available as a function in the number of accesses. Note that these functions mark the maximum information that can be obtained; a particular analysis may be less precise. Figure 5.3 shows plots of these functions. In the case of LRU replacement these functions are quite obvious, which will be different in the following cases of FIFO, MRU, and PLRU. Must information rises with every access up to the minimal life-span $mls^{\mathrm{LRU}}(8)$, which is equal to $fill^{\mathrm{LRU}}_{\mathrm{HM}}(8)$ and $evict^{\mathrm{LRU}}_{\mathrm{HM}}(8)$. Up to $evict(k)$ accesses, any memory block mapped to the cache set may reside in the set.

We have determined these functions by exhaustively generating all successor states of all possible initial cache-set states, exploiting symmetries. For LRU and FIFO replacement this could be rather easily determined analytically, but for the other cases this would have been very difficult. This automatic computation was only possible up to associativity 8 as the number of states grows rapidly with rising associativity.

5.5 FIFO Caches

FIFO cache sets can also be seen as a queue: new elements are inserted at the front evicting elements at the end of the queue. In contrast to LRU, hits do not change

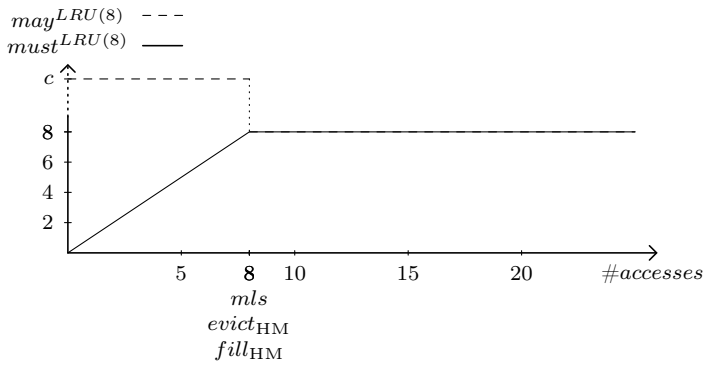


Figure 5.3: Evolution of may and must information of a 8-way LRU cache set. c is the number of blocks that can be mapped to the cache set. May and must information is shown by the dashed and the solid curve, respectively. From $fill(k)$ on the two functions have the same value.

the queue. Our representation of FIFO cache sets has to be interpreted in this way: In $[b, c, e, d]$, d will be replaced on a miss on x resulting in $[x, b, c, e]$, as described in Section 2.2.4.

Implementations use a modulo-k counter for each set pointing to the cache line to replace next. This counter is increased if an element is inserted into a set, while a hit does not change this counter.

In the case of misses only, FIFO behaves like LRU. Thus, the following tight bounds are obvious:

$$evict_{\mathrm{M}}^{\mathrm{FIFO}}(k) = fill_{\mathrm{M}}^{\mathrm{FIFO}}(k) = k$$

For the HM-case, we need the following lemma:

Lemma 5.7 (Surviving elements)**.**
Of $i \leq 2k-1$ pairwise different accesses, at least $\lceil \frac{i}{2} \rceil$ survive in a FIFO *cache set.*

Proof. Assume there were m misses and h hits, $m + h = i$. First, assume $m \geq h$. Every miss places a memory block at the front of the queue. $\min\{m, k\} \geq \lceil \frac{i}{2} \rceil$ of the memory blocks that caused misses must reside in the cache after sequence.

If $m \leq h$, we use the fact that each miss evicts at most one "known" memory block from the cache set, while inserting itself. Hence, with $h \leq k$ at least $m + (h - m) = h \geq \lceil \frac{i}{2} \rceil$ of the accessed memory blocks must be contained in the cache. □

Theorem 5.8 ($evict_{\mathrm{HM}}^{\mathrm{FIFO}}$)**.**
After accessing $2k-1$ pairwise different memory blocks in a k-way FIFO *set, the set contains only memory blocks from these $2k-1$ accesses. This bound is tight.*

Proof. Using Lemma 5.7 with $i = 2k-1$ gives $e_{\mathrm{HM}}^{\mathrm{FIFO}}(k) \leq 2k-1$. The following example shows the tightness. The access sequence $\langle x_1, \ldots, x_{k-1}, y_1, \ldots, y_{k-1} \rangle$ of length $2k-2$ conducted on the initial cache-set state $[z, x_1, \ldots, x_{k-1}]$ results in the state $[y_{k-1}, \ldots, y_1, z]$. Since z survived, $e_{\mathrm{HM}}^{\mathrm{FIFO}}(k) > 2k-2$. □

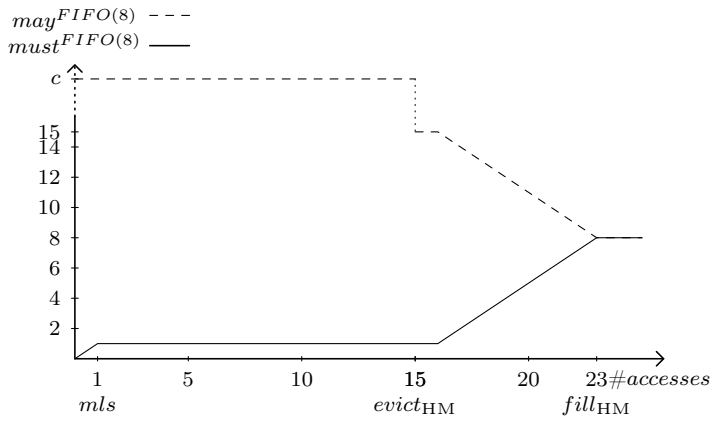


Figure 5.4: Evolution of may and must information of a 8-way FIFO cache set. c is the number of blocks that can be mapped to the cache set.

Theorem 5.9 ($fill_{\mathrm{HM}}^{\mathrm{FIFO}}$)**.**
One needs at most $3k-1$ *accesses for any initial cache-set state to reach a completely known cache-set state. This bound is tight.*

Proof. Theorem 5.8 states that after $2k-1$ accesses no more hits can occur. Since the next k accesses will be misses, $3k-1$ is a bound on $f_{\mathrm{HM}}^{\mathrm{FIFO}}(k)$. It is also a tight bound as shown by a similar example as in the proof of Theorem 5.8. Again, assume initial cache-set state $[z, x_1, \ldots, x_{k-1}]$. The sequence $\langle x_1, \ldots, x_{k-1}\rangle \circ \langle y_1, \ldots, y_{k-1}\rangle \circ \langle z, w_1, \ldots, w_{k-1}\rangle$ of length $3k-2$ results in the cache-set state $[w_{k-1}, \ldots, w_1, y_{k-1}]$, which does not contain z, which is one of the last k memory blocks that were accessed. So $f_{\mathrm{HM}}^{\mathrm{FIFO}}(k) > 3k-2$. □

Theorem 5.10 (mls^{FIFO})**.**
The minimum life-span of a memory block in a FIFO*-cache is* 1*.*

Proof. Since the queue is not changed on a hit, the memory block just accessed may reside at the end of the queue. Thus, it may be evicted with the next access. □

As in the LRU-case we have determined the evolution of must and may information for associativity 8 experimentally. Figure 5.4 illustrates the results. Disappointingly from a predictability point-of-view, must information exceeding the minimal life-span of 1 is only attained after 17 accesses.

5.6 MRU Caches

MRU stores one status bit for each cache line. In the following, we call these bits MRU-bits. Every access to a line sets its MRU-bit to 1, indicating that the line was recently used. Whenever the last remaining 0 bit of a sets status bits is set to 1, all other bits are reset to 0. This asymmetry in the last bit set to 1 will play a role as we will see

later. At cache misses, the line with lowest index (in our representation the left-most) whose MRU-bit is 0 is replaced.

We represent a sample state of an MRU cache set as $[a, b, c, d]_{0101}$, where 0101 are the MRU-bits and $a, \ldots, d$ are the contents of the set. On this state an access to e would yield a cache miss and the new state $[e, b, c, d]_{1101}$. Accessing d leaves the state unchanged. A hit on c forces a reset of the MRU-bits: $[e, b, c, d]_{0010}$.

Theorem 5.11 ($evict_{\mathrm{M}}^{\mathrm{MRU}}$ and $evict_{\mathrm{HM}}^{\mathrm{MRU}}$)**.**

$$evict_{\mathrm{M}}^{\mathrm{MRU}}(k) = evict_{\mathrm{HM}}^{\mathrm{MRU}}(k) = 2k - 2$$

gives a tight bound on the number of misses/accesses sufficient to evict all entries from a k-way set-associative MRU *cache set.*

Proof. We prove the tight bounds by showing $2k-2$ to be an upper bound for $evict_{\mathrm{HM}}^{\mathrm{MRU}}(k)$ and a lower bound for $evict_{\mathrm{M}}^{\mathrm{MRU}}(k)$. This suffices to prove the tightness for both, since by definition $evict_{\mathrm{M}}(k) \leq evict_{\mathrm{HM}}(k)$.

For the lower bound, consider the initial cache-set state $s = [x_1, \ldots, x_k]_{0\ldots001}$ and access sequence $\langle y_1, \ldots, y_{k-1}\rangle \circ \langle z_1, \ldots, z_{k-2}\rangle$. After the first part, the MRU-bits are reset, and state $s' = [y_1, \ldots, y_{k-1}, x_k]_{0\ldots010}$ results. The second part of the sequence replaces the memory blocks $y_1, \ldots, y_{k-2}$ resulting in the state $s'' = [z_1, \ldots, z_{k-2}, y_{k-1}, x_k]_{1\ldots110}$. x_k is still part of the set proving $evict_{\mathrm{M}}^{\mathrm{MRU}}(k) > 2k - 3$.

For the upper bound, notice that at some point during any k pairwise different accesses (hits or misses), the MRU-bits are reset. MRU-bits of lines that have not been accessed until this point are then set to 0. If it took k accesses to reset the bits, exactly these k memory blocks make up the cache set. Otherwise (less than k accesses), after the reset $k - 1$ MRU-bits are 0, and an additional $k - 1$ accesses are sufficient because accesses to memory blocks with MRU-bit 1 are impossible, from the reset point on. They would be hits and violate our assumption of pairwise different accesses. □

Theorem 5.12 ($fill^{\mathrm{MRU}}$)**.**
For the MRU *replacement policy it is impossible to give a bound on the number of accesses needed to reach a completely known cache-set state:*

$$fill_{\mathrm{HM}}^{\mathrm{MRU}}(k) = fill_{\mathrm{M}}^{\mathrm{MRU}}(k) = \infty$$

Proof. Consider an access sequence of pairwise different accesses. After at most $2k - 2$ accesses there will be only misses. Therefore a cache-set state $s = [x_1, \ldots, x_k]_{0\ldots01}$ will eventually occur for some $x_1, \ldots, x_k$. It will take $2k - 2$ further misses to eliminate x_k, hence future states following s will not consist of the last k accessed memory blocks. Even worse, we will reach similar states $[y_1, \ldots, y_k]_{0\ldots01}$ over and over again. □

The next two lemmas compensate this gap in the results by giving results similar to $fill^{\mathrm{MRU}}(k)$.

Lemma 5.13. *Consider an* MRU *cache-set state* $[x_1, \ldots, x_k]_{0\ldots010\ldots0}$ *and an access sequence that only produces misses. Every memory block from that sequence will remain in the cache set for at least* $k-1$ *accesses.*

Proof. Consider an arbitrary memory block e of the sequence. Since memory blocks are inserted from left to right, all memory blocks in the set left of e will be replaced earlier (after the next reset). Right of e there can be at most one memory block with MRU-bit 1. Thus, at least $k-2$ other cache lines will be accessed before the next reset and thus before e is replaced. □

Theorem 5.14. *Let* $k > 2$. *After at most* $2k-4$ *misses the last* $k-1$ *accessed memory blocks are present in the cache set, and the set is stable with respect to this weaker property. This bound is tight.*

$$\widetilde{fill}_{\mathrm{M}}^{\mathrm{MRU}}(k) := \min\left\{n \mid must_{\mathrm{M}}^{P(k)}(n) = k-1\right\} = 2k-4$$

Proof. The first reset of the MRU-bits occurs after at most $k-1$ accesses. If it takes exactly $k-1$ accesses the initial cache-set state fits the requirements of Lemma 5.13 proving the theorem for this case. Otherwise, the reset takes place after at most $k-2$ accesses. $k-2$ additional accesses are sufficient due to Lemma 5.13 because the miss causing the reset has an MRU-bit of 1 and cannot be evicted by the next $k-2$ misses.

Tightness is shown by the initial state $[x_1, \ldots, x_k]_{0\ldots011}$ and the sequence $\langle y_1, \ldots, y_{k-2}\rangle \circ \langle z_1, \ldots, z_{k-3}\rangle$: $[x_1, \ldots, x_k]_{0\ldots011} \to [y_1, \ldots, y_{k-2}, x_{k-1}, x_k]_{0\ldots0100}$ $\to [z_1, \ldots, z_{k-3}, y_{k-2}, x_{k-1}, x_k]_{1\ldots100}$. $y_{k-2}, z_1, \ldots, z_{k-3}$ are the last $k-2$ misses but neither x_{k-1} nor x_k which are still in the cache set belong to the last $k-1$ misses. □

Theorem 5.15. *Let* $k > 2$. *After at most* $3k-4$ *accesses (hits or misses) the last* $k-1$ *accessed memory blocks are present in the cache set, and the set is stable with respect to this weaker property. This bound is tight.*

$$\widetilde{fill}_{\mathrm{HM}}^{\mathrm{MRU}}(k) := \min\left\{n \mid must_{\mathrm{HM}}^{P(k)}(n) = k-1\right\} = 3k-4$$

Proof. Due to our general assumption about pairwise different accesses it holds that after the MRU-bits have been reset the second time, no more hits are possible because every line has been accessed at least once: every MRU-bit must have been 0 at some time and 1 later on. Now, Lemma 5.13 is applicable and $k-2$ further accesses are sufficient.

The first reset occurs after at most k accesses, the second one after exactly $k-1$ additional accesses. Adding the $k-2$ accesses after the second reset yields $3k-3$. We now exclude the cases where k accesses are needed for the first reset proving the upper bound of $3k-4$: if exactly k accesses were needed to reset the bits for the first time every cache line with MRU-bit 1 must have been accessed. Thus there are no further hits possible after the first reset, already.

Consider the following cache-set states and access sequences:

$$\begin{array}{rl} & [x_1, \ldots, x_{k-1}, x_k]_{0\ldots00011} \\ \xrightarrow{\langle x_k, u_1, \ldots, u_{k-2}\rangle} & [u_1, \ldots, u_{k-2}, x_{k-1}, x_k]_{0\ldots00100} \\ \xrightarrow{\langle v_1, \ldots, v_{k-4}, x_{k-1}\rangle} & [v_1, \ldots, v_{k-4}, u_{k-3}, u_{k-2}, x_{k-1}, x_k]_{1\ldots10110} \\ \xrightarrow{\langle v_{k-3}, v_{k-2}\rangle} & [v_1, \ldots, v_{k-4}, v_{k-3}, u_{k-2}, x_{k-1}, v_{k-2}]_{0\ldots00001} \\ \xrightarrow{\langle w_1, \ldots, w_{k-3}\rangle} & [w_1, \ldots, w_{k-3}, u_{k-2}, x_{k-1}, v_{k-2}]_{1\ldots11001}. \end{array}$$

The last $k-1$ accesses were $v_{k-3}, v_{k-2}, w_1, \ldots, w_{k-3}$, but v_{k-3} has just been evicted by w_{k-3}. Only the next access (evicting u_{k-2}) will make sure the last $k-1$ accessed memory blocks are present in the cache set.

This shows tightness for $k > 2$. Note that for $k = 4$ the accesses $v_1, \ldots, v_{k-4}$ and the MRU-bit prefixes $0 \ldots 0$ and $1 \ldots 1$ do not exist. □

Theorem 5.16 (mls^{MRU})**.**
The minimum life-span of a memory block in a MRU*-cache is* 2*.*

Proof. The MRU-bit of an accessed memory block e is always set to 1 resulting in $mls^{\mathrm{MRU}}(k) > 1$. But the next access may reset all the MRU-bits. If e is the left-most memory block it will be replaced with the next access, which yields $mls^{\mathrm{MRU}}(k) = 2$. □

The evolution of may and must information is depicted in Figure 5.5. As complete must information is never attained, the must-curve peaks at 7. Interestingly, may information never drops from the $14 = 2k - 2$ memory blocks that are reached after *evict* accesses. This can be explained quite easily: the memory block that causes the reset of the MRU-bits remains in the set for $2k - 2$ further accesses. Due to the unknown initial state any access could have caused the reset. This behavior is in contrast to that of LRU, FIFO, and PLRU, where eventually only the last k accessed memory blocks may reside in a cache set.

5.7 PLRU Caches

PLRU (Pseudo-LRU) is a tree-based approximation of the LRU policy. It arranges the cache lines at the leaves of a tree with $k - 1$ "tree bits" pointing to the line to be replaced next. A 0 indicating the left subtree, a 1 indicating the right. See Figure 5.6 or Section 2.2.6 for an explanation of the replacement policy. PLRU is much cheaper to implement than true LRU in terms of storage requirements and update logic. This comes at a price: it does not always replace the least-recently-used element.

PLRU has a special treatment of invalid lines. On a cache miss, invalid lines are filled from left to right, ignoring the tree bits. However, the tree bits are still updated on such an access.

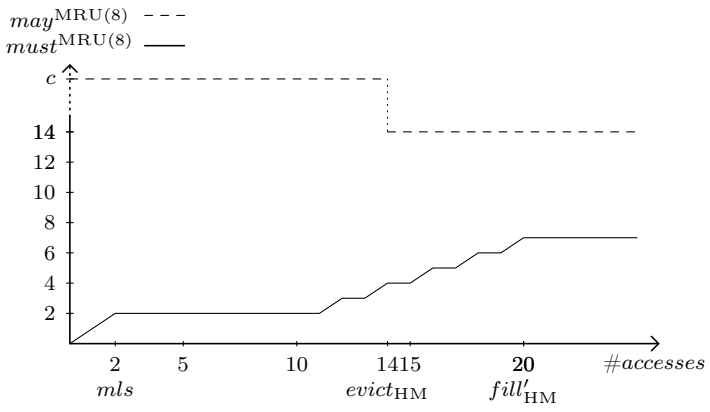


Figure 5.5: Evolution of may and must information of a 8-way MRU cache set. c is the number of blocks that can be mapped to the cache set. Note that complete must information cannot be obtained, thus $fill'$.

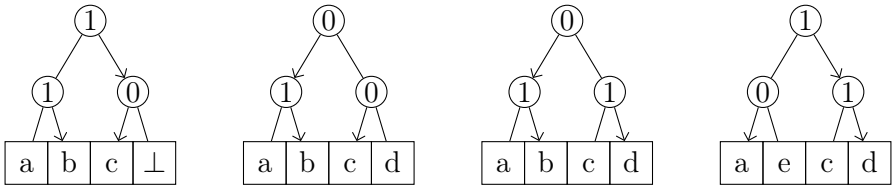
Since illustrating the states of these cache sets is rather complicated and sometimes are logically equivalent we introduce the notion of a *normalized cache-set state*. With no invalid lines, equivalent cache-set states with same content and same order of replacements can be obtained by interchanging neighboring subtrees and flipping the corresponding tree bit. We represent a concrete cache set by the equivalent one with all tree bits set to 1. For instance the concrete cache-set state $[a, b, c, d]_{010}$ with tree bits 010 in Figure 5.6 is represented by $[d, c, a, b]^{\cong}$. Flipping the 0-bits to 1 and interchanging the corresponding subtrees yields $[a, b, d, c]_{011}$ and finally $[d, c, a, b]_{111}$. Disregarding invalid lines the right-most element will be replaced in the normalized representation on a cache miss; it is pointed at by the tree bits. An access moves an element to the left-most position.

An *access path* to a cache line is a sequence of bits indicating the directions one has to take to walk from the root to this line in the normalized representation of the cache set; 0 for left, 1 for right. E.g. the access path of d in $[a, b, c, d, e, f, g, h]^{\cong}$ is 011.

We will interpret access paths as binary numbers. We will use two operators: $\overleftarrow{p_1 \dots p_n} = p_n \dots p_1$ to reverse the order of bits and $\overline{1100101} = 0011010$ to invert bits on paths.

Observation (Access path update)**.**
Consider elements $a \neq b$ with access paths p_a and p_b. Let $p_a = pre \circ p_1 \circ post_a$ and $p_b = pre \circ \overline{p_1} \circ post_b$, where $|p_1| = 1$, i.e., p_a and p_b have a common (possibly empty) prefix until they diverge and finish with (possibly empty) suffixes $post_a$ and $post_b$, respectively. Accessing b moves it to the front with access path $p'_b = 0 \dots 0$. Since a and b share a prefix, flipping the bits on the path to b also affects a's prefix: its new access path is $0 \dots 01 \circ post_a$.

Definition 5.17 (Miss replacement distance)**.**
The miss replacement distance *$mrd(e)$ of an element e is the minimum number of consecutive misses that are necessary to evict an element from a cache set q. For elements $e \notin q$ we define $mrd(e) = 0$.*



(a) Initial cache-set state $[a, b, c, \bot]_{110}$ with representation $[d, c, a, b]^{\cong}$.

(b) After a miss on d it becomes $[c, d, a, b]^{\cong}$.

(c) After a hit on c it becomes $[e, a, c, d]^{\cong}$.

(d) After a miss on e it becomes $[a, b, \bot, c]^{\cong}$.

Figure 5.6: Three accesses to a set of a 4-way set-associative PLRU cache: a miss on d followed by a hit on c and a miss on e. On a miss, one allocates invalid lines from left to right. If all lines are valid one replaces the line the tree bits point to. After every access all tree bits on the path from the accessed line to the root are set to point away from the line. Other tree bits are left untouched.

Lemma 5.18 (Miss replacement distance)**.**
A cache line e with access path $p_1 \dots p_n$ has miss replacement distance $mrd(e) = \overline{p_n \dots p_1}+1$ assuming no invalid lines.

Proof. Assuming no invalid lines, all misses will go to access path $1 \dots 1$. Each miss decrements $\overline{p_n \dots p_1}$ by 1 for $p_1 \dots p_n \neq 1 \dots 1$: consider the dissection of $p_1 \dots p_n$ into $1 \dots 10p_{post}$. A miss updates $p_1 \dots p_n$ to $0 \dots 01p_{post}$ by Observation 5.7. For $\overline{p_n \dots p_1}$ this means going from $\overleftarrow{p_{post}}10 \dots 0$ to $\overleftarrow{p_{post}}01 \dots 1$, i.e., $\overline{p_n \dots p_1}$ is decremented by one, unless $p_1 \dots p_n = 1 \dots 1$. In that case, it is evicted by the next miss. □

The cache line d with access path 011 from the example above will be replaced after $001 + 1 = 2$ consecutive misses: $011 \to 111 \to$ replaced.

Theorem 5.19 (mls^{PLRU})**.**
The minimum life-span of a memory block in a PLRU*-cache is $mls(k) = log_2 k + 1$. In fact, the $log_2 k + 1$ most-recently-used memory blocks alway reside in a* PLRU*-cache set.*

Proof. After the access to a memory block its access path is $0 \dots 0$. To replace this memory block all bits on its access path must be flipped to $1 \dots 1$. By Observation 5.7 each access to other memory blocks flips at most one of the bits of the access path to 1. To reach the lower bound of $log_2 k + 1$ one must access the neighboring subtrees in a bottom-up fashion, to avoid flipping bits back to 0. Obviously, repeatedly accessing memory blocks cannot accelerate eviction. Therefore, at least the last $log_2 k + 1$ *different* accesses to a cache set, or in other words the $log_2 k + 1$ most-recently-used memory blocks, reside in the set. □

Evict

Theorem 5.20 ($evict_{\mathrm{M}}^{\mathrm{PLRU}}$)**.**

$$evict_{\mathrm{M}}^{\mathrm{PLRU}}(k) = \begin{cases} 2k - \sqrt{2k} & : k = 2^{2i+1}, i \in \mathbb{N}_0 \\ 2k - \frac{3}{2}\sqrt{k} & : otherwise \end{cases}$$

is a tight bound on the number of misses to evict all entries from a k-way set-associative PLRU *cache set.*

Proof. Assuming no invalid lines, this proof is easy. It is a simple consequence of Lemma 5.18 that k misses suffice to evict a complete set. If all lines are invalid, the problem is equally easy. It becomes more complicated if some subset of size $0 < i < k$ of the lines is invalid. The first i misses will then go into these invalid lines instead of following the standard PLRU replacement policy. These accesses do however modify the tree bits in the standard way, as if they had been hits.

The number of misses needed to completely evict the cache set is then determined by the positions of the remaining $k' = k - i$ non-accessed lines. Each line can be associated with the number of misses necessary to replace the content of that line. By Lemma 5.18 the line with access path $p_1 \dots p_n$ will be replaced after $\overline{p_n \dots p_1} + 1$ consecutive misses, i.e., the number of trailing 0s in $p_1 \dots p_n$ mainly determines the miss replacement distance. To have m trailing 0s none of the $2^m - 1$ neighbors in the particular subtree of height m may have been accessed in the first phase, filling up the invalid lines. Any access in the subtree would have flipped at least one of the final m bits. If k' lines have not been accessed yet, the maximal number of trailing 0s in any of these lines' access paths may be $\lfloor log_2 k' \rfloor$.

So, the maximal distance to eviction of any untouched line is bounded by

$$\begin{aligned} \overline{\underbrace{0\dots0}_{\lfloor log_2 k' \rfloor}10\dots0} + 1 &= \underbrace{1\dots1}_{\lfloor log_2 k' \rfloor}01\dots1 + 1 \\ &= \underbrace{1\dots1}_{\lfloor log_2 k' \rfloor + 1}0\dots0 = \underbrace{1\dots1}_{log_2 k} - \underbrace{1\dots1}_{log_2 k - (\lfloor log_2 k' \rfloor + 1)} \\ &= (2^{log_2 k} - 1) - (2^{log_2 k - (\lfloor log_2 k' \rfloor + 1)} - 1) = k - \frac{k}{2^{\lfloor log_2 k' \rfloor + 1}} \end{aligned}$$

All in all, we get $z = i + k - \frac{k}{2^{\lfloor log_2 k' \rfloor + 1}} = 2k - k' - \frac{k}{2^{\lfloor log_2 k' \rfloor + 1}}$ as an upper bound for the number of accesses to evict a PLRU-set with misses only. Obviously, z is maximized by a power of two (for any non power of two $k' = 2^l + \delta, 0 < \delta < 2^l$, $k'' = 2^l$ results in a higher value of z), which allows us to simplify the formula to $2k - k' - \frac{k}{2k'}$, assuming k' is a power of two. Maximizing this yields

$$evict_{\mathrm{M}}^{\mathrm{PLRU}}(k) = \begin{cases} 2k - \sqrt{2k} & : k = 2^{2i+1}, i \in \mathbb{N}_0 \\ 2k - \frac{3}{2}\sqrt{k} & : \text{otherwise} \end{cases}$$

with

$$k' = \begin{cases} \frac{1}{2}\sqrt{2k} & : k = 2^{2i+1}, i \in \mathbb{N}_0 \\ \sqrt{k} & : \text{otherwise} \end{cases}$$

This proves the given $evict_{\mathrm{M}}^{\mathrm{PLRU}}(k)$ to be an upper bound. To prove its tightness we can give access sequences and initial cache configurations that exactly reach the bounds. Assume $[\bot_1, \ldots, \bot_{k-k'}, x_1, \ldots, x_{k'}]$ with arbitrary tree bits as the initial configuration. Then, the access sequence $\langle y_1, \ldots, y_{k-k'} \rangle$ results in the normalized cache-set state $\left[y_{i_1}, \ldots, y_{i_{k'}}, x_{i_1}, \ldots, x_{i_{k'}}, y_{i_{k'+1}}, \ldots, y_{i_{k-k'}}\right]^{\cong}$. Since k' is a power of two the $x_{i_1}, \ldots, x_{i_{k'}}$ make up a complete subtree. Therefore, they are not torn apart by accessing other lines in the normalized representation. Furthermore, $y_{k-k'}$ fills $\bot_{k-k'}$ which is adjacent to $x_{i_1}, \ldots, x_{i_{k'}}$, moving the x_i-subtree to second position from the left. Observe that the access path $0 \ldots 01\underbrace{0 \ldots 0}_{log_2 k'}$ leads to x_{i_1}. By Lemma 5.18, it takes $\underbrace{1 \ldots 1}_{log_2 k'}01 \ldots 1{+}1 = k - \frac{k}{2k'}$ further misses to eliminate x_{i_1}. Together with the $k - k'$ previous accesses to fill the invalid lines, it sums up to the given upper bound, proving its tightness. □

If one cannot assume that only misses will occur, the number of accesses for eviction gets even larger. However, we do not have to consider invalid lines because allocations to invalid lines are equivalent to hits at those position.

For the case of hits and misses we need a simple lemma that relates the number of accesses to the two halves of a cache set:

Lemma 5.21. *The number of accesses to the two halves c_1, c_2 of a $2k$-way cache set differs by at most k.*

Proof. Consider a situation with h_i hits and m_i misses to c_i. For each but the first miss on c_2 there must be an access to c_1 to flip the bits back to c_2: $h_1 + m_1 \geq m_2 - 1$. Thus the difference $d = (h_2 + m_2) - (h_1 + m_1) \leq m_2 + h_2 - m_2 + 1 = h_2 + 1$. If $h_2 < k$ then $d \leq k$. The last possible case is $h_2 = k$, in which all hits h_2 must have preceded all misses m_2 due to the accesses in the sequence being pairwise different. But every further access to c_2 must then be directly preceded by at least one access to c_1 again yielding $d \leq k$. $(h_1 + m_1) - (h_2 + m_2) \leq k$ by a similar argument. □

Theorem 5.22 ($evict_{\mathrm{HM}}^{\mathrm{PLRU}}$)**.**
It takes at most $\frac{k}{2}\log_2 k + 1$ pairwise different accesses to evict all entries from a k-way set-associative PLRU *cache set. Again, this is a tight bound.*

Proof. Claim: let $z(k)$ be an upper bound for the number of accesses needed to evict a cache set of associativity k. Then $z(2k) = 2z(k) + k - 1$ is an upper bound for a set of associativity $2k$.

We consider a set of size $2k$ to be composed of two halves c_1, c_2 of size k. Wlog. let c_1 be the first half with no initial contents left. Let a_1 and a_2 be the number of accesses on c_1 and c_2 respectively to reach this state. Then c_2 needs at most $z(k) - a_2$ further accesses. Since c_1 consists of memory blocks from the access sequence only, every subsequent

access to c_1 will be a miss. Therefore, there can be at most one access to c_1 between two consecutive accesses to c_2 from now on.

Combining the last two statements there can be at most $2(z(k)-a_2)-1$ further accesses until c_2 is completed, too. Adding the first a_1+a_2 accesses results in $a_1+a_2+2(z(k)-a_2)-1 = 2z(k)+a_1-a_2-1$. Using Lemma 5.21 this is bounded by $2z(k)+k-1$.

Solving the recurrence for z with the trivial value $z(2)=2$ proves the upper bound.

To prove tightness assume a worst-case initial cache-set state c_k and a worst case access sequence $s_k = \langle u_1, \ldots, u_{z(k)} \rangle$ for associativity k are known. The access sequence $\langle x_1, \ldots, x_k, u_1, v_1, \ldots, u_{z(k)-1}, v_{z(k)-1}, u_{z(k)} \rangle$ evicts the contents of the cache set with initial state $[x_1, \ldots, x_k] \circ c_k$ with no less than $k+2z(k)-1$ accesses.

For $k=2$ all cache sets states and all access sequences of length 2 are worst case initial cache-set states serving as a basis for the recursion. □

Fill

Theorem 5.23 ($fill_{\mathrm{M}}^{\mathrm{PLRU}}$)**.**
After at most $fill_{\mathrm{M}}^{\mathrm{PLRU}}(k) = 2k-1$ *misses the cache-set state is completely known. This bound is tight for* $k > 2$*. For* $k = 2$*,* 2 *is an obvious tight bound for* $fill_{\mathrm{M}}^{\mathrm{PLRU}}$*.*

Proof. At most k misses can go into invalid lines. The last of these accesses resides in the line with access path $0 \ldots 0$ in the normalized cache set. According to Lemma 5.18, it will be evicted after k further misses, i.e., the $k-1$ subsequent misses fill up the cache set. Further misses result in a FIFO behavior. The following example proves tightness: assume the initial cache-set state $c = [\perp_1, \ldots, \perp_k]$ consisting of invalid lines only. Now, consider the access sequence $\langle x_1, \ldots, x_k \rangle \circ \langle y_1, \ldots, y_{k-2} \rangle$. After processing $\langle x_1, \ldots, x_{\frac{k}{2}} \rangle$ $x_{\frac{k}{2}}$ has access path $0 \ldots 0$. The next accesses x_i go to the other half of c. Thus, the access paths of $x_{\frac{k}{2}}$ and x_i have no common prefix. By Observation 5.7, $x_{\frac{k}{2}}$ has access path $10 \ldots 0$ after $\langle x_{\frac{k}{2}+1}, \ldots, x_k \rangle$. By Lemma 5.18, it will take $1 \ldots 10+1 = k-1$ further misses to eliminate it, after $k-2$ accesses it is still in the cache set. Thus, the cache set does not consist of the last k accessed memory blocks, in particular it has not stabilized yet. □

Lemma 5.24. *If it takes* $evict_{\mathrm{HM}}^{\mathrm{PLRU}}(k)$ *accesses to evict a cache set, the last two accesses must have gone to different halves of the cache set.*

Proof. Assuming this is false one could insert an additional miss-access between the last two accesses on the half not accessed. Thus the number of accesses for eviction would be increased by one contradicting the assumption of a worst case. □

Theorem 5.25 ($fill_{\mathrm{HM}}^{\mathrm{PLRU}}$)**.**
After at most $\frac{k}{2}\log_2 k + k - 1$ *pairwise different accesses the* PLRU *cache-set state is completely known. This bound is tight.*

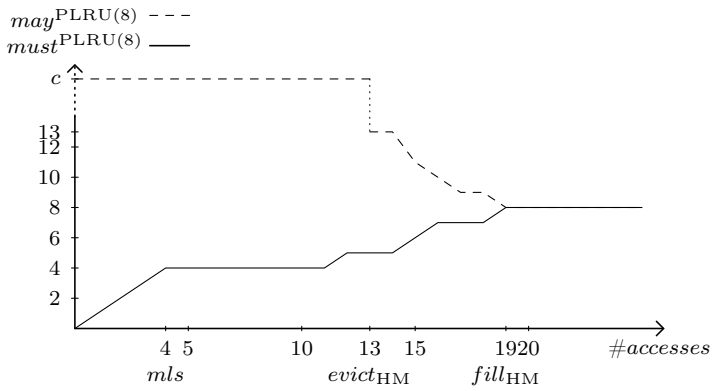


Figure 5.7: Evolution of may and must information of a 8-way PLRU cache set. c is the number of blocks that can be mapped to the cache set.

Proof. We want to prove the given bound based on our results for $evict_{\mathrm{HM}}^{\mathrm{PLRU}}(k)$. The difference $fill_{\mathrm{HM}}^{\mathrm{PLRU}}(k) - evict_{\mathrm{HM}}^{\mathrm{PLRU}}(k)$ is $k-2$. Since the last access to a set always resides in the left-most position with access path $0\ldots0$, $k-1$ additional misses suffice to fill the set due to Lemma 5.18. This still leaves us one short of the given bound if eviction took exactly $evict_{\mathrm{HM}}^{\mathrm{PLRU}}(k)$ steps. In that case, however, the last two accesses must have gone to different halves due to Lemma 5.24. Thus, they have access paths $0\ldots0$ and $10\ldots0$. Due to Lemma 5.18 they will be replaced after k and $k-1$ misses. Thus $k-2$ further accesses suffice.

Tightness is shown by modifying a generic worst-case example for $e_{\mathrm{HM}}^{\mathrm{PLRU}}(k)$. Let $s = \langle x_1, \ldots, x_e\rangle$ be this worst-case access sequence (assuming the same initial cache-set state). Let $|$ denote the center of the cache set. Then $s' = \langle x_1, \ldots, x_{e-2}, h\rangle \circ \langle y_1, \ldots, y_{k-2}\rangle$ of length $evict_{\mathrm{HM}}^{\mathrm{PLRU}}(k)+k-3$ results in the intermediate cache-set state $[h, \ldots, x_{e-2}, \ldots |$ $x_{e-3}, \ldots]^{\cong}$ The final cache-set state is $\left[y_{i_1}, \ldots, y_{i_{k-2}}, x_{e-3}\right]^{\cong}$.

Effectively, we remove the last two accesses from the old example and insert a hit h into the access sequence accessing the left side of the (normalized) cache set. Knowing that the last two accesses x_{e-3}, x_{e-2} accessed different halves of the set[2], the hit h changes the order in which these two memory blocks will be replaced. Thus x_{e-3} must be evicted from the set to stabilize it. Due to Lemma 5.18 this takes $k-1$ additional accesses because x_{e-3} has access path $10\ldots0$ after the hit. Carrying out s' only, will result in the cache-set state depicted above, which is not yet stabilized. □

The evolution of may and must information for a PLRU-set of associativity $k = 8$ is depicted in Figure 5.7. As in every policy, must information initially rises up to $mls(k)$ and reaches k after $fill(k)$ accesses; may information drops to *evict* after *evict* accesses. The further development of both curves is less uniform than in the other cases, which might be attributed to the more complicated policy.

[2]This is due to the construction of our former worst-case example, see Theorem 5.22.

5.8 Related Work

Sleator and Tarjan [Sleator and Tarjan, 1985] examine replacement policies from a different point of view. They investigate the amortized efficiency of the list update and paging rules LRU, FIFO, LIFO, and LFU. As a reference they take optimal offline policy OPT [Belady, 1966]. They show that any online algorithm must fare worse than OPT by a certain factor and go on to prove that LRU and FIFO do perform as well as possible for an online algorithm. Their work concerns theoretical performance limits rather than predictability of replacement policies. In Chapter 6, we slightly extend Sleator and Tarjan's notion of competitiveness to that of relative competitiveness.

Al-Zoubi et al. perform measurements using the SPEC CPU2000 benchmarks [Al-Zoubi et al., 2004], comparing the performance of different associativities and replacement policies including FIFO, LRU, PLRU, MRU, and OPT. They conclude that LRU, PLRU, and MRU show nearly the same performance. These policies exhibit similar miss ratios as caches of half the size with OPT replacement. On the average they clearly outperform FIFO. This interesting experimental result yields insights concerning average-case performance in practice. It does however, not deal with predictability. Similarly, [Grund and Reineke, 2008] efficiently estimate the performance of different cache replacement policies on particular workloads using a Markov model. However, both [Al-Zoubi et al., 2004] and [Grund and Reineke, 2008] give no *guarantees* on the worst- or even the average-case performance.

Heckmann et al. provide must and may analyses for LRU, PLRU, and a pseudo round-robin replacement policy in the context of worst-case execution time tools [Heckmann et al., 2003]. Cache lines are assigned ages where "old" lines are close to eviction. Newly introduced lines assume the minimum age 0. Updates change these ages to account for all possible concrete scenarios: in the may analysis, the minimal possible age is taken, in the must analysis the maximal. For LRU, this yields very precise and efficient analyses. For PLRU, the must analysis loses precision while staying efficient. It can maximally infer 4 of the 8 lines of an 8-way set-associative PLRU cache set which is strongly related to our Theorem 5.19 and the competitiveness of PLRU relative to LRU, as discussed in Chapter 6. The may analysis becomes useless since only ages 0 and 1 are reachable.

The manual of the PowerPC 75x series [Freescale Semiconductor Inc., 2002] gives the number of uniquely addressed misses to flush an 8-way PLRU cache set used in these CPUs, namely 12, which is an instance of $evict_{\mathrm{M}}^{\mathrm{PLRU}}$.

In summary, Al-Zoubi et al. provide empirical performance results whereas Sleator and Tarjan present a theoretical performance analysis that is independent of any particular benchmark.

In contrast to performance, predictability in the sense of this chapter is concerned with the obtainable precision of provable upper and lower bounds on execution times. Static analysis is used to determine such bounds. Heckmann et al. provide specific static cache analyses for several replacement policies and compare their precision. Our work presents the theoretical limits of *any* static cache analysis.

Table 5.1: Summary of the main results for all policies.

Policy	$e_{\mathrm{M}}(k)$	$f_{\mathrm{M}}(k)$	$e_{\mathrm{HM}}(k)$	$f_{\mathrm{HM}}(k)$	$mls(k)$
LRU	k	k	k	k	k
FIFO	k	k	$2k-1$	$3k-1$	1
MRU	$2k-2$	$\infty/2k-4$[§]	$2k-2$	$\infty/3k-4$[§]	2
PLRU	$\left\{ \begin{array}{l} 2k-\sqrt{2k} \\ 2k-\frac{3}{2}\sqrt{k} \end{array} \right\}$	$2k-1$	$\frac{k}{2}\log_2 k+1$	$\frac{k}{2}\log_2 k+k-1$	$\log_2 k+1$

Table 5.2: Examples for *evict* and $fill$ for $k = 4, 8$.

	$k=4$					$k=8$				
Policy	e_{M}	f_{M}	e_{HM}	f_{HM}	mls	e_{M}	f_{M}	e_{HM}	f_{HM}	mls
LRU	4	4	4	4	4	8	8	8	8	8
FIFO	4	4	7	11	1	8	8	15	23	1
MRU	6	$\infty/4$	6	$\infty/8$	2	14	$\infty/12$	14	$\infty/20$	2
PLRU	5	7	5	7	3	12	15	13	19	4

5.9 Summary, Conclusions, and Future Work

An important part in the design of hard real-time systems is the proof of timeliness, which is determined by the worst-case performance of the system. Performance boosting components like caches have an increasing impact on both the average and the worst-case performance. We investigated the predictability of four popular cache replacement policies. To this end, we introduced the metrics *evict* and $fill$ and determined their values.

In these metrics, no policy can perform better than LRU because k is an obvious lower bound for any replacement policy. The other policies under investigation, PLRU, MRU, and FIFO, perform considerably worse: in the more interesting cases of $evict_{\mathrm{HM}}(k)$ and $fill_{\mathrm{HM}}(k)$, FIFO and MRU exhibit linear growth in terms of k, while PLRU grows super-linearly. However, instantiating k with the common values 4 and 8 shows a different picture, see Table 5.2. Here, PLRU even fares slightly better than FIFO and MRU. Yet, compared to 8-way LRU, PLRU, MRU, and FIFO take more than twice as long to regain complete information. In particular, this differs from the worst-case *performance* results obtained in [Sleator and Tarjan, 1985], where FIFO and LRU fared equally well.

Our analysis of the evolution of may and must information further substantiates the findings: MRU and even more so FIFO should not be considered for use in hard-real time systems. These results support previous practical experience in static cache analysis [Heckmann et al., 2003].

[§]See Theorem 5.14 and Theorem 5.15.

The metrics allow us to investigate the precision of different analyses. Does an analysis ever regain any may or complete must information? If so, does it need longer access sequences to derive safe information about the cache contents than suggested by $fill(k)$ and $evict(k)$, or is it optimal with respect to these metrics?

Future work could drop the restriction that all memory blocks of access sequences are different. This could allow for the construction of precise and efficient (as possible) cache analyses, as we are now aware of the limits. A first step would be to investigate the normalization of arbitrary access sequences, e.g. $\langle x_1, \ldots, x_n, y, y \rangle$ can be simplified to $\langle x_1, \ldots, x_n, y \rangle$ in all replacement policies we considered. For LRU it suffices to keep the last access to each memory block within the sequence, which means keeping at most k memory blocks. Can we do something similar regarding FIFO or PLRU?

6

Relative Competitiveness of Replacement Policies

We present a tool to automatically compute relative competitive ratios for a large class of replacement policies, including LRU, FIFO, MRU, PLRU and OPT. Relative competitive ratios bound the performance of one policy relative to the performance of another policy. These performance relations allow us to use cache-performance predictions for one policy to compute predictions for another, including policies that could previously not be dealt with.

Most of the work presented in this chapter has been published in [Reineke and Grund, 2008a]. An extended abstract appeared in [Reineke and Grund, 2008b]. The chapter goes beyond [Reineke and Grund, 2008b, Reineke and Grund, 2008a] in its treatment of MRU and OPT, and in several general theorems about relative competitiveness.

6.1 Introduction

Developing cache analyses – analyses that statically determine whether a memory access associated with an instruction will always be a hit or a miss – is a difficult problem. Precise and efficient analyses have been developed for set-associative caches that employ the least-recently-used (LRU) replacement policy [Ferdinand et al., 1997, Ferdinand and Wilhelm, 1999, White et al., 1997, Ghosh et al., 1998, Chatterjee et al., 2001]. A state-of-the-art LRU analysis has been described in Section 4.2. Other commonly used policies, like first-in-first-out (FIFO) or Pseudo-LRU (PLRU), are more difficult to analyze, as we have seen in Chapter 5. We are not aware of any published analysis that may safely predict cache misses in the presence of FIFO, MRU or PLRU replacement.

Relative competitive analyses yield upper (lower) bounds on the number of misses (hits) of a policy P relative to the number of misses (hits) of another policy Q. For example, a competitive analysis may find out that policy P will incur at most 30% more misses than policy Q and at most 20% less hits in the execution of any task. Note that P and Q may have different associativities.

We propose the following approach to determine safe bounds on the number of cache hits and misses by a task T under FIFO(k), PLRU(l)[1], or any another replacement policy:

1. Determine competitiveness of the desired policy P relative to a policy Q for which a cache analysis exists, like LRU.
2. Perform cache analysis of task T for policy Q to obtain a cache-performance prediction, i.e., upper (lower) bounds on the number of misses (hits) by Q.
3. Calculate upper (lower) bounds on the number of misses (hits) for P using the cache analysis results for Q and the competitiveness results of P relative to Q.

Note that step 1 has to be performed only once for each pair of replacement policies.

The approach is safe due to the monotonicity of the performance relations: They preserve upper (lower) bounds on the number of misses (hits).

A limitation of this approach is that it only produces upper (lower) bounds on the number of misses (hits) for the whole program execution. It does not reveal at which program points the misses (hits) will happen, something many timing analyses need. We will demonstrate that relative competitiveness results can also be used to obtain sound *may* and *must* cache analyses [Ferdinand and Wilhelm, 1999], i.e., analyses that can classify individual accesses as hits or misses.

In this chapter, we present a tool to automatically compute relative competitiveness results for a large class of replacement policies, including LRU, FIFO, MRU, PLRU, and OPT. We generalize some of the automatically computed results, which hold for fixed associativities, to arbitrary associativities. This is aided by the ability of our tool to generate example memory access sequences that exhibit the worst-case relative behavior. One of our results is that for any associativity k and any workload, FIFO(k) shows at least half the number of hits that LRU(k) would show.

Outline

In Section 6.2 we formally introduce our notion of relative competitiveness and show how to use it to obtain cache-performance predictions. In addition we discuss some general properties of the definitions. In Section 6.3, we describe how to compute competitive ratios automatically. Section 6.4 presents results obtained with our tool and a number of generalizations to arbitrary associativities. We delineate our work from previous work on competitive analysis in Section 6.5. Important consequences of our results and possibilities of future work are summarized in Section 6.6.

6.2 Relative Competitiveness

In this section, we formally define our notion of relative competitiveness and show how to use it to obtain cache-performance predictions. In addition we derive some useful general properties of our concepts.

[1] k and l denote the respective associativities of FIFO(k) and PLRU(l).

a, b, c	$\in$	M	the set of memory blocks
$\langle b, b\rangle, \langle b, c, d\rangle, s, t$	$\in$	$S = M^*$	the set of finite access sequences
$\langle b, c, d\rangle, s, t$	$\in$	$S^{\neq} \subset S$;	the set of finite access sequences with pairwise different accesses
P, Q	$\in$	$Policy$	the class of replacement policies
$[b, e, c, f]_P, i^P, p$	$\in$	C^P	the set of reachable cache-set states of policy P with i^P the initial state of P after starting up the hardware
$update_P$	:	$C^P \times S \to C^P$	function computing the effect of an access sequence on a cache-set state under policy P
m_P	:	$C^P \times S \to \mathbb{N}$	the number of misses by policy P on a given access sequence and cache-set state.
h_P	:	$C^P \times S \to \mathbb{N}$	the number of hits by policy P on a given access sequence and cache-set state.

Figure 6.1: Domains and notations.

6.2.1 Definition of Relative Competitiveness

For the following definitions consider the domains and notations introduced in Section 2.2.9 which are briefly recapitulated in Figure 6.1. The most important notions are $m_P(q, s)$ and $h_P(q, s)$, which compute the number of of misses and hits, respectively, of policy P starting in state q processing access sequence s. $update_P(q, s)$ computes the cache-set state after accessing a sequence s in state q under policy P. C^P is the set of reachable cache-set states of policy P and S denotes the set of finite access sequences.

Before giving the central definitions we first need to introduce compatible cache-set states. We want to relate two policies in compatible state only. Intuitively, this ensures that no policy is given an undue advantage by the state it is starting in.

Definition 6.1 (Compatible states)**.**
Two cache-set states $p \in C^P$ and $q \in C^Q$ are called compatible, denoted $p \sim q$, iff there is some access sequence $s \in S$, such that $p = update_P(i^P, s)$ and $q = update_Q(i^Q, s)$.

Now we are ready to define our notion of relative competitiveness. It captures the worst-case performance of one policy relative to another policy:

Definition 6.2 (Relative miss-competitiveness)**.**
A policy P is k-miss-competitive relative to policy Q with additive constant c, short (k, c)-miss-competitive, if

$$m_P(p, s) \leq k \cdot m_Q(q, s) + c$$

for all access sequences $s \in S$ and cache-set states $p \in C^P, q \in C^Q$ that are compatible $p \sim q$.

In other words, policy P will incur at most k times the number of misses of policy Q plus a constant c on any access sequence. Observe that this relation holds for individual cache sets *not* for entire set-associative caches consisting of several cache sets. In the original definition of competitiveness [Sleator and Tarjan, 1985], online policies were compared to the optimal offline policy OPT, i.e., instantiating Q in our definition with OPT yields the original definition except for the treatment of compatible states. In [Sleator and Tarjan, 1985], the two states p and q do not have to be compatible.

We can define relative hit-competitiveness analogously to relative miss-competitiveness. The definition has no counterpart in the original competitiveness definition of [Sleator and Tarjan, 1985]. This is because no online policy is hit-competitive relative to the optimal offline policy OPT[2]. However, in our setting of comparing two online policies this is possible.

Definition 6.3 (Relative hit-competitiveness)**.**
A policy P is k-hit-competitive relative to policy Q with subtractive constant c, short (k, c)-hit-competitive, if

$$h_P(p, s) \geq k \cdot h_Q(q, s) - c$$

for all access sequences $s \in S$ and cache-set states $p \in C^P, q \in C^Q$ that are compatible $p \sim q$.

Notice, that the two definitions are not redundant. If policy A is k-miss-competitive relative to policy B, with $k > 1$, this does not give us any clue regarding the hit-competitiveness of A relative to B: Rewriting Definition 6.2 in terms of hits ($h_P(q, s) = |s| - m_P(q, s)$) yields $|s| - h_P(q, s) \leq k \cdot (|s| - h_Q(q', s)) + c$. For $k > 1$ this inequality depends on $|s|$, the length of the access sequence.

We sometimes say that a policy is competitive relative to another policy without specifying an appropriate additive (subtractive) constant. In such cases, we implicitly demand that such a constant exists. The following definition is an example of such a case:

Definition 6.4 (Competitive ratio)**.**
The competitive miss *and* hit ratios *$c^m_{P,Q}$ and $c^h_{P,Q}$ of P relative to Q are defined as*

$$\begin{aligned} c^m_{P,Q} &= \mathit{inf}\,\{k \mid P \text{ is } k\text{-miss-competitive relative to } Q\} \\ \mathit{and}\ c^h_{P,Q} &= \mathit{sup}\,\{k \mid P \text{ is } k\text{-hit-competitive relative to } Q\}. \end{aligned}$$

In our cases, there is always a smallest (greatest) k, such that P is k-miss-competitive (k-hit-competitive) relative to Q, which is not the case in general. Our focus will be on computing competitive ratios and appropriate additive (subtractive) constants.

Why are we interested in competitive ratios? Consider a policy A that is k-miss-competitive relative to policy B. A is also l-miss-competitive relative to B for $l > k$.

[2]For every online policy P one can incrementally construct an arbitrarily long access sequence that incurs no hits under P: Always access the element that was just evicted by the online policy. Due to its knowledge of future accesses, many of these accesses will be hits in OPT.

However, the former statement is clearly a better characterization of the policy's relative competitiveness. In this sense, the competitive ratio is the *best* characterization of the policy's relative competitiveness. In particular, there are access sequences, such that the ratio between the number of misses (hits) in policy A and the number of misses (hits) in policy B approaches the competitive ratio in the limit.

Every policy is by definition 0-hit-competitive relative to every other policy. However, a policy may not be k-miss-competitive relative to another policy for any k. In that case, we will call it ∞-miss-competitive. For a policy A that is ∞-miss-competitive relative to policy B, the number of misses incurred in A cannot be bounded by the number of misses in B.

As noted before the notions of relative competitiveness are defined for individual cache sets. However, they can easily be lifted to set-associative caches, which can be seen as the composition of a number of cache sets.

6.2.2 Computing Bounds on Cache Performance

Assume policy P is k-miss-competitive with additive constant c relative to policy Q for which we have a cache analysis, i.e., an analysis that can compute upper bounds on the number of cache misses incurred by a task T. How can we obtain a sound upper bound on the number of cache misses incurred by T under P using the bound computed for Q?

To answer this question, we need to dig into what such an upper bound is. Let $S(T) \subseteq S$ be the set of possible access sequences performed by task T. Depending on its input T may exhibit different access sequences. If T's input is known $S(T)$ may be a singleton set. Then $\widehat{m_Q(T)}$ is an upper bound on the number of misses incurred by T under Q, if

$$\max_{s \in S(T)} \max_{q \in C^Q} m_Q(q, s) \leq \widehat{m_Q(T)}.$$

We quantify over all cache-set states $q \in C^Q$ of the policy because we usually have no knowledge about the cache-set state in which the execution of T begins[3]. We claim that $k \cdot \widehat{m_Q(T)} + c$ is an upper bound for P:

Theorem 6.5 (Global bounds)**.**
Let policy P be (k, c)-miss-competitive relative to policy Q, and let $\widehat{m_Q(T)}$ be an upper bound on the number of misses incurred by policy Q on task T. Then,

$$\max_{s \in S(T)} \max_{p \in C^P} m_P(p, s) \leq k \cdot \widehat{m_Q(T)} + c,$$

i.e., $k \cdot \widehat{m_Q(T)} + c$ is an upper bound on the number of misses incurred by P on task T.

[3]Do not confuse this with the initial state i^P of the policy. This is the state at startup of the hardware. When execution of T begins, other tasks have typically already been executed and thus modified the state.

Proof. We have defined every cache-set state in C^Q to be reachable (see Figure 2.4), i.e., for every state $q \in C^Q$ there is a sequence $s \in S$, such that $q = update_Q(i^Q, s)$. Observation $(*)$: This implies that every state $q \in C^Q$ is compatible with at least one state $p \in C^P$, i.e., $q \sim p$. Then,

$$
\begin{aligned}
& \max_{s \in S(T)} \max_{p \in C^P} m_P(p, s) \\
\overset{\text{Def. 6.2}}{\leq} \quad & \max_{s \in S(T)} \max_{p \in C^P} \max_{q \in C^Q, q \sim p} k \cdot m_Q(q, s) + c \\
= \quad & k \cdot (\max_{s \in S(T)} \max_{p \in C^P} \max_{q \in C^Q, q \sim p} m_Q(q, s)) + c \\
\overset{\text{Observation } (*)}{=} \quad & k \cdot (\max_{s \in S(T)} \max_{q \in C^Q} m_Q(q, s)) + c \\
\leq \quad & k \cdot \widehat{m_Q(T)} + c
\end{aligned}
$$

□

Lower bounds on the number of hits can be computed similarly using hit-competitiveness results.

We have made the assumption that nothing is known about the state of the policy T is starting in. If we do have such knowledge, we can make use of it: Let $I^P \subseteq C^P$ be the possible starting states. Then we can restrict the set of starting states for the cache analysis of Q to the subset $\{q \in C^Q \mid q \sim p, p \in I^P\}$ of C^Q.

6.2.3 Obtaining May and Must Analyses

As mentioned in the introduction, we can also build sound *may* and *must* cache analyses [Ferdinand et al., 1997,Ferdinand and Wilhelm, 1999] from competitiveness results. *may* cache analyses determine for each program point a superset of the set of memory blocks that may be in the cache when the control flow reaches this program point. Analogously, *must* cache analyses determine a subset of the set of memory blocks that are in the cache whenever the control flow reaches the program point. Results of a *must* analysis can be used to safely predict cache hits. Cache misses can be predicted using the results of a *may* analysis. Analyses are often not explicitly phrased as *may* or *must* analyses. Any analysis that safely predicts cache hits can be seen as a *must* analysis; any analysis that safely predicts cache misses can be seen as a *may* analysis.

As noted before, the notions of hit- and miss-competitiveness are in general not redundant. In the special case of 1-competitiveness, however, they are:

Theorem 6.6 (1-Competitiveness)**.**
Given two policies P and Q. The following two statements are equivalent:

(i) P is $(1, c)$-miss-competitive relative to Q.

(ii) P is $(1, c)$-hit-competitive relative to Q.

Proof. Trivial: $m_P(q,s) \leq 1 \cdot m_Q(q',s) + c \Leftrightarrow |s| - m_P(q,s) \geq |s| - m_Q(q',s) - c \Leftrightarrow h_P(q,s) \geq 1 \cdot h_Q(q',s) - c$. □

One can obtain sound *may* and *must* analyses from competitiveness results in the special case of 1-competitiveness:

Theorem 6.7 (*may* and *must* analyses)**.**
If policy P is $(1,0)$-miss-competitive relative to policy Q, then

(i) a must *analysis for Q is also a sound* must *analysis for P, and*

(ii) a may *analysis for P is also a sound* may *analysis for Q.*

Proof. For (i): As P is $(1,0)$-miss-competitive relative to Q, the contents of P always subsume the contents of Q. Memory blocks that are definitely in Q must also be in P. For (ii): Anything not contained in P cannot be in Q either. In other words, a cache miss in P must also be a cache miss in Q. □

6.2.4 Relation to Predictability Metrics

The predictability metrics of Chapter 5 are defined on access sequences with pairwise different accesses only. In contrast, relative competitiveness is defined on arbitrary accesses sequences. Therefore, one cannot draw any conclusions on the relative competitiveness of two policies based on their predictability metrics. However, one can derive constraints on the predictability metrics of two policies based on their relative competitiveness:

Theorem 6.8 (Relation to evict and mls)**.**
Let policy $P(k)$ be $(1,0)$-miss-competitive relative to policy $Q(l)$, then

(i) $evict^P_{\mathrm{HM}}(k) \geq evict^Q_{\mathrm{HM}}(l)$,

(ii) $mls^P(k) \geq mls^Q(l)$.

Proof. (i): Assume for a contradiction that $evict^P_{\mathrm{HM}}(k) < evict^Q_{\mathrm{HM}}(l)$. It is a simple consequence of Lemma 5.3 that for every $p \in C^{P(k)}$ and $s \in S^{\neq}$ such that $|s| = evict^P_{\mathrm{HM}}(k)$, $CC_{P(k)}(update_{P(k)}(p,s)) \subseteq SC(s)$. On the other hand, there must be a state $q \in C^{Q(l)}$ and a sequence $s \in S^{\neq}, |s| = evict^P_{\mathrm{HM}}(k)$, such that $CC_{Q(l)}(update_{Q(l)}(q,s)) \not\subseteq SC(s)$. Recall the following observation from the proof of Theorem 6.5: Every state $q \in C^{Q(l)}$ is compatible with at least one state $p \in C^{P(k)}$, i.e., $q \sim p$. Let $q \in C^{Q(l)}$ and $s \in S^{\neq}, |s| = evict^P_{\mathrm{HM}}(k)$ be such that $CC_{Q(l)}(update_{Q(l)}(q,s)) \not\subseteq SC(s)$. Let $p \in C^{P(k)}$ be compatible with q, i.e. $p \sim q$. Then $update_{P(k)}(p,s) \sim update_{Q(l)}(q,s)$. However, $CC_{Q(l)}(update_{Q(l)}(q,s)) \not\subseteq CC_{P(k)}(update_{P(k)}(p,s))$ contradicting the assumption that $P(k)$ is 1-miss-competitive relative to policy $Q(l)$ with additive constant 0.

(ii): Assume for a contradiction that $mls^P(k) < mls^Q(l)$. According to Lemma 5.5, there must be a state $p \in C^{P(k)}$ and a sequence $s \in S^{\neq}, |s| = mls^P(k) + 1$, such that $a \notin update_{P(k)}(p,s)$. Take a state $q \in C^{Q(l)}$ that is compatible with p, i.e. $p \sim q$. Since $mls^Q(l) \geq mls^P(k)$, $a \in update_{Q(l)}(q,s)$. Accessing a would thus yield a miss

in $update_{P(k)}(p, s)$, but not in $update_{Q(l)}(q, s)$. Since $update_{P(k)}(p, s) \sim update_{Q(l)}(q, s)$, this again contradicts the assumption that $P(k)$ is $(1, 0)$-miss-competitive relative to policy $Q(l)$. □

6.2.5 General Competitiveness Properties

Before describing how to automatically compute competitiveness values, we state a number of general properties of the definitions and of classes of policies.

In their fundamental work, [Mattson et al., 1970] introduce the *inclusion property* (stack property). A replacement policy F satisfies the inclusion property, if for any two instantiations $F(k)$ and $F(l)$ with different associativities k and l, the instantiation with the higher associativity always subsumes the contents of the instantiation with the lower associativity. LRU is an *inclusive* replacement policy, while PLRU and FIFO are not. For *inclusive* policies we can state the following theorem:

Theorem 6.9 (Inclusion property)**.**
For a replacement policy F that satisfies the inclusion property, if $l \geq k$, then

(i) $F(l)$ is $(1, 0)$-miss-competitive relative to $F(k)$,

(ii) $F(l)$ is $(1, 0)$-hit-competitive relative to $F(k)$.

Can a policy with lower associativity ever be competitive relative to one with a higher associativity?

Theorem 6.10 (Lower associativity)**.**
Let the policies P and Q have associativities l and m, respectively, with $l < m$. Then, P is neither k-miss- nor k-hit-competitive relative to Q for any k.

Proof. Observation: Since Q has a higher associativity, a cache set of Q always contains at least one element that is not contained in the corresponding cache set of P. We can construct an access sequence that accesses precisely these elements. □

Given that P is competitive relative to Q and Q competitive relative to R, then P is also competitive relative to R:

Theorem 6.11 (Transitivity of relative miss-competitiveness)**.**
Let P be (k, c)-miss-competitive relative to Q and let Q be (k', c')-miss-competitive relative to R. Then P is $(k{\cdot}k', k{\cdot}c' + c)$-miss-competitive relative to R.

Proof. We need to show that $m_P(p, s) \leq (k{\cdot}k') \cdot m_R(r, s) + (k{\cdot}c' + c)$ for all access sequences $s \in S$ and cache-set states $p \in C^P, r \in C^R$ that are compatible. If $p \in C^P$ is compatible with $r \in C^R$, then there is a sequence s, such that $p = update_P(i^P, s)$ and $r = update_R(i^R, s)$ by definition. Let $q = update_Q(i^Q, s)$. q is compatible with p and r by definition. By our assumptions, $m_P(p, s) \leq k{\cdot}m_Q(q, s){+}c$ and $m_Q(q, s) \leq k'{\cdot}m_R(r, s){+}c'$. Thus, $m_P(p, s) \leq k \cdot (k' \cdot m_R(r, s) + c') + c = (k{\cdot}k') \cdot m_R(r, s) + (k{\cdot}c' + c)$. □

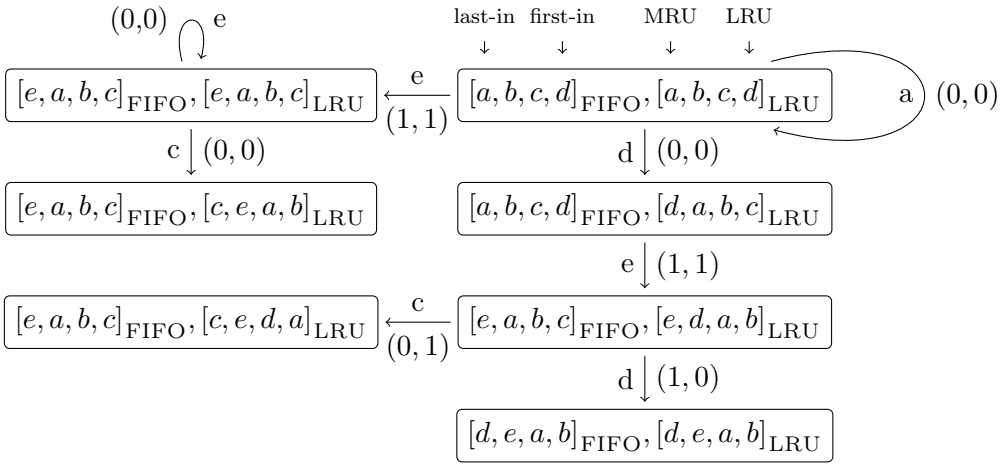


Figure 6.2: Running example. Small part of the transition system in the computation of competitiveness results for FIFO(4) vs LRU(4). In the LRU cache-set states, elements are ordered from most- to least-recently-used. In the FIFO cache-set states, they are ordered from last- to first-in. Transitions are labelled with the number of misses incurred. To explain the transitions, we have additionally labelled them with the corresponding accesses.

Theorem 6.12 (Transitivity of relative hit-competitiveness)**.**
Let P be (k,c)-hit-competitive relative to Q and let Q be (k',c')-hit-competitive relative to R. Then P is $(k{\cdot}k', k{\cdot}c'+c)$-hit-competitive relative to R.

Proof. There is a $q \in C^P$, such that $p \sim q$ and $q \sim r$ for every pair $p \in C^P, q \in C^Q$, $p \sim r$ as in the proof of Theorem 6.11. By our assumptions, $h_P(p,s) \geq k \cdot h_Q(q,s) - c$ and $h_Q(q,s) \geq k' \cdot h_R(r,s) - c'$. Thus, $h_P(p,s) \geq k \cdot (k' \cdot h_R(r,s) - c') - c = (k{\cdot}k') \cdot h_R(r,s) - (k{\cdot}c' + c)$. □

6.3 Computing Competitive Ratios

We have developed a tool that allows us to compute competitive ratios automatically. In the following we will describe our methodology.

The two policies P and Q under consideration induce a transition system that captures how the two policies act relative to each other, processing the same memory accesses. States of this transition system are pairs of cache-set states of policy P and policy Q. Figure 6.2 shows a small part of such a transition system. Competitive ratios and appropriate additive (subtractive) constants are properties of this system. To obtain competitive ratios, we roughly have to determine the maximum (minimum) ratio of misses (hits) in policy P relative to the number of misses (hits) in policy Q on any path through it.

The main obstacle is that there are infinitely many cache-set states if one assumes the set of memory blocks to be infinite. Although the set of memory blocks is finite in practice, the state space of the transition system is in most cases still prohibitively large. To overcome this problem we directly compute a finite quotient structure with respect to an equivalence relation on states that preserves competitiveness properties.

6.3.1 Induced Transition System

Let us formally define the transition system induced by a pair of policies P and Q.

Definition 6.13 (Induced transition system)**.**
Two policies P and Q induce a transition system $T_{P,Q} = (S_{P,Q}, R_{P,Q})$, where

$$S_{P,Q} = \{(p,q) \mid p \sim q, p \in C^P, q \in C^Q\} \subseteq C^P \times C^Q,$$

the states, are pairs of compatible cache-set states of policies P and Q.

$$\begin{aligned} T_{P,Q} \quad = \quad & \{((p,q),(m_p,m_q),(p',q')) \mid (p,q) \in S_{P,Q}, a \in M, \\ & (p',q') = (update_P(p,\langle a\rangle), update_Q(q,\langle a\rangle)), \\ & (m_p,m_q) = (m_P(p,\langle a\rangle), m_Q(q,\langle a\rangle))\} \end{aligned}$$

is the transition relation. Transitions are labelled with the number of misses (0 or 1) incurred by the access in the two cache-set states, respectively.

Competitiveness values depend on the number of misses (hits) on paths through the transition system:

Definition 6.14 (Path)**.**
A path *through a transition system $T = (S, R)$, where $R \subseteq S \times L \times S$ and L is a set of labels, is a sequence of labels $\pi = l_1 \dots l_n \in L^n$, such that*

$$\exists s_1, \dots, s_{n+1} \in S : \ \forall i \in \{1, \dots, n\} : \ (s_i, l_i, s_{i+1}) \in R.$$

The set of all paths *of a transition system T is denoted by $\Pi(T)$.*

In our case, labels are pairs (m_p, m_q). The definitions of hit- and miss-competitiveness translate directly to properties of paths of the induced transition system. A policy P is k-miss-competitive relative to policy Q with additive constant c, if

$$\sum_i \pi(i)_{|1} \leq k \cdot \sum_i \pi(i)_{|2} + c \text{ for every path } \pi \in \Pi(T_{P,Q}),$$

where $|1$ and $|2$ select the first and second component of a tuple, respectively. A policy P is k-hit-competitive relative to policy Q with subtractive constant c, if

$$\left(\sum_i 1 - \pi(i)_{|1}\right) \geq k \cdot \left(\sum_i 1 - \pi(i)_{|2}\right) - c \text{ for every path } \pi \in \Pi(T_{P,Q}).$$

Cache-Set States

In Section 2.2.1, we have introduced C^P, the set of reachable cache-set states of policy P. $C^P \subseteq M_\perp^k \times \mathbb{B}^l$, where k is the associativity of P and l is the number of status bits. In order to ease presentation, we adopt the following notation:

$$C_k^l \quad = \quad M_\perp^k \times \mathbb{B}^l$$

So $C^P \subseteq C_k^l$, where k is the associativity of P and l the number of status bits required. For instance, $C^{\mathrm{LRU}(k)} \subseteq C_k^0$, $C^{\mathrm{MRU}(k)} \subseteq C_k^k$, $C^{\mathrm{PLRU}(k)} \subseteq C_k^{k-1}$. PLRU can also be modeled without status bits as described in Section 5.7.

Accesses can have two effects on such states:

- The order of the elements in the tuple is changed, depending *only* on the *position* of the accessed memory block in the tuple and the status bits. In LRU, for instance, the elements are ordered from most to least recently used and no status bits are needed. In contrast, to represent states of MRU, we need k status bits.
- The position in the tuple of the element to be replaced is determined based on the status bits. In our example of LRU, elements are replaced at a fixed position, the right-most, i.e., the least-recently-used. In MRU, the position is determined by the first status bit being 0.

Cache-set states of all the policies that we consider in this thesis, i.e., LRU, PLRU, FIFO, and MRU, behave this way as well as all realistic policies we can imagine. The main point is that they do *not* base their replacement or update decisions on the particular memory block or its address.

6.3.2 Quotient Transition System

Constructing the induced transition system is not feasible. As noted above, it is either prohibitively large or even infinitely large, depending on the number of memory blocks one assumes. To make the analysis feasible, we construct a finite quotient structure, which has the same set of paths as the original system, but whose size is independent of the number of memory blocks.

To build this finite quotient transition system, we rely on the following property, that is satisfied by policies representable in the above way and all other cache replacement policies we are aware of: Let $h : M \to M$ be a bijective renaming of the memory blocks and h^* the point-wise extension of h to cache-set states, that additionally maps $\perp$ (empty cache lines) to $\perp$. Let $q \in C^P$, then

$$update_P(h^*(q), \langle h(a) \rangle) = h^*(update_P(q, \langle a \rangle)), \tag{6.1}$$

i.e., isomorphic cache-set states behave the same. Obviously,

$$m_P(h^*(q), \langle h(a) \rangle) = m_P(q, \langle a \rangle), \tag{6.2}$$

because $h(a)$ is contained in $h^*(q)$ if and only if a is contained in q.

$$\begin{array}{ccc} [a,b,c]_{\mathrm{LRU}}, [b,\bot,e]_{\mathrm{FIFO}} & \approx_h & [f,g,l]_{\mathrm{LRU}}, [g,\bot,m]_{\mathrm{FIFO}} \\ c \Big\downarrow (0,1) & & h(c) = l \Big\downarrow (0,1) \\ [c,a,b]_{\mathrm{LRU}}, [c,b,\bot]_{\mathrm{FIFO}} & \approx_h & [l,f,g]_{\mathrm{LRU}}, [l,g,\bot]_{\mathrm{FIFO}} \end{array}$$

Figure 6.3: In this example, we assume LRU and FIFO replacement. $q_1 = [a,b,c]_{\mathrm{LRU}}, [b,\bot,e]_{\mathrm{FIFO}} \approx_h q_2 = [f,g,l]_{\mathrm{LRU}}, [g,\bot,m]_{\mathrm{FIFO}}$ with an appropriate h-function. An access to c yields a hit in the first part of q_1 and a miss in the second part. The transition is therefore labelled with $(0,1)$. Accessing $h(c) = l$ on q_2 has the same effect in terms of hits and misses. Also, the two resulting states are in the $\approx$ relation by the same function h.

This means that the particular contents of a pair of cache-set states are irrelevant for the *possible* future ratio of misses. What is important is the relation between the two cache-set states, i.e., the relative positions of elements contained in both sets. For instance, take the two pairs of cache-set states $q_1 = [a,b,c]_{\mathrm{LRU}}, [b,\bot,e]_{\mathrm{FIFO}}$ and $q_2 = [f,g,l]_{\mathrm{LRU}}, [g,\bot,m]_{\mathrm{FIFO}}$. q_1 and q_2 are different regarding the contents of the cache-set states. Yet, we do not want to distinguish the two states, as they will show the same relative hit/miss behavior, albeit on different access sequences. Recall that our relative competitiveness definitions quantify over all access sequences.

Definition 6.15 ($\approx$-Equivalence)**.**
Let $h : M \to M$ be a bijective renaming of memory blocks, and $h^{\#}$ be the point-wise extension of h to pairs *of cache-set states. We say that two states p and q are* $\approx$-equivalent*, denoted $p \approx q$, if they can be transformed into each other by renaming their contents:*

$$p \approx q :\Leftrightarrow \exists h : M \to M . q = h^{\#}(p)$$

To indicate a particular feasible renaming function h we also write $p \approx_h q$, if $q = h^{\#}(p)$.

The rationale behind identifying two states $q \approx_h q'$ is that given any access $a \in M$ on q, $h(a)$ will have the "same" effect on q':

$$q \approx_h q' \Longrightarrow \left\{ \begin{array}{rcl} m_{P,Q}(q, \langle a \rangle) & = & m_{P,Q}(q', \langle h(a) \rangle) \\ update_{P,Q}(q, \langle a \rangle) & \approx_h & update_{P,Q}(q', \langle h(a) \rangle) \end{array} \right. \tag{6.3}$$

This follows directly from Equation 6.1 and Equation 6.2. Figure 6.3 illustrates Equation 6.3 with the two example states q_1, q_2 given above.

Definition 6.16 (Quotient transition system)**.**
A transition system $T = (S, R)$, where $R \subseteq S \times L \times S$, and an equivalence relation $\equiv \subseteq S \times S$ on the states of T induces a quotient transition system $T/{\equiv} = (S/{\equiv}, R/{\equiv})$,

where

$$S/{\equiv} \quad \textit{is a set of unique representatives of the equivalence classes of } S \textit{ with respect to } \equiv,\ S/{\equiv} \subseteq S,$$

$$R/{\equiv} \quad = \quad \{(\overline{s}, l, \overline{t}) \mid \overline{s}, \overline{t} \in S/{\equiv}, \exists s, t \in S : (s, l, t) \in R, s \equiv \overline{s}, t \equiv \overline{t}\}.$$

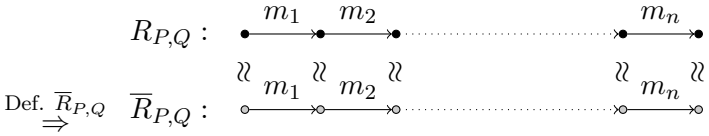
There is a transition $(\overline{s}, l, \overline{t})$ between two representatives iff there is a transition (s, l, t) between two states s and t that are represented by $\overline{s}$ and $\overline{t}$, respectively.

The relation $\approx$ defines an equivalence on states. It can therefore be used to partition the states of $S_{P,Q}$ into equivalence classes. This induces the quotient transition system $\overline{T}_{P,Q} = (\overline{S}_{P,Q}, \overline{R}_{P,Q}) = T/{\approx}$. Figure 6.4 shows which states of our running example are equivalent and the resulting quotient structure.

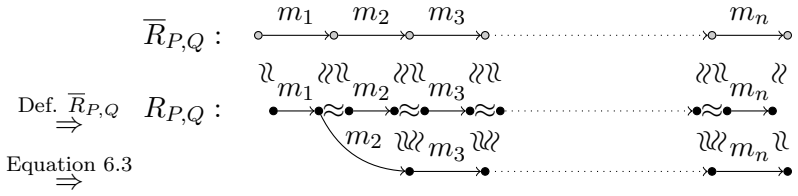
By the following theorem we can safely work with the quotient structure $\overline{T}_{P,Q}$ instead of $T_{P,Q}$ when computing competitiveness values.

Theorem 6.17 (Path equivalence)**.**
The transition systems $T_{P,Q}$ and $\overline{T}_{P,Q}$ are path equivalent, i.e., $\Pi(T_{P,Q}) = \Pi(\overline{T}_{P,Q})$.

Proof. We need to show $\pi = m_1 \dots m_n \in \Pi(T_{P,Q}) \Leftrightarrow \pi \in \Pi(\overline{T}_{P,Q})$.
"$\Rightarrow$" follows from the definition of $\overline{R}_{P,Q}$: Let $s_1, \dots, s_{n+1}$ be such that $(s_i, m_i, s_{i+1}) \in R_{P,Q}, \forall i \in \{1, \dots, n\}$. Let $\overline{s_1}, \dots, \overline{s_{n+1}}$ be the representatives of $s_1, \dots, s_{n+1}$ in $\overline{S}_{P,Q}$. Then $(\overline{s_i}, m_i, \overline{s_{i+1}}) \in \overline{R}_{P,Q}, \forall i \in \{1, \dots, n\}$:



"$\Leftarrow$" is just a little more complicated. For each of the edges $(\overline{s_i}, m_i, \overline{s_{i+1}})$ in $\overline{R}_{P,Q}$ there is a corresponding edge in $R_{P,Q}$ by definition. By Equation 6.3 we can construct a path through $R_{P,Q}$ from these edges as illustrated below:



□

Observation. *The set of equivalence classes of $\approx$ is finite, as it only depends on the relative positions of elements contained in both sets and the cache sets' status bits. In particular, the index of $\approx$ is independent of the number of memory blocks.*

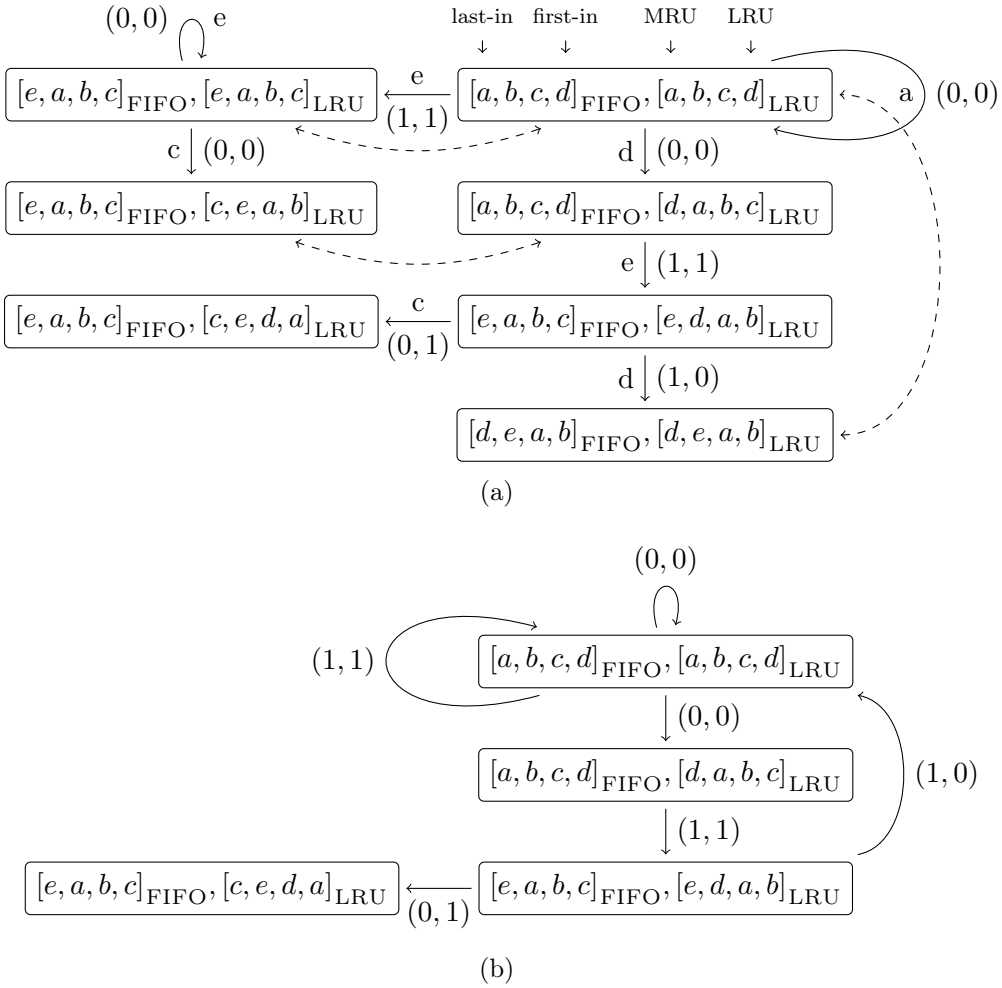


Figure 6.4: Running example revisited. In (a), states that are equivalent according to $\approx$ are connected by dashed lines. "Merging" equivalent states yields the quotient structure depicted in (b).

Therefore, the quotient transition system $\overline{T}_{P,Q}$ is finite. Note that we directly construct $\overline{T}_{P,Q}$, in particular we never construct the underlying transition system $T_{P,Q}$.

Theorem 6.18 (Index of $\approx$)**.**
The index $|(C_k^l \times C_{k'}^{l'})/\approx|$ *of* $\approx$ *is*

$$|(C_k^l \times C_{k'}^{l'})/\approx| = \underbrace{2^{l+l'}}_{\substack{\text{status bits}\\ \text{of } P \text{ and } Q}} \cdot \underbrace{\sum_{i=0}^{k}\binom{k}{i}}_{\text{non-empty lines in } P} \cdot \underbrace{\sum_{i'=0}^{k'}\binom{k'}{i'}}_{\text{non-empty lines in } Q} \cdot \underbrace{\sum_{j=0}^{\min\{i,i'\}}\binom{i}{j}\binom{i'}{j}j!}_{\substack{\text{number of overlappings}\\ \text{in non-empty lines}}}$$

This can be bounded by

$$2^{l+l'+k+k'} \leq |(C_k^l \times C_{k'}^{l'})/\approx| \leq 2^{l+l'+k+k'} \cdot \underbrace{e \cdot k! \cdot k'!}_{\text{bound on number of overlappings}}$$

Proof. If two states differ in their status bits, they cannot be equivalent. This explains the factor $2^{l+l'}$. To be equivalent, empty lines have to be in the same positions of the tuples. Therefore we sum over all possible configurations of empty lines. For a given configuration of empty lines, the contents of the two cache sets may overlap in 0 to $\min\{i,i'\}$ positions, if i and i' are the numbers of non-empty lines in the two cache sets. The j overlapping lines may be in $\binom{i}{j}$ different positions in the first set and in $\binom{i'}{j}$ different positions in the second set. There can be $j!$ orderings of these elements in the second set relative to the first one (and vice versa).

The upper bound can be explained by considering the innermost factor $\sum_{j=0}^{\min\{i,i'\}}\binom{i}{j}\binom{i'}{j}j!$. Expansion of the binomial coefficients and simplification yields $\sum_{j=0}^{\min\{i,i'\}}\frac{i!}{(i-j)!\cdot j!}\cdot\frac{i'!}{(i'-j)!}$. i and i' are bounded by k and k', respectively, which yields $k!\cdot k'!\cdot\sum_{j=0}^{\min\{k,k'\}}\frac{1}{(k-j)!\cdot j!\cdot(k'-j)!}$. Finally, the sum can be bounded by $\sum_{j=0}^{\infty}\frac{1}{(k-j)!\cdot j!\cdot(k'-j)!} \leq \sum_{j=0}^{\infty}\frac{1}{j!} = e$.

With the innermost factor bounded by $e \cdot k! \cdot k'!$, the two surrounding sums can be simplified by the binomial theorem to $2^k \cdot 2^{k'}$. The lower bound is trivial with 1 being a lower bound for the innermost factor. □

So $|(C_k^l \times C_{k'}^{l'})/\approx|$ grows exponentially with the associativities k, k' and status bits l, l'. Still, associativities up to 8 for both policies are usually tractable. This is for several reasons. Firstly, most policies do not require status bits. Their state can be fully represented by the permutation of the memory blocks. Status bits were only required for MRU. Secondly, the set of reachable states C^P of a policy P with associativity k and l status bits is a proper subset of C_k^l, i.e. some of the states in C_k^l are not reachable by policy P. In LRU and FIFO, for instance, cache-set states can only be filled from left to right, the state $[\bot, a, \bot, c]$ is not reachable. Thirdly, and most importantly, usually only a fraction of $(C^P \times C^Q)/\approx \subseteq (C_k^l \times C_{k'}^{l'})/\approx$ is actually reachable, i.e., in $\overline{S}_{P,Q}$. The more similar the two policies P and Q behave, the fewer *compatible* pairs of cache-set states are reachable. For instance, $|\overline{S}_{\text{PLRU(8)},\text{LRU(5)}}| = 576$, whereas $|(C_8^0 \times C_5^0)/\approx| = 1129472$. In contrast, $|\overline{S}_{\text{PLRU(8)},\text{FIFO(5)}}| = 30808$, which is still way below $|(C_8^0 \times C_5^0)/\approx|$.

Computing Unique Representatives

In order to construct the quotient transition system on-the-fly, we have to compute unique representatives of the states that we encounter in the construction of the transition system. Given a well-founded total order $\leq_{M_\perp} \subseteq M_\perp \times M_\perp$ on memory blocks M and empty lines, such that $\perp$ is the least element of $\leq_{M_\perp}$, we define a well-founded partial order $\leq_{S_{P,Q}} \subseteq S_{P,Q} \times S_{P,Q}$ on the states in $S_{P,Q}$:

$$(p, q) \leq_{S_{P,Q}} (p', q') :\Leftrightarrow p <_{C_k^l} p' \vee (p = p' \wedge q \leq_{C_{k'}^{l'}} q'),$$

where cache-set states in $C^P \subseteq C_k^l$ and $C^Q \subseteq C_{k'}^{l'}$ are as well ordered lexicographically based on $\leq_{M_\perp}$:

$$[m_1, \ldots, m_k]_{b_1 \ldots b_l} \leq_{C_k^l} [m'_1, \ldots, m'_k]_{b'_1 \ldots b'_l} \quad :\Leftrightarrow \quad \exists i > 0 : \forall j < i : m_j = m'_j \wedge m_i \leq_{M_\perp} m'_i \\ \wedge \forall i : b_i = b'_i.$$

Note that $\leq_{S_{P,Q}}$ is only partial in general as the status bits have to agree for a pair of states to be in $\leq_{S_{P,Q}}$. However, it is total on $\approx$-equivalent states. Therefore, there is always a *least* state in an equivalence class.

We choose the least state of an equivalence class in this order as the representative of its equivalence class:

$$\overline{S}_{P,Q} := \{s_{P,Q} \in S_{P,Q} \mid \forall s'_{P,Q} : s_{P,Q} \approx s'_{P,Q} \Rightarrow s_{P,Q} \leq_{S_{P,Q}} s'_{P,Q}\}$$

These unique representatives can be efficiently computed by constructing a renaming function h traversing a state from left to right and choosing the minimal memory block still available for the renaming:

Example. *Consider the total order $\leq_M$ to order memory blocks alphabetically: $\perp \leq_{M_\perp} a \leq_{M_\perp} b \leq_{M_\perp} c \leq_{M_\perp} \ldots$. Then, the representative of $[g, f, \perp, h], [g, h, l, \perp]$ can be computed by consecutively renaming the elements of the state:*

$$\begin{aligned} [g, f, \perp, h], [g, h, l, \perp] &\rightsquigarrow [a, f, \perp, h], [a, h, l, \perp] \\ \rightsquigarrow [a, b, \perp, h], [a, h, l, \perp] &\rightsquigarrow [a, b, \perp, c], [a, c, l, \perp] \\ \rightsquigarrow [a, b, \perp, c], [a, c, d, \perp] & \end{aligned}$$

Building the Quotient Transition System

The quotient transition system can be incrementally computed by Algorithm 1. The algorithm proceeds by taking a yet *unprocessed* normalized state from the *Unprocessed* queue and by computing all its normalized successor states until all states have been processed. It starts with the pair of compatible states (i^P, i^Q). Pairs of cache-set states are normalized by NORMALIZE(p, q) as described in the previous paragraph.

The key insight is that NORMALIZE$(update_P(p, \langle a \rangle), update_Q(q, \langle a \rangle))$ is equal for all $a \notin CC_P(p) \cup CC_Q(q)$. All of these accesses will be misses in both p and q and thus result

in $\approx$-equivalent successor states. Therefore, it is sufficient to compute successors under the finite number of accesses $CC_P(p) \cup CC_Q(q) \cup \{\textsc{SelectOne}(\overline{CC_P(p) \cup CC_Q(q)})\}$, where $\textsc{SelectOne}(S)$ selects one of the memory blocks in S. The computation of the set of transitions is straightforward.

Algorithm 1: Building Quotient Transition System

Input: Policies P, Q
Output: Quotient Transition System $\overline{T}_{P,Q} = (\overline{S}_{P,Q}, \overline{R}_{P,Q})$
begin
  $\overline{S}_{P,Q} \leftarrow \{\textsc{Normalize}(i^P, i^Q)\}$
  $\overline{R}_{P,Q} \leftarrow \emptyset$
  $Unprocessed \leftarrow [\textsc{Normalize}(i^P, i^Q)]$
  while $\neg\textsc{Empty}(Unprocessed)$ **do**
    $(p, q) \leftarrow \textsc{Pop}(Unprocessed)$
    foreach $a \in CC_P(p) \cup CC_Q(q) \cup \{\textsc{SelectOne}(\overline{CC_P(p) \cup CC_Q(q)})\}$ **do**
      $(p', q') \leftarrow \textsc{Normalize}(update_P(p, \langle a \rangle), update_Q(q, \langle a \rangle))$
      $(m_p, m_q) \leftarrow (m_P(p, \langle a \rangle), m_Q(q, \langle a \rangle))$
      $\overline{R}_{P,Q} \leftarrow \overline{R}_{P,Q} \cup \{((p, q), (m_p, m_q), (p', q'))\}$
      if $(p', q') \notin \overline{S}_{P,Q}$ **then**
        $\textsc{Push}(Unprocessed, (p', q'))$
        $\overline{S}_{P,Q} \leftarrow \overline{S}_{P,Q} \cup \{(p', q')\}$
end

6.3.3 Computation of Competitive Ratios

Once we have built up the quotient transition system, determining the minimal k such that P is hit- (miss-) competitive relative to Q amounts to computing the *minimum (maximum) cycle ratio* [Lawler, 1966, Ahuja et al., 1993].

In the setting of miss-competitiveness, we wish to find a cycle through the quotient transition system that maximizes the ratio of misses in P relative to the misses in Q. Maximum ratio problems can easily be converted into minimum ratio problems by changing the sign of the numerator or the denominator.

The *minimum cycle ratio* problem, also known as the *minimum cost-to-time ratio cycle problem*, is the following: Given a directed graph G with both a cost and a travel time associated with each edge, we wish to find a cycle in the graph with the smallest ratio of its cost to its travel time.

Definition 6.19 (Minimum cycle ratio)**.**
The minimum cycle ratio λ^* of G is

$$\lambda^* = \min_{Cycle\ C \in G} \frac{\sum_{Edge\ (i,j) \in C} c_{ij}}{\sum_{Edge\ (i,j) \in C} \tau_{ij}},$$

where c_{ij} and τ_{ij} are the cost and travel time associated with edge (i, j).

[Lawler, 1966,Ahuja et al., 1993] describe how to solve the minimum cycle ratio problem by repeated applications of the negative cycle detection algorithm. Their algorithm relies on the following observation: Let λ^* be the minimum cycle ratio of a graph G, then

$$\min_{\text{Cycle } C \in G} \sum_{\text{Edge } (i,j) \in C} c_{ij} - \lambda^* \tau_{ij} = 0.$$

Based on this observation, one can do a binary search for λ^*. If the graph G^λ with edge lengths $l_{ij} = c_{ij} - \lambda \tau_{ij}$ contains negative cycles, then $\lambda^* < \lambda$. Otherwise, if all cycles have positive length, $\lambda^* > \lambda$. As one can bound the numerators and denominators of any cycle ratio in the system[4], one can terminate the binary search if under these constraints only a single rational number may reside in the computed interval. This way the exact minimum cycle ratio is obtained.

As noted above, we need to compute the maximum cycle ratio of $\overline{T}_{P,Q}$ to obtain the competitive miss ratio:

Theorem 6.20 (Maximum cycle ratio)**.**
The maximum cycle ratio k of $\overline{T}_{P,Q}$, where the cost associated with a transition is the number of misses incurred by P and the associated travel time is the number of misses incurred by Q, is equal to the competitive miss ratio of P relative to Q.

Proof. We need to show that

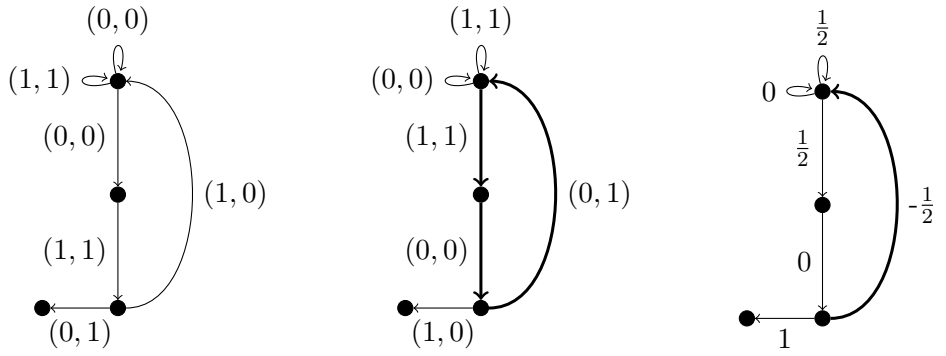
1. P is k-miss-competitive relative to Q with some additive constant c.
2. P is not k'-miss-competitive relative to Q with any additive constant c' for $k' < k$.

For 1. we need to show

$$\sum_i \pi(i)_{|1} \leq k \cdot \sum_i \pi(i)_{|2} + c \text{ for every path } \pi \in \Pi(T_{P,Q}) = \Pi(\overline{T}_{P,Q}).$$

Any path π can be split into three (possibly empty) parts $\pi = \pi_0\pi_1\pi_2$, such that π_0 and π_2 correspond to acyclic traversals of the state space $\overline{S}_{P,Q}$ and π_1 corresponds to a cycle in $\overline{S}_{P,Q}$. Since $\overline{S}_{P,Q}$ is finite, $|\pi_0| < |\overline{S}_{P,Q}|$ and $|\pi_2| < |\overline{S}_{P,Q}|$. For the cyclic part π_1 we know that $\sum_i \pi_1(i)_{|1} \leq k \cdot \sum_i \pi_1(i)_{|2}$. The acyclic paths π_0 and π_2 can be "covered" by an appropriate constant $c \leq 2 \cdot |\overline{S}_{P,Q}|$.

[4]By the number of states in the transition system.



(a) Part of the Quotient Structure $\overline{T}_{\text{FIFO(4),LRU(4)}}$

(b) Labels reflect hits instead of misses. Minimum cycle ratio $\frac{1+0+0}{1+0+1} = \frac{1}{2}$.

(c) Shortest path, w.r.t. edge lengths $w = h_1 - \frac{1}{2}h_2$. Cycle has length 0.

Figure 6.5: Running example. Hit-competitiveness of FIFO(4) vs LRU(4). FIFO(4) is $\frac{1}{2}$-hit-competitive relative to LRU(4) with subtractive constant $\frac{3}{2}$. The small part of $\overline{T}_{\text{FIFO(4),LRU(4)}}$ does not contain an acyclic path of length $-\frac{3}{2}$. The shortest path, in the example, is only of length $-\frac{1}{2}$.

For 2.: Choose a "cyclic" path π that corresponds to the maximal cycle ratio k. As $k > k'$, $\sum_i \pi(i)_{|1} = k \cdot \sum_i \pi(i)_{|2} > k' \cdot \sum_i \pi(i)_{|2}$. By repeating π sufficiently often $\sum_i \pi^n(i)_{|1} > k' \cdot \sum_i \pi^n(i)_{|2} + c'$ for any additive constant c'. □

The proof shows that the additive constant c can only stem from finite acyclic pre- or suffixes of paths. Once the minimum cycle ratio k has been determined, finding the appropriate additive (subtractive) constant c reduces to computing the length of the shortest path through $\overline{T}_{P,Q}$, where the edge weights w are chosen as $c_{ij} - k\tau_{ij}$. As k is the minimum cycle ratio, the graph has no negative cycles. Paths with negative weight correspond to situations where P can do "worse" than suggested by k relative to Q for a limited number of steps. As an example, assume P to be 2-miss-competitive relative to Q. Then, there must be paths of length one, such that P incurs a miss and Q does not.

Similarly, the competitive hit ratio is the maximum cycle ratio of $\overline{T}_{P,Q}$, where the cost associated with a transition is the number of hits by P and the associated travel time is the number of hits by Q.

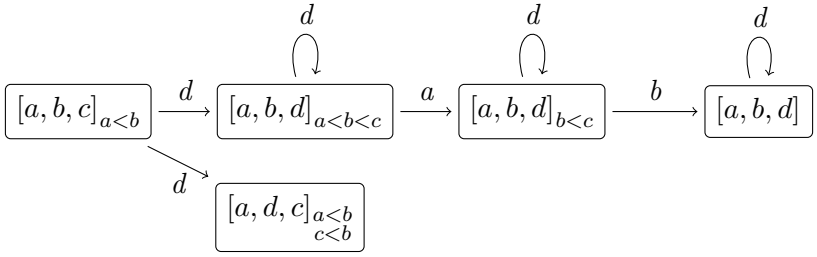
In Figure 6.5, we illustrate the two steps of our algorithm for our running example and the case of hit-competitiveness. To improve readability of the example, we omit the node labels. In the course of computing competitiveness values, we generate example access sequences and start states that correspond to the minimum cycle ratio k and the constant c. This may help to understand results and to identify patterns, that lead to more general analytical theorems.

6.3.4 Competitiveness Relative to OPT

It is also possible to compute the competitiveness of an online policy P relative to OPT, the optimal offline policy. Except for the treatment of compatible states, this yields competitiveness results in the sense of the definition of [Sleator and Tarjan, 1985]. To obtain competitiveness results in the precise sense of Sleator and Tarjan, one would have to consider $(C^P \times C^{\mathrm{OPT}(k)})/\approx$ instead of $S_{P,\mathrm{OPT}(k)}$, which is possible in principle but yields more expensive analyses.

OPT is fundamentally different from online policies, as it bases its decisions on *future* memory accesses rather than on memory accesses in the *past*. OPT replaces the memory block that will not be accessed for the longest time in the future. It is thus not possible to define a deterministic *update*-function for OPT because it would have to guess future events. However, it is possible to capture the behavior of OPT by introducing non-determinism whenever it "does not know" which memory block to replace. Each of the possible replacements is then based on an assumption about the future. To enforce this assumption, one needs to augment each OPT-state with a set of *future constraints*. These *future constraints* are strict partial orders that constrain the order in which future memory accesses are expected.

Example. *Consider the following portion of the* OPT*-state space in which the* future constraints *are attached to the contents of the* OPT*-states:*



In the first state $[a,b,c]_{a<b}$*, a is constrained to be accessed earlier than b in the future. If d is accessed in this state,* OPT *cannot replace a, as it knows that b will definitely be accessed later than a. It can only replace b or c. If it replaces b, the next access to c must precede the next access to b. Therefore, the constraint* $c < b$ *is introduced. Conversely, if it replaces c, the next access to b must precede the next access to c. Introducing the constraint* $b < c$ *results in the partial order* $a < b < c$*.* OPT *can only make transitions on accesses to memory blocks that are minimal according to the attached partial order. In* $[a,b,d]_{a<b<c}$ *it is not possible to access b or c. Upon an access to a, a is removed from the partial order. d is unconstrained, therefore hits on d do not change the state.*

The initial state of OPT is $i^{\mathrm{OPT}(k)} = [\bot,\ldots,\bot]_{\emptyset}$. The non-deterministic $update_{\mathrm{OPT}(k)} : C^{\mathrm{OPT}(k)} \times M \to \mathcal{P}(C^{\mathrm{OPT}(k)})$ as described above can be formalized by

$$update_{\mathrm{OPT}(k)}([a_1,\ldots,a_k]_{<_{fut}},c) :=$$

$$\begin{cases} \emptyset & \text{if } \exists b : b <_{fut} c \\ \{[a_1,\ldots,a_k]_{<_{fut}\setminus\{(c,a)|a\in M\}}\} & \text{else if } \exists i : a_i = c \\ \{[a_1,\ldots,a_{i-1},c,a_{i+1},\ldots,a_k]_{<_{fut}} \mid a_i = \bot\} & \text{else if } \exists i : a_i = \bot \\ \{[a_1,\ldots,a_{i-1},c,a_{i+1},\ldots,a_k]_{<_{fut}\setminus\{(c,a)|a\in M\}\cup\{(a_j,a_i)|j\neq i\}} \mid \forall j \neq i : a_i \not<_{fut} a_j\} & \text{else} \end{cases}$$

where $C^{\mathrm{OPT}(k)} := M_\bot^k \times \mathcal{P}(M \times M)$. (Case 1): If the accessed element c is not minimal in the order $<_{fut}$, the access is not possible. (Case 2): If c causes a hit, all constraints related to c are dropped as they were satisfied by this access. (Case 3): If c incurs a miss and the set is not full, c is inserted into the set[5]. (Case 4): If c incurs a miss, and the set is full, each maximal element among $\{a_1,\ldots,a_k\}$ in the order might be replaced. If element a_i is replaced, constraints are added to ensure that a_i is indeed accessed last among $\{a_1,\ldots,a_k\}$.

$update_{\mathrm{OPT}}$ can be lifted to access sequences in the obvious way:

$$\begin{aligned} update_{\mathrm{OPT}(k)}(p,\langle\rangle) &= \{p\} \\ update_{\mathrm{OPT}(k)}(p,\langle a\rangle \circ s) &= \bigcup_{p'\in update_{\mathrm{OPT}(k)}(p,a)} update_{\mathrm{OPT}(k)}(p',s) \end{aligned}$$

Note that $update_{\mathrm{OPT}(k)}([a_1,\ldots,a_k]_{<_{fut}},s)$ does not get "stuck" as long as s does not contradict $<_{fut}$: The disjunction of the constraints that are added in the case of a miss $\bigvee_i \bigwedge_{j\neq i} a_j < a_i$ is a tautology. Therefore, there is always at least one successor whose constraints are satisfied by the rest of the sequence.

Due to the non-determinism of $update_{\mathrm{OPT}(k)}$, the definition of the transition relation of the induced transition system $T_{P,\mathrm{OPT}(k)}$ changes slightly:

$$\begin{aligned} R_{P,\mathrm{OPT}(k)} = \{&((p,q),(m_p,m_q),(p',q')) \mid (p,q) \in S_{P,\mathrm{OPT}(k)}, a \in M, \\ &(p',q') \in \{update_P(p,\langle a\rangle)\} \times update_{\mathrm{OPT}(k)}(q,\langle a\rangle), \\ &(m_p,m_q) = (m_P(p,\langle a\rangle), m_{\mathrm{OPT}(k)}(q,\langle a\rangle))\} \end{aligned}$$

The notion of *compatibility* needs to be slightly adapted as well: Two cache-set states $p \in C^P$ and $q \in C^{\mathrm{OPT}(k)}$ are called compatible, denoted $p \sim q$, iff there is some access sequence $s \in S$, such that $p = update_P(i^P, s)$ and $q \in update_{\mathrm{OPT}(k)}(i^{\mathrm{OPT}(k)}, s)$.

The *future constraints* in the OPT-states may grow arbitrarily large. However, constraints concerning memory blocks that are neither contained in the OPT-state nor in the state of the online policy P can be dropped in the quotient transition system. Dropping these constraints does not change the observable behavior of $\overline{T}_{P,\mathrm{OPT}(k)}$ regarding hits and misses: For each "constrained" memory block that would miss in both OPT- and P-state there are many yet unaccessed and thus "unconstrained" memory blocks that result in the same behavior anyway.

[5] As the set is not full, no replacements have yet taken place. So there cannot be any constraints.

6.4 Results

We have implemented the method in Java. Running our tool on a Core 2 Duo E6750 with 2GB of RAM, we have obtained a vast amount of competitiveness results for LRU, FIFO, MRU PLRU, and OPT at associativities ranging from 2 to 8. To avoid any paging activity, we have limited the heap space to be allocated by our tool to 1.5GB. If the computation required more than 30 minutes or exceeded the heap space limit of 1.5GB, we report MEM or TIME, respectively.

In this section, we will present the most interesting results. The full set of results is included in the appendix.

We also understand our tool as a means to come up with more general conjectures about relative competitiveness of policies. Competitiveness results for several associativities often reveal a pattern that may then be generalized by the user. In the following description we will point out such cases.

6.4.1 Miss-Competitiveness

Figure 6.6 depicts relative miss-competitiveness results among FIFO, PLRU, and LRU if compared at the same level of associativity. One observation is that LRU(k) is k-miss-competitive relative to FIFO(k) and vice-versa for all of the investigated associativities. The same holds for FIFO(k) vs PLRU(k), and FIFO(k) vs MRU(k), but not for PLRU(k) vs FIFO(k), and MRU(k) vs FIFO(k). For $k \leq 8$, MRU(k) is $(k-1, k-2)$-miss-competitive relative to LRU(k) and vice versa.

By [Sleator and Tarjan, 1985], FIFO(k) and LRU(k) are k-miss-competitive relative to the optimal offline policy OPT. This implies at least k-miss-competitiveness relative to any online algorithm. Conversely, as MRU(k) and PLRU(k) are not k-miss-competitive relative to FIFO(k), they cannot be k-competitive relative to OPT(k).

PLRU(2) and LRU(2) are 1-miss-competitive relative to each other, as LRU and PLRU coincide for associativity 2. In contrast, PLRU(4) and PLRU(8) are not k-miss-competitive relative to LRU(4) and LRU(8), respectively, for any k. This is particularly interesting as it has been suggested in [Hergenhan and Rosenstiel, 2000] to model a PLRU(k)-cache by an LRU(k)-cache to simplify WCET prediction, as it "does not add a significant error."

It is also interesting to compare policies of different associativities. We observe the following for PLRU vs LRU:

k	2	4	8	16	32
l	2	3	4	5	6
PLRU(k) vs LRU(l)	$1,0$	$1,0$	$1,0$	$1,0$	$1,0$

Generalizing this, we conjectured that PLRU(k) is $(1,0)$-miss-competitive relative to LRU($1 + log_2 k$).

Associativity:	2	3	4	5	6	7	8
LRU vs FIFO	$2,1$	$3,2$	$4,3$	$5,4$	$6,5$	$7,6$	$8,7$
LRU vs PLRU	$1,0$	–	$2,1$	–	–	–	$5,4$
LRU vs MRU	$1,0$	$2,1$	$3,2$	$4,3$	$5,4$	$6,5$	$7,6$
FIFO vs LRU	$2,1$	$3,2$	$4,3$	$5,4$	$6,5$	$7,6$	$8,7$
FIFO vs PLRU	$2,1$	–	$4,4$	–	–	–	$8,8$
FIFO vs MRU	$2,1$	$3,3$	$4,4$	$5,5$	$6,6$	MEM	MEM
PLRU vs LRU	$1,0$	–	∞	–	–	–	∞
PLRU vs FIFO	$2,1$	–	∞	–	–	–	∞
PLRU vs MRU	$1,0$	–	∞	–	–	–	MEM
MRU vs LRU	$1,0$	$2,1$	$3,2$	$4,3$	$5,4$	$6,5$	$7,6$
MRU vs FIFO	$2,1$	$4,3$	$6,5$	$8,7$	$10,9$	MEM	MEM
MRU vs PLRU	$1,0$	–	$4,3$	–	–	–	MEM

Figure 6.6: Miss-Competitiveness ratios k and additive constants c relating FIFO, PLRU, LRU, and MRU at the same associativity. PLRU is only defined for powers of two. As an example of how this should be read, LRU(4) is 2-miss-competitive relative to PLRU(4) with additive constant 1, whereas PLRU(4) is not miss-competitive relative to LRU(4) at all. ∞ indicates that there is no k such that the policy on the left is k-miss-competitive relative to the policy on the right. MEM indicates that the computation required more than 1.5GB of heap space.

Associativity:	2	3	4	5	6	7	8
LRU vs FIFO	0	0	0	0	0	0	0
LRU vs PLRU	$1,0$	–	$\frac{1}{2},1$	–	–	–	$\frac{1}{8},\frac{15}{8}$
LRU vs MRU	$1,0$	0	0	0	0	0	0
FIFO vs LRU	$\frac{1}{2},\frac{1}{2}$	$\frac{1}{2},1$	$\frac{1}{2},\frac{3}{2}$	$\frac{1}{2},2$	$\frac{1}{2},\frac{5}{2}$	$\frac{1}{2},3$	$\frac{1}{2},\frac{7}{2}$
FIFO vs PLRU	$\frac{1}{2},\frac{1}{2}$	–	$\frac{1}{4},\frac{5}{4}$	–	–	–	$\frac{1}{11},\frac{19}{11}$
FIFO vs MRU	$\frac{1}{2},-\frac{1}{2}$	0	0	0	0	MEM	MEM
PLRU vs LRU	$1,0$	–	$\frac{1}{2},1$	–	–	–	$\frac{1}{4},\frac{3}{2}$
PLRU vs FIFO	0	–	0	–	–	–	0
PLRU vs MRU	$1,0$	–	0	–	–	–	MEM
MRU vs LRU	$1,0$	0	0	0	0	0	0
MRU vs FIFO	0	0	0	0	0	MEM	MEM
MRU vs PLRU	$1,0$	–	0	–	–	–	MEM

Figure 6.7: Hit-Competitiveness ratios k and subtractive constants c relating FIFO, PLRU, LRU, and MRU at the same levels of associativity. Again, MEM indicates that the computation required more than 1.5GB.

Theorem 6.21 (PLRU(k) vs LRU($1 + log_2 k$))**.**
PLRU(k) *is* $(1, 0)$*-miss-competitive relative to* LRU($1 + log_2 k$)*.*

Proof. The property follows directly from Theorem 5.19 in Chapter 5. The theorem claims that a PLRU-cache set of associativity k always contains the $1 + log_2 k$ most-recently-used elements. It therefore subsumes LRU-cache sets of that associativity. □

By Theorem 6.6, PLRU(k) is also 1-hit-competitive relative to LRU($1 + log_2 k$). Interestingly, PLRU(k) is not miss-competitive at all relative to LRU(l), if $l > 1 + log_2 k$.

6.4.2 Hit-Competitiveness

The hit-competitiveness results show less symmetry than the miss-competitiveness results. Figure 6.7 depicts results relating FIFO, LRU, and PLRU at the same associativities. LRU(k) is not hit-competitive relative to FIFO(k) for any k.
For the case of LRU(3) vs FIFO(3) consider the following example access sequence generated by our tool:

$$[a, b, c]_{\text{LRU}}, [c, b, a]_{\text{FIFO}} \xrightarrow{d} [d, a, b], [d, c, b] \xrightarrow{c} [c, d, a], [d, c, b] \xrightarrow{b} [b, c, d], [d, c, b]$$

The sequence d, c, b incurs 2 hits in the FIFO-part and no hits in the LRU-part. The final state $[b, c, d], [d, c, b]$ is equivalent to the first state $[a, b, c], [c, b, a]$ by the $\approx$-relation, i.e., we can go through this sequence arbitrarily often by renaming d, c, b accordingly.

Theorem 6.22 (LRU(k) relative to FIFO(k))**.**
LRU(k) *is not hit-competitive relative to* FIFO(k)*.*

Proof. For the general case of LRU(k) vs FIFO(k) we can generalize the example sequence given above: Start in the state $[a_1, \ldots, a_k]_{\text{LRU}}, [a_k, \ldots, a_1]_{\text{FIFO}}$. Then, incur a miss in both sets, resulting in $[b, a_1, \ldots, a_{k-1}], [b, a_k, \ldots, a_2]$. Accessing $\langle a_k, \ldots, a_2 \rangle$ will incur $k - 1$ hits in the LRU-part and no hits in the FIFO-part. Furthermore, the resulting state $[a_2, \ldots, a_k, b], [b, a_k, \ldots, a_2]$ is equivalent to the first state according to the $\approx$-relation. $[a_1, \ldots, a_k]_{\text{LRU}}$ and $[a_k, \ldots, a_1]_{\text{FIFO}}$ are compatible, as they are reachable from the initial states $[\bot, \ldots, \bot]_{\text{LRU}}$, $[\bot, \ldots, \bot]_{\text{FIFO}}$ by the sequence $\langle a_1, \ldots, a_k, a_k, \ldots, a_1 \rangle$. □

In contrast, FIFO(k) is $\frac{1}{2}$-hit-competitive relative to LRU(k) for all of the investigated k. Consider the following theorem and its proof.

Theorem 6.23 (FIFO(k) relative to LRU(k))**.**
FIFO(k) *is* $(\frac{1}{2}, \frac{k-1}{2})$*-hit-competitive relative to* LRU(k) *for all* k*.*

Proof. Consider any access sequence s. We argue that for every access a in s that incurs a hit in LRU but a miss in FIFO, the previous access (if there is one) to a in s must have been a hit in FIFO. We prove this by contradiction. Assume the previous access to a

was also a miss in FIFO. Immediately after this access a is in the "last-in" position. For the next access to a to be a miss in FIFO, there must be at least k distinct accesses in between. Otherwise, the next access to a would have incurred a hit. k distinct accesses also evict a from LRU. This is in contradiction to the assumption that the access is a hit in LRU.

There can be at most $k-1$ situations where an access to a incurs a hit in LRU and there is no previous access to a in the sequence, as the set can hold only k elements and the contents of FIFO and LRU overlap at least in the most-recently-used element in compatible states. □

Considering the prominence of the two policies it is quite surprising that this relation has apparently not been discovered before.

Again, it is worthwhile to compare policies of different associativities. Our analysis results suggest that an LRU($2k-1$) is 1-hit-competitive (and therefore also 1-miss-competitive) relative to FIFO(k):

$2k-1$	3	5	7
k	2	3	4
LRU($2k-1$) vs FIFO(k)	$1,0$	$1,0$	$1,0$

Theorem 6.24 (LRU($2k-1$) relative to FIFO(k))**.**
LRU($2k-1$) *is* $(1,0)$*-hit-competitive relative to* FIFO(k).

Proof. We prove this by showing that an element a which is not contained in LRU($2k-1$) cannot be contained in FIFO(k). If a is not in LRU($2k-1$), there must have been at least $2k-1$ accesses to distinct elements after the last access to a. At most $k-1$ of these accesses may have been hits in the FIFO(k). Thus, there must have been at least k misses in the FIFO(k) since the last access to a. Therefore, a must have been evicted from the FIFO(k) as well. □

Due to Theorem 6.7, a *may* analysis for LRU($2k-1$) may be used as a *may* analysis for FIFO(k) as well. Of course, one would expect a FIFO(k) to incur more misses than an LRU($2k-1$) in practice. Still, no analysis has been published to date, that is able to infer misses for a FIFO cache, i.e., a *may* analysis. Observe that it is critical to have *may* information to gain precise *must* information for FIFO. Memory blocks that are guaranteed to miss the cache can safely be assumed to be inserted in the last-in position, which is far from eviction. In contrast, memory blocks that might hit the cache may reside in the first-in position.

In contrast, increasing the associativity of FIFO relative to LRU never results in 1-competitiveness, i.e., FIFO(l) is not 1-competitive relative to LRU(k) for any k and $l > 1$:

Theorem 6.25 (FIFO(l) relative to LRU(k))**.**
For $k \geq 2$*,* FIFO(l) *is* not 1*-hit-competitive relative to* LRU(k) *for any* l*.*

Proof. Consider an access sequence s, such that every kth access in s accesses memory block a and all other accesses are pairwise different. All but the first access to a will hit in LRU(k). All other accesses will miss in both FIFO(l) and LRU(k). Since hits to a do not move a back to the "front" of the FIFO-queue, every $\lceil \frac{l}{k-1} \rceil$th access to a will miss in FIFO(l). □

The analysis results of LRU vs MRU suggest that LRU($2k-2$) is 1-competitive relative to MRU(k):

Theorem 6.26 (LRU($2k-2$) relative to MRU(k))**.**
LRU($2k-2$) *is* $(1,0)$*-hit-competitive relative to* MRU(k)*.*

Proof. We prove that a memory block older than $2k-2$ cannot be contained in a cache-set of MRU(k). LRU($2k-2$) contains all memory blocks of age $0 \ldots 2k-3$.
After accessing a memory block a, its MRU-bit is 1. After accessing at most $k-1$ different memory blocks, the MRU-bits are reset, and a's MRU-bit is 0. At this point, its age is at most $k-1$. Each further access to memory blocks older than a (which will "age" a), will cause one of the $k-1$ MRU-bits, that are 0, to flip. After at most $k-1$ accesses, a must have been replaced, since its MRU-bit was 0. □

Due to Theorem 6.7, a *may* analysis for LRU($2k-2$) may therefore be used as a *may* analysis for MRU(k) as well. The previous theorem and the $\frac{1}{2}$-hit-competitiveness of FIFO relative to MRU yield the following corollary:

Corollary 6.27 (FIFO($2k-2$) relative to MRU(k))**.**
FIFO($2k-2$) *is* $\frac{1}{2}$*-hit-competitive relative to* MRU(k)*.*

Proof. This follows from the $\frac{1}{2}$-hit-competitiveness of FIFO($2k-2$) relative to LRU($2k-2$) (Theorem 6.23) and the 1-competitiveness of LRU($2k-2$) relative to MRU(k) (Theorem 6.26) by Theorem 6.12. □

Hit-competitive ratios obtained with our tool suggest that this is tight.

6.4.3 Miss-Competitiveness Relative to OPT

In the context of paging, [Sleator and Tarjan, 1985] studied the competitiveness of LRU, FIFO, LIFO and LFU relative to OPT, the optimal offline policy [Belady, 1966]. As described in Section 6.3.4, it is possible to automatically compute the competitiveness of a policy relative to OPT.

Before discussing our results, let us recapitulate the main results of [Sleator and Tarjan, 1985]. They first show how poorly any online policy performs compared with OPT:

Theorem 6.28 (Online algorithms vs OPT, [Sleator and Tarjan, 1985])**.**
Let P be an online algorithm. Then there exist arbitrarily long access sequences $s \in S$,

Associativity:	2	3	4	5	6	7	8
LRU vs OPT	$2,1$	$3,2$	$4,3$	$5,4$	TIME	MEM	MEM
FIFO vs OPT	$2,1$	$3,3$	$4,4$	MEM	MEM	MEM	MEM
PLRU vs OPT	$2,1$	$-$	∞	$-$	$-$	$-$	MEM
MRU vs OPT	$2,1$	$4,3$	$6,5$	MEM	MEM	MEM	MEM

Figure 6.8: Miss-Competitiveness ratios k and additive constants c relating FIFO, PLRU, MRU, and LRU with OPT at the same associativity. If the computation required more than 30 minutes or exceeded the heap space limit of 1.5GB, we report MEM or TIME, respectively.

and cache-set states $p \in C^{P(k)}, q \in C^{\mathrm{OPT}(l)}$*, such that*

$$m_{P(k)}(p,s) \geq \frac{k}{k-l+1} \cdot m_{\mathrm{OPT}(l)}(q,s)$$

for $k \geq l$*.*

They go on to show that both LRU and FIFO achieve the optimal competitive ratio $\frac{k}{k-l+1}$:

Theorem 6.29 (LRU and FIFO vs OPT, [Sleator and Tarjan, 1985])**.**
Let P *be* LRU *or* FIFO*. Then*

$$m_{P(k)}(p,s) \leq \frac{k}{k-l+1} \cdot m_{\mathrm{OPT}(l)}(q,s) + l$$

for $k \geq l$*, all access sequences* $s \in S$*, and all cache-set states* $p \in C^{P(k)}, q \in C^{\mathrm{OPT}(l)}$*.*

This immediately implies that FIFO(k) and LRU(k) are $(\frac{k}{k-l+1}, l)$-miss-competitive relative to OPT(l) and any other policy $P(l)$. As we have seen earlier, neither PLRU(k) nor MRU(k) are k-competitive relative to FIFO(k). Therefore PLRU(k) and MRU(k) are not optimal in the sense of [Sleator and Tarjan, 1985]; they cannot be k-competitive relative to OPT(k).

We have used our tool to compute competitive ratios and appropriate constants for LRU, FIFO, PLRU, and MRU relative to OPT. Results relating the policies at the same associativities are depicted in Figure 6.8 other results can be found in the appendix. Whenever our tool did not run out of time or space, it correctly reproduced the competitive ratios of LRU and FIFO. The quotient transition systems $\overline{T}_{P(k),\mathrm{OPT}(l)}$ grow much more rapidly for rising associativities k and l than for two online policies, as OPT-states carry *future constraints*. Due to the non-determinism, the average number of transitions per state is also greater, than in the case of comparing two online policies.

Based on the limited set of results, we conjecture but do not prove that MRU(k) is $(2k-2, 2k-3)$-miss-competitive relative to OPT(k). For PLRU, the following theorem is easily proven:

Theorem 6.30 (PLRU vs OPT)**.**
PLRU(k) *is* $(\frac{log_2 k+1}{log_2 k-l+2}, l)$*-miss-competitive relative to* OPT(l) *for* $\log_2 k + 1 \geq l$.

Proof. By Theorem 6.21, PLRU(k) is $(1, 0)$-miss-competitive relative to LRU($\log_2 k+1$). By Theorem 6.29, LRU($\log_2 k+1$) is $(\frac{\log_2 k+1}{\log_2 k+1-l+1}, l)$-miss-competitive relative to OPT(l) for $\log_2 k + 1 \geq l$. These two relations can be combined by Theorem 6.11. □

Our tool shows that PLRU(4) is $(2, 2)$-miss-competitive relative to OPT(3). However, the theorem only yields $(\frac{log_2 4+1}{log_2 4-3+2}, 3) = (3, 3)$, i.e., there is still room for improvement.

6.5 Related Work

There is a large body of work on competitive analysis. The field was pioneered by [Sleator and Tarjan, 1985] in 1985. Amongst others, they studied LRU and FIFO relative to OPT, the optimal offline policy [Belady, 1966], as discussed in the previous section. We have observed that relative to each other LRU(k) and FIFO(k) are "only" k-competitive, as is LRU(k) relative to PLRU(k). As PLRU(k) is neither competitive relative to FIFO(k) nor relative to LRU(k) it is obviously also not competitive relative to OPT(k).

In [Aspnes and Waarts, 1996], *relative competitiveness* is used with a different meaning than in our work. Their definition captures that an algorithm A is k-competitive relative to the competitiveness of a subroutine B that it uses. If B is in turn implemented by an l-competitive algorithm, the resulting algorithm is $k \cdot l$-competitive overall. This definition allows to perform modular competitive analyses.

In an effort to overcome weaknesses of traditional competitive analysis, [Koutsoupias and Papadimitriou, 1994] propose two refinements of competitive analysis. One of the refinements is coined *comparative analysis*. There, two classes of algorithms $\mathcal{A}$ and $\mathcal{B}$ are compared to determine how powerful $\mathcal{A}$ is relative to $\mathcal{B}$. Choosing $\mathcal{A}$ and $\mathcal{B}$ to be singletons, *comparative analysis* coincides with our notion of *relative competitive analysis*.

Smaragdakis et al. introduce *Early Eviction LRU* (EELRU) [Smaragdakis et al., 2003], an adaptive algorithm based on the LRU stack model. They prove *robustness* of EELRU with respect to LRU. They call a policy A *robust* with respect to policy B if A is k-miss-competitive relative to B for some k. In subsequent work [Smaragdakis, 2004] shows how to construct a policy AB from two given policies A, B, that is 2-robust, i.e., 2-miss-competitive, with respect to both A and B.

There is one major difference between our approach and most of the existing work on competitive analysis that we are aware of: Most work interprets the competitive ratio as an indicator of the quality of a policy. In such work, the definition of competitiveness is relaxed to better distinguish policies that perform very differently in practice, like LRU and FIFO. For this purpose, our work is rather useless. If some policy A is not competitive relative to some other policy B, this is not a good indicator of the

performance A and B are going to show on the average. For our purpose of giving guarantees on the number of hits or misses, the strict worst-case definition is, however, the right choice.

Previous work on cache analysis of set-associative caches has been discussed in Chapter 4. It mostly considered LRU replacement [Ferdinand et al., 1997, Ferdinand and Wilhelm, 1999, White et al., 1997, Ghosh et al., 1998, Chatterjee et al., 2001]. Cache analyses embedded in timing analyses [Ferdinand et al., 1997, Ferdinand and Wilhelm, 1999, White et al., 1997] classify individual accesses as hits or misses. Other approaches using Cache Miss Equations [Ghosh et al., 1998] or Presburger formulas [Chatterjee et al., 2001] compute the number of misses of larger parts of a program, like loop nests. The scope of both approaches can be extended by our results. In [Heckmann et al., 2003] a *must* analysis for an 8-way PLRU is discussed, that maintains the 4 most-recently-used elements. The soundness of this approach can be easily explained by the 1-miss-competitiveness of PLRU(8) relative to LRU(4).

6.6 Summary, Conclusions, and Future Work

We have studied the relative competitiveness of the four well-known and widely-used replacement policies LRU, FIFO, PLRU, and MRU. By building a finite quotient structure of the transition system induced by a pair of policies, we were able to compute competitive ratios automatically. The competitiveness analyses revealed interesting previously unknown relations between different policies. The results can be used in two ways: One may bound the number of hits and misses for the execution of a program, or, in the case of $(1, 0)$-competitiveness, one may build sound *may* and *must* analyses. The relations between LRU$(2k - 1)$, LRU$(2k - 2)$, and FIFO(k), MRU(k), respectively, allow to construct *may* cache analyses for caches with FIFO and MRU replacement. These *may* analyses are even optimal with respect to the predictability metrics of Chapter 5. In recent work, [Grund et al., 2008] have successfully applied the *may* analysis for FIFO in the timing analysis of the Branch Target Buffer (BTB) of the MOTOROLA POWERPC 56X family.

The competitiveness results we are computing hold for arbitrary access sequences. Therefore, they hold for any program. Most programs do not exhibit the worst-case relative behavior. By restricting the possible access sequences it would be possible to obtain smaller competitive ratios. Such restrictions could be for a particular program or for classes of programs, like structured programs. However, restricting access sequences potentially yields larger quotient transition systems. Another line of future work would be to consider other relations between replacement policies, like the maximal difference of the miss rates, which would require rather simple extensions of our current framework.

7

Sensitivity of Replacement Policies – On the Correctness of Measurement-based Timing Analysis

The sensitivity of a cache replacement policy expresses to what extent the initial state of the cache may influence the number of cache hits and misses during program execution. We have slightly modified the method presented in Chapter 6 to precisely compute sensitivity properties for a large class of replacement policies including LRU, FIFO, PLRU, and MRU. Analysis results demonstrate that the initial state can have a strong impact on the cache performance if FIFO, MRU, or PLRU is used. A simple model of execution time is used to evaluate the impact of cache sensitivity on measured execution times. The model shows that underestimating the number of misses as strongly as is possible for FIFO, MRU, and PLRU may yield worst-case-execution-time estimates that are dramatically wrong.

7.1 Introduction

To obtain tight bounds on the execution time of a task, timing analyses *must* take into account the cache architecture. In general, execution times of tasks vary depending on inputs and the initial state of the hardware. A large part of this variation can usually be attributed to the cache performance. At the level of individual instructions, the influence of the initial state is particularly obvious: cache misses, pipeline stalls, etc. introduce great variance into the execution time of an instruction. It is expected that some of the variances in execution times of multiple instructions cancel each other out, i.e., worst-cases do not coincide [Bernat et al., 2002]. In addition, one may expect different initial states to eventually converge. This is probably true for pipeline states. In many cache architectures, however, this is not the case [Berg, 2006].

Different methods have been proposed for timing analysis [Wilhelm et al., 2008]; measurement[1] [Petters, 2002,Bernat et al., 2002,Wenzel, 2006] and static analysis [Ferdinand

[1]Measurement-based timing analysis as discussed here is also referred to as hybrid measurement-based timing analysis as opposed to end-to-end measurement-based analysis.

et al., 2001, Theiling et al., 2000] being the most prominent. Both methods compute estimates of the worst-case execution times for program fragments like basic blocks. If these estimates are correct, i.e., they are upper bounds on the worst-case execution time of the program fragment, they can be combined to obtain an upper bound on the worst-case execution time of the task. This combination takes into account user-annotated or automatically computed loop bounds.

While using similar methods in the combination of execution times of program fragments, the two methods take fundamentally different approaches to compute these times:

- Static analyses based on abstract models of the underlying hardware compute invariants about the set of all execution states at each program point under *all* possible initial states and inputs and derive upper bounds on the execution time of program fragments based on these invariants.

- Measurement executes each program fragment with a subset of the possible initial states and inputs. The maximum of the measured execution times is in general an underestimation of the worst-case execution time.

If the abstract hardware models are correct, static analysis computes safe upper bounds on the WCETs of program fragments and thus also of tasks. However, creating abstract hardware models is an error-prone and laborious process, especially if no precise specification of the hardware is available. Recent work [Schlickling and Pister, 2007] explores automation of the creation of abstract hardware models given a concrete VHDL model.

The advantage of measurement over static analysis is that it is more easily portable to new architectures, as it does not rely on an abstract model of the architecture. In addition it may compute more precise estimates of the WCET. On the other hand, soundness of measurement-based approaches is often hard to guarantee. Measurement would trivially be sound if all initial states and inputs would be covered. Due to their huge number this is usually not feasible. Instead, only a subset of the initial states and inputs can be considered in the measurements. Relatively simple architectures without any performance-enhancing features like pipelines, caches, etc., exhibit the same timing independently of the initial state. For such architectures, measurement-based timing analysis is sound [Wenzel, 2006]. [Deverge and Puaut, 2005] and [Wenzel, 2006] propose to lock the cache contents [Puaut and Decotigny, 2002, Vera et al., 2003] and to flush the pipeline at program points where measurement starts. This is not possible on all architectures and it also has a detrimental effect on both the average- and the worst-case execution times of tasks. In this chapter, we study whether measurement-based timing analysis can be performed in the presence of "unlocked" caches. To this end, we introduce the notion of sensitivity of a cache replacement policy.

Sensitivity of a cache replacement policy expresses to what extent the initial state of the cache may influence the number of cache hits and misses during program execution. We first describe how to adapt the method described in Chapter 6 to automatically compute sensitivity properties for a large class of cache replacement policies, including LRU, FIFO, PLRU, and MRU. However, our main contributions besides the introduction of sensitivity are the application of the analysis to relevant policies and the interpretation

of the analysis results with respect to measurement-based timing analysis: Analysis results demonstrate that the initial state of the cache can have a strong impact on the number of cache hits and misses during program execution if FIFO, MRU, or PLRU replacement is used. A simple model of execution time is used to evaluate the impact of cache sensitivity on measured execution times. The model shows that underestimating the number of misses as strongly as is possible for FIFO, MRU, and PLRU may yield worst-case-execution-time estimates that are dramatically wrong. In a slightly modified analysis we show that the "empty cache is worst-case initial state" assumption [Petters, 2002] is wrong for FIFO, MRU and PLRU. On the other hand, our analysis results show that LRU lends itself well to measurement- or simulation-based approaches as the influence of the initial cache state is minimal.

Outline

In Section 7.2 we formally introduce our notion of sensitivity. In Section 7.3 we describe how to compute sensitive ratios automatically. Our results are presented in Section 7.4. Their impact on measured execution times is evaluated in Section 7.5. Consequences of our results are discussed in Section 7.6.

7.2 Sensitivity

Section 2.2.9 introduces important domains and notations used in the following definitions and throughout the chapter. The most important notions are $m_P(q, s)$ and $h_P(q, s)$, which compute the number of of misses and hits, respectively, of policy P starting in state q processing access sequence s.

7.2.1 Definition of Sensitivity

We would like to investigate the influence of the *state* on the performance of a cache replacement policy. As cache sets are usually independent of each other, we consider a single cache set, not the entire cache. I.e. we are interested in how sensitive a policy is to the particular state a cache set is in when beginning to process an access sequence. The results can easily be translated to sensitivity properties of entire caches.

Definition 7.1 (Miss-sensitivity to initial state)**.**
A policy P is k-miss-sensitive with additive constant c, short (k, c)-miss-sensitive, if

$$m_P(q, s) \leq k \cdot m_P(q', s) + c$$

for all access sequences $s \in S$ and all cache-set states $q, q' \in C^P$.

The definition captures the maximal influence of the current state of a replacement policy on the future number of misses. Policy P will incur at most k times the number

of misses plus constant c on any access sequence starting in state q instead of any other state q'. Miss-sensitivity can be cast as competitiveness of a policy relative to itself. However, the states q, q' are not restricted to be compatible.

Likewise we define hit-sensitivity.

Definition 7.2 (Hit-sensitivity to initial state)**.**
A policy P is k-hit-sensitive with subtractive constant c, short (k, c)-hit-sensitive, if

$$h_P(q, s) \geq k \cdot h_P(q', s) - c$$

for all access sequences $s \in S$ and all cache-set states $q, q' \in C^P$.

Policy P will induce at least k times the number of hits minus constant c on any access sequence starting in state q instead of state q'.

As in relative competitiveness, we sometimes say that a policy is k-sensitive without specifying an appropriate additive (subtractive) constant. In such cases, we implicitly demand that such a constant exists. The following definition is an example of such a case:

Definition 7.3 (Sensitive ratio)**.**
The sensitive miss and hit ratios s_P^m and s_P^h of P are defined as

$$\begin{aligned} s_P^m &= \mathit{inf}\ \{k \mid P \text{ is } k\text{-miss-sensitive}\} \\ \mathit{and}\ s_P^h &= \mathit{sup}\ \{k \mid P \text{ is } k\text{-hit-sensitive}\}\,. \end{aligned}$$

Our focus will be on computing these sensitive ratios and appropriate additive (subtractive) constants. As the competitive ratio is the best characterization of a policy's relative competitiveness, the sensitive ratio is the best characterization of a policy's sensitivity.

Every policy is by definition 0-hit-sensitive. However, a policy may not be k-miss-sensitive for any k. In that case, we will call it ∞-miss-sensitive. For a policy P that is ∞-miss-sensitive, the number of misses incurred in P in state q cannot be bounded by the number of misses starting in another state q'.

Lifting Sensitivity Results to Set-Associative Caches

The sensitivity definitions talk about the number of hits and misses in individual cache sets[2]. It is, however, easy to lift the sensitivity results to entire set-associative caches: One simply needs to multiply the additive (subtractive) constant by the number of cache sets. The factor k is not affected.

7.3 Computing Sensitive Ratios

We have described the computation of relative competitive ratios in Chapter 6. Sensitive ratios can be obtained in a very similar way. Differences arise when building the quotient

[2] An individual cache set can also be considered to be a fully-associative cache.

transition system. In contrast to the competitiveness case, we do not only have to consider compatible states. Instead, all pairs of states $q, q' \in C^P$ must be handled. This currently limits our approach to associativities smaller than 9.

7.3.1 Induced Transition System

As in the case of relative competitiveness, the policy P induces a transition system. Hit- and miss-sensitivity are properties of this system.

Definition 7.4 (Induced transition system)**.**
A policy P induces a labelled transition system $T_P = (S_P, R_P)$, where

$$S_P = \left\{(q, q') \mid q \in C^P, q' \in C^P\right\} = C^P \times C^P,$$

the states, are pairs of cache set states of policy P.

$$\begin{aligned} R_P \quad = \quad & \{((p,q),(m_p,m_q),(p',q')) \mid (p,q) \in S_P, a \in M, \\ & (p',q') = (update_P(p,\langle a\rangle), update_P(q,\langle a\rangle)), \\ & (m_p,m_q) = (m_P(p,\langle a\rangle), m_P(q,\langle a\rangle))\} \end{aligned}$$

is the transition relation. Transitions are labelled with the number of misses (0 or 1) incurred by the accesses in the two cache set states, respectively.

In contrast to $S_{P,P}$ in relative competitiveness, states in S_P do not have to be compatible. This yields a potentially much larger state space and less efficient analyses.

Sensitivity values depend on the number of misses (hits) on paths through the transition system: The definitions of hit- and miss-sensitivity translate directly to properties of paths (see Definition 6.14) of the induced transition system. A policy P is k-miss-sensitive with additive constant c, iff

$$\sum_i \pi(i)_{|1} \leq k \cdot \sum_i \pi(i)_{|2} + c \text{ for every path } \pi \in \Pi(T_P),$$

where $|1$ and $|2$ select the first and second component of a tuple, respectively. Likewise, a policy P is k-hit-sensitive with subtractive constant c, iff

$$\left(\sum_i 1 - \pi(i)_{|1}\right) \geq k \cdot \left(\sum_i 1 - \pi(i)_{|2}\right) - c \text{ for every path } \pi \in \Pi(T_P).$$

7.3.2 Quotient Transition System

The relation $\approx$ of Definition 6.15 can be used to partition the states of S_P into equivalence classes as in the case of relative competitiveness. This induces the finite quotient transition system $\overline{T}_P = (\overline{S}_P, \overline{R}_P) = T_P/\!\approx$.

The sensitive analysis can be performed on the quotient system:

Theorem 7.5 (Path equivalence)**.** *The transition systems T_P and $\overline{T}_P$ are path equivalent, i.e., $\Pi(T_P) = \Pi(\overline{T}_P)$.*

Proof. The proof carries over directly from the proof of Theorem 6.17. □

Building the Quotient Transition System

The only difference to the computation of relative competitive ratios lies in the construction of the quotient transition system. Algorithm 2 consists of two steps: The computation of $\overline{S}_P$ and the computation of $\overline{R}_P$.

To compute $\overline{S}_P$, the algorithm proceeds by taking a yet *unprocessed* normalized state from the *Unprocessed* queue and by computing all its normalized successor states until all states have been processed. It starts with the pair of compatible states (i^P, i^Q). Instead of computing successors under the same accesses only, we have to take into account arbitrary uncorrelated accesses to both cache-set states. Pairs of cache-set states are normalized by NORMALIZE(p, q) as described in Section 6.3.2.

Note that for a fixed a, NORMALIZE$(update_P(p, \langle a \rangle), update_Q(q, \langle b \rangle))$ yields $\approx$-equivalent states for all $b \in \overline{CC_P(p) \cup CC_Q(q) \cup \{a\}}$. Similarly, $\approx$-equivalent states arise for a fixed b in NORMALIZE$(update_P(p, \langle a \rangle), update_Q(q, \langle b \rangle))$, for all $a \in \overline{CC_P(p) \cup CC_Q(q) \cup \{b\}}$. Therefore, it suffices to consider the following five cases of access pairs (a, b), which are implicitly covered by the algorithm:

1. $(a, b) \in (CC_P(p) \cup CC_P(q)) \times (CC_P(p) \cup CC_P(q))$.
2. $(a, b) \in (CC_P(p) \cup CC_P(q)) \times \{m_1\}$.
3. $(a, b) \in \{m_1\} \times (CC_P(p) \cup CC_P(q))$.
4. $(a, b) = (m_1, m_1)$.
5. $(a, b) = (m_1, m_2)$.

where $m_1 \in \overline{CC_P(p) \cup CC_P(q)}$ and $m_2 \in \overline{CC_P(p) \cup CC_P(q) \cup \{m_1\}}$.

Once $\overline{S}_P$ has been computed, $\overline{R}_P$ can be computed as in the case of relative competitiveness.

7.3.3 Computation of Sensitive Ratios

Once we have built up the quotient transition system, determining the minimal k such that P is miss-sensitive amounts to computing the *maximum cycle ratio* [Lawler, 1966, Ahuja et al., 1993].

Theorem 7.6 (Maximum cycle ratio)**.** *The maximum cycle ratio k of $\overline{T}_P$, where the cost associated with a transition is the number of misses incurred in the first component and the associated travel time is the number of misses incurred in the second component, is equal to the sensitive miss ratio of P.*

Proof. The proof of carries over directly from the proof of Theorem 6.20. □

Algorithm 2: Building Quotient Transition System

Input: Policy P
Output: Quotient Transition System $\overline{T}_P = (\overline{S}_P, \overline{R}_P)$
begin
  $\overline{S}_P \leftarrow \{\textsc{Normalize}(i^P, i^P)\}$
  $Unprocessed \leftarrow [\textsc{Normalize}(i^P, i^P)]$
  while $\neg\textsc{Empty}(Unprocessed)$ **do**
    $(p, q) \leftarrow \textsc{Pop}(Unprocessed)$
    $m_1 \leftarrow \textsc{SelectOne}(\overline{CC_P(p) \cup CC_P(q)})$
    $m_2 \leftarrow \textsc{SelectOne}(\overline{CC_P(p) \cup CC_P(q) \cup \{m_1\}})$
    foreach $a \in CC_P(p) \cup CC_P(q) \cup \{m_1\}$ **do**
      $p' \leftarrow update_P(p, \langle a \rangle)$
      foreach $b \in CC_P(p) \cup CC_P(q) \cup \{m_1, m_2\}$ **do**
        $(p', q') \leftarrow \textsc{Normalize}(p', update_P(q, \langle b \rangle))$
        if $(p', q') \notin \overline{S}_P$ **then**
          $\textsc{Push}(Unprocessed, (p', q'))$
          $\overline{S}_P \leftarrow \overline{S}_P \cup \{(p', q')\}$
  $\overline{R}_P \leftarrow \emptyset$
  foreach $(p, q) \in \overline{S}_P$ **do**
    foreach $a \in CC_P(p) \cup CC_P(q) \cup \{\textsc{SelectOne}(\overline{CC_P(p) \cup CC_P(q)})\}$ **do**
      $(p', q') \leftarrow \textsc{Normalize}(update_P(p, \langle a \rangle), update_P(q, \langle a \rangle))$
      $(m_p, m_q) \leftarrow (m_P(p, \langle a \rangle), m_P(q, \langle a \rangle))$
      $\overline{R}_P \leftarrow \overline{R}_P \cup \{((p, q), (m_p, m_q), (p', q'))\}$
end

Similarly, the sensitive hit ratio of P is the minimum cycle ratio of $\overline{T}_P$, where the cost associated with a transition is the number of hits in the first component and the associated travel time is the number of hits in the second component.

7.4 Results

Running our tool on a Core 2 Duo E6750 at 2.66GHz with 2GB of RAM, we have obtained sensitivity results for LRU, FIFO, PLRU, and MRU at associativities ranging from 2 to 8. To avoid any paging activity, we have limited the heap space to be allocated by our tool to 1.5GB. If the computation required more than 30 minutes or exceeded the heap space limit of 1.5GB, we report MEM or TIME, respectively. Note that we have computed the precise sensitive ratios not just upper bounds. I.e. there are arbitrarily long access sequences and pairs of initial states that exhibit the computed hit and miss ratios.

Figure 7.1(a) depicts our results for the miss-sensitivity of LRU, FIFO, and PLRU. results for LRU, FIFO, MRU PLRU, and OPT at associativities ranging from 2 to 8. LRU is very insensitive to its state. The difference in misses is bounded by the associativity k. This is unavoidable for any policy, as the initial states may have completely disjoint contents. FIFO, MRU, and PLRU are much more sensitive to their state than LRU.

For the analyzed associativities, depending on their state, FIFO(k) may have up to k times as many misses, and MRU(k) may have up to $2k - 3$ times as many misses. PLRU(2) coincides with LRU(2). For greater associativities, the number of misses incurred starting in one state cannot be bounded by the number of misses incurred starting in another state. Of course, the number of misses is always bounded by the length of the access sequence. However, given *only* the number of misses and not the length of the sequence no bound can be given.

As the number of misses may only differ by a constant for LRU, the number of hits may only differ by the same constant. For FIFO, the situation is different: no lower bound on the number of hits can be given for one state, given the number of hits in another state. The same holds for MRU(k), if $k > 2$. For associativity $k = 2$, MRU(k), LRU(k), and PLRU(k) coincide. The results for PLRU are only slightly more encouraging than in the miss-sensitivity case. At associativity 8, a sequence may cause only 1/11 of the number of hits starting in one state that it would cause starting in another state. See Figure 7.1(b) for the analysis results.

Summarizing, FIFO, MRU, and PLRU may in the worst-case be heavily influenced by the starting state. LRU is very robust, in that the number of hits and misses is affected in the least possible way.

	2	3	4	5	6	7	8
LRU	$1,2$	$1,3$	$1,4$	$1,5$	$1,6$	$1,7$	$1,8$
FIFO	$2,2$	$3,3$	$4,4$	$5,5$	$6,6$	$7,7$	$8,8$
PLRU	$1,2$	$-$	∞	$-$	$-$	$-$	∞
MRU	$1,2$	$3,4$	$5,6$	$7,8$	MEM	MEM	MEM

(a) Miss-Sensitive ratio, k, and additive constant, c, for LRU, FIFO, PLRU, and MRU.

	2	3	4	5	6	7	8
LRU	$1,2$	$1,3$	$1,4$	$1,5$	$1,6$	$1,7$	$1,8$
FIFO	$0,0$	$0,0$	$0,0$	$0,0$	$0,0$	$0,0$	$0,0$
PLRU	$1,2$	$-$	$\frac{1}{3},\frac{5}{3}$	$-$	$-$	$-$	$\frac{1}{11},\frac{19}{11}$
MRU	$1,2$	$0,0$	$0,0$	$0,0$	MEM	MEM	MEM

(b) Hit-Sensitive ratio, k, and subtractive constant, c, for LRU, FIFO, PLRU, and MRU.

Figure 7.1: Miss- and Hit-Sensitivity Results. As an example of how this should be read, PLRU(4) is $(\frac{1}{3},\frac{5}{3})$-hit-sensitive with subtractive constant . ∞ indicates that a policy is not k-miss-sensitive for any k. PLRU is only defined for powers of two.

Is the Empty Cache the Worst-Case Initial State?

One could argue that it is still safe to assume an empty cache or equivalently a cache filled with irrelevant data only as the starting state [Petters, 2002, page 39ff], assuming that an empty cache were worse than any non-empty cache. This is *not* true for FIFO, MRU, and PLRU. We have performed a second analysis that fixed the reference starting state (q' in the definitions) to be empty. The analysis revealed the same sensitive ratios as in the general case with all additive (subtractive) constants being zero. For LRU, this is in fact a positive result, as it confirms that the empty cache is indeed the worst-case for any access sequence.

This is the example produced by the tool for the miss-sensitivity of FIFO(4):

Example (Miss-sensitivity of FIFO(4))**.**
Consider the pair of cache-set states $[b,c,d,e]_{\text{FIFO}}, [\bot,\bot,\bot,\bot]_{\text{FIFO}}$*.*
The sequence $\langle d,e,a,b\rangle$ *leads it to the pair* $[a,b,c,d]_{\text{FIFO}}, [b,a,e,d]_{\text{FIFO}}$*:*

$$[b,c,d,e],[\bot,\bot,\bot,\bot] \xrightarrow{d} [b,c,d,e],[d,\bot,\bot,\bot] \xrightarrow{e} [b,c,d,e],[e,d,\bot,\bot]$$
$$\xrightarrow{a} [a,b,c,d],[a,e,d,\bot] \xrightarrow{b} [a,b,c,d],[b,a,e,d]$$

Now, consider the access sequence $\langle e,d,f,b\rangle$*, which leads the pair into the* $\approx$*-equivalent pair* $[b,f,d,e]_{\text{FIFO}}, [f,b,a,e]_{\text{FIFO}}$*:*

$$[a,b,c,d],[b,a,e,d] \xrightarrow{e} [e,a,b,c],[b,a,e,d] \xrightarrow{d} [d,e,a,b],[b,a,e,d]$$
$$\xrightarrow{f} [f,d,e,a],[f,b,a,e] \xrightarrow{b} [b,f,d,e],[f,b,a,e]$$

The state that originated from the empty state $[\perp, \perp, \perp, \perp]_{\text{FIFO}}$ *incurs only one miss on this sequence, while the other state misses in each of the four accesses. As the resulting states are* $\approx$*-equivalent there is another sequence that will show the same behavior in the two states and so on.*

It has been observed earlier [Berg, 2006], that the empty cache is not necessarily the worst-case starting state for PLRU. Our work demonstrates to what extent it may be better than the real worst-case initial state in the case of FIFO, MRU, and PLRU. It turns out that except for the additive (subtractive) constant, starting with an empty cache may be as bad as starting in any other state.

Pathological Cases?

Of course, it is not very likely to start measurements in a state that minimizes the number of misses for the following access sequence. Yet, it is difficult to associate a particular probability with this event. One should also realize that many states in between the worst- and the best-case (i.e. even if one does not start in the state that minimizes the number of misses) may still perform significantly better than the worst-case initial state.

7.5 Impact of Results on Timing Analysis

To illustrate on a simplified scenario the impact of the sensitivity results on measured execution times, we adopt a simple model of execution time in terms of cache performance of [Hennessy and Patterson, 2003]. In this model, the execution time is the product of the clock cycle time and the sum of the CPU cycles (the pure processing time) and the memory stall cycles:

$$\text{Exec. time} = (\text{CPU cycles} + \text{Memory stall cycles}) \times \text{Clock cycle}$$

The equation makes the simplifying assumption that the CPU is stalled during a cache miss. Furthermore, it assumes that the CPU clock cycles include the time to handle cache hits.

Let CPI_{hit} be the average number of cycles per instruction if no cache misses occur. Then, the CPU cycles are simply a product of the number of instructions IC and CPI_{hit}:

$$\text{CPU cycles} = \text{IC} \times \text{CPI}_{hit}$$

The number of memory stall cycles depends on the number of instructions IC, the number of misses per instruction and the cost per miss, the miss penalty:

$$\begin{aligned}
\text{Memory stall cycles} &= \text{Number of misses} \times \text{Miss penalty} \\
&= \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty} \\
&= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}
\end{aligned}$$

Now assume we have measured an execution time of T_{meas} in a system with a 4-way set-associative FIFO cache. By which factor may the "real" worst-case execution time T_{wc} differ from T_{meas} due to different initial states of the cache? Let the number of memory accesses per instruction be 1.2[3] and let the miss penalty be 50. Due to pipeline stalls let the CPI_{hit} be 1.5. Further assume, the miss rate Miss rate_{meas} during the measurement was 5%. The sensitive miss-ratio of FIFO(4) is 4. Neglecting the additive constant, the worst-case miss rate Miss rate_{wc} could thus be as high as 20%. Plugging the above assumptions into the equations and simplification yields

$$\begin{aligned}\frac{T_{wc}}{T_{meas}} &= \frac{\text{CPI}_{hit} + \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate}_{wc} \times \text{Miss penalty}}{\text{CPI}_{hit} + \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate}_{meas} \times \text{Miss penalty}} \\ &= \frac{1.5+1.2\times0.20\times50}{1.5+1.2\times0.05\times50} = \frac{13.5}{4.5} = 3\end{aligned}$$

So, in the example of a 4-way set-associative FIFO cache, the worst-case execution time may be a factor of 3 higher than the measured time only due to the influence of the initial cache state. If PLRU or MRU were used as a replacement policy the difference could be even greater. As measurement usually does not allow to determine the miss rate (or simply the number of misses) it is not even possible to add a conservative overhead to the measured execution times to account for the sensitivity to the initial state.

The above analysis considers the impact of cache sensitivity on an individual measurement. Measurement-based timing analysis as described in the literature [Petters, 2002, Bernat et al., 2002, Wenzel, 2006, Deverge and Puaut, 2005] does not advocate end-to-end measurements. Instead, measurements of program fragments are performed and later combined to obtain an estimate of the worst-case execution time of the whole program. The above arguments apply to any of the measurements of program fragments. If the measurement of an important fragment like the body of an inner loop is far off, the estimate for the whole program will as a consequence be far off as well.

7.6 Summary, Conclusions, and Future Work

We have introduced a notion of sensitivity of cache replacement policies that captures the influence of the initial state of the cache on the future cache performance. Employing techniques described in Chapter 6 allows to compute sensitive ratios of a large class of replacement policies, including the well-known and widely-used families of replacement policies, LRU, FIFO, MRU, and PLRU.

The analysis results revealed great differences among LRU, FIFO, and PLRU, that yield another argument in favor of using LRU in the design of predictable real-time systems. In the case of FIFO, MRU, and PLRU, the initial state can have a great influence on the number of hits and misses during program execution. A simple model of execution time demonstrates the impact of cache sensitivity on measured execution times. It shows that underestimating the number of misses as strongly as is possible for FIFO, MRU, and

[3]Each instruction causes one instruction fetch and possibly data fetches.

PLRU yields worst-case-execution-time estimates that are dramatically wrong. Further analysis revealed that the "empty cache is worst-case initial state" assumption [Petters, 2002] is wrong for FIFO, MRU, and PLRU.

To obtain safe results by measurement with respect to cache performance the cache contents should be locked as proposed in [Deverge and Puaut, 2005, Wenzel, 2006, Vera et al., 2003, Puaut and Decotigny, 2002], which may have an adverse effect on average- and worst-case execution time.

Our study of predictability of the above policies in Chapter 5 provides limits on the precision of static cache analyses. In contrast to measurement, static analysis can always compute correct approximations. However, for those policies that are particularly sensitive to the initial state, soundness comes at a price: their precision is limited for FIFO, MRU, and PLRU. In contrast, static analyses can be very precise for LRU, which is least sensitive.

Our results hold for arbitrary access sequences. Therefore, they hold for any program. Many programs do not exhibit the worst-case cache behavior. By restricting the possible access sequences it would be possible to obtain smaller sensitive ratios for particular programs. However, computing precise sensitive ratios in such restricted scenarios is more difficult than in the present case as we cannot compute a quotient transition system in a similar way. States that are equivalent in the current setting may not be equivalent if memory accesses are restricted in some way.

8

Summary, Conclusions, and Future Work

An important part in the design of hard real-time systems is the proof of timeliness, which is determined by the worst-case performance of the system. Performance boosting components like caches have an increasing impact on both the average- and the worst-case performance. WCET analyses need to account for the cache behavior in a *sound* and *precise* way. Sound and precise *may* and *must* cache analyses are known for LRU. Prior to our work, only imprecise *must* analyses were known for other policies, like FIFO and PLRU. It was unclear whether more precise analyses were yet to be discovered or whether these policies simply do not permit greater precision.

8.1 Summary of Contributions

Predictability Metrics

We have introduced *predictability metrics* that capture how quickly cache analyses can obtain *may*- and *must*-information under a replacement policy. The metrics are independent of any particular cache analysis. They mark a limit on the precision of *any* cache analysis. Under these metrics, LRU is optimal, i.e., *may*- and *must*-information can be obtained in the least possible number of memory accesses. PLRU, MRU, and FIFO perform considerably worse. Compared to an 8-way LRU, it takes more than twice as many accesses to regain complete *must*-information for equally-sized PLRU, MRU, and FIFO caches. As a consequence, it is *impossible* to construct cache analyses for PLRU, MRU, and FIFO that are as precise as known LRU analyses.

Relative Competitiveness

Given the limits imposed by the predictability metrics, there was still potential for improvement in the known analyses for MRU, FIFO, etc. We have slightly generalized the notion of *competitiveness* to that of *relative competitiveness*. Relative competitive ratios bound the performance of a policy P relative to the performance of another

policy Q. In the special case of $(1, 0)$-competitiveness, a sound *may* analysis for P is also a sound *may* analysis for Q, and a sound *must* analysis for Q is also a sound *must* analysis for P. By constructing a finite quotient structure of the transition system induced by a pair of policies, we are able to compute competitive ratios automatically. We generalized a number of automatically computed relations to arbitrary associativities. This includes the $(1, 0)$-competitiveness of LRU$(2k - 1)$ and LRU$(2k - 2)$ relative to FIFO(k) and MRU(k), respectively, which yield the first *may* analyses for FIFO and MRU. Notably, these analyses are optimal with respect to the predictability metric *evict*. The $(1, 0)$-competitiveness of PLRU(k) relative to LRU$(\log_2 k + 1)$ concisely explains an existing PLRU analysis of [Heckmann et al., 2003].

Sensitivity of Replacement Policies

Measurement has been proposed as an alternative to static analysis in WCET analysis. The advantage of measurement over static analysis is that it is more easily portable to new architectures, as it does not rely on an abstract model of the architecture. However, its soundness suffers from non-determinism in timing introduced by caches and other performance-enhancing features. To evaluate the influence of the initial state of the cache on the future cache performance and thus the soundness of measurement-based timing analysis, we have introduced the notion of *sensitivity* of a cache replacement policy. Sensitive ratios can be computed automatically similarly to competitive ratios. Analysis reveals that for FIFO, MRU, and PLRU, measurement may strongly underestimate the worst-case number of misses. This may yield WCET estimates that are dramatically wrong. In contrast, LRU is as insensitive to the initial state as possible for any replacement policy.

8.2 Conclusions

Concluding, it is recommended to employ LRU replacement in hard-real time systems. Static timing analysis will benefit in the form of high precision, while measurement-based timing analysis may become sound – at least with respect to cache behavior. If the use of FIFO, PLRU, or MRU is unavoidable, static analyses can benefit from our relative competitiveness results and the induced *may* and *must* analyses. In such cases, measurement-based timing analysis is not recommendable, as it may be dramatically wrong.

8.3 Future Work

The three studies presented in this thesis have been based on purely worst-case assumptions. It would be interesting to examine real software for hard real-time systems and answer the two related questions in such a scenario:

1. How great is the inherent uncertainty about the cache behavior?
2. How strong is the influence of the initial state on the cache performance?

Even for small programs these questions are very difficult to answer precisely, as the collecting cache semantics is in general uncomputable. Other properties of caches like the block size, the write policy, and the allocation policy could also be taken into account in such a study as they have some impact on cache predictability as well.

Static cache analyses have to join abstract states where control-flow merges. Information loss through joins accounts for part of the uncertainty in cache analysis. However, its extent depends on the particular abstract domain. Can we still somehow argue about it in a domain-independent fashion as in the predictability metrics? If not, what are reasonable assumptions about abstract domains used in practice?

The advent of multi-core architectures in embedded systems seems to be inevitable. The memory architecture of such systems has to be designed very carefully to enable sound and precise cache analyses. Unrestricted sharing of caches would likely make cache analysis impossible. What is a good compromise between sharing resources for performance and limiting interferences between different cores and tasks for predictability?

Bibliography

[Ackland et al., 2000] Ackland, B., Anesko, D., Brinthaupt, D., Daubert, S. J., Kalavade, A., Knoblock, J., Micca, E., Moturi, M., Nicol, C. J., O'Neill, J. H., Othmer, J., Sackinger, E., Singh, K. J., Sweet, J., Terman, C. J., and Williams, J. (2000). A single-chip, 1.6 billion, 16-b mac/s multiprocessor dsp,. *IEEE Journal of Solid-state circuits*, 35(3):412–423.

[Ahuja et al., 1993] Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network flows: theory, algorithms, and applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Al-Zoubi et al., 2004] Al-Zoubi, H., Milenkovic, A., and Milenkovic, M. (2004). Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 267–272, New York, NY, USA. ACM Press.

[Aspnes and Waarts, 1996] Aspnes, J. and Waarts, O. (1996). Modular competitiveness for distributed algorithms. In *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 237–246, New York, NY, USA. ACM Press.

[Belady, 1966] Belady, L. (1966). A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, 5:78–101.

[Berg, 2006] Berg, C. (2006). PLRU cache domino effects. In *WCET '06: 6th Intl. Workshop on Worst-Case Execution Time Analysis*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[Bermudo et al., 2000] Bermudo, N., Vera, X., González, A., and Llosa, J. (2000). Optimizing cache miss equations polyhedra. *SIGARCH Comput. Archit. News*, 28(1):43–52.

[Bernat et al., 2002] Bernat, G., Colin, A., and Petters, S. M. (2002). WCET analysis of probabilistic hard real-time systems. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium*, page 279, Washington, DC, USA. IEEE Computer Society.

[Chatterjee et al., 2001] Chatterjee, S., Parker, E., Hanlon, P. J., and Lebeck, A. R. (2001). Exact analysis of the cache behavior of nested loops. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 286–297, New York, NY, USA. ACM Press.

[Cousot and Cousot, 1976] Cousot, P. and Cousot, R. (1976). Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France.

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA. ACM Press.

[Deverge and Puaut, 2005] Deverge, J.-F. and Puaut, I. (2005). Safe measurement-based WCET estimation. In *WCET '05: Proceedings of 5th Intl. Workshop on Worst-Case Execution Time Analysis*, Dagstuhl, Germany. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[Ferdinand, 1997] Ferdinand, C. (1997). *Cache Behaviour Prediction for Real-Time Systems*. PhD thesis, Saarland University.

[Ferdinand et al., 2001] Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., and Wilhelm, R. (2001). Reliable and precise WCET determination for a real-life processor. In *Embedded Software Workshop*, volume 2211, pages 469 – 485, Lake Tahoe, USA.

[Ferdinand et al., 1999] Ferdinand, C., Kästner, D., Langenbach, M., Martin, F., Schmidt, M., Schneider, J., Theiling, H., Thesing, S., and Wilhelm, R. (1999). Runtime guarantees for real-time systems — the USES approach. In *Proceedings of Informatik '99 – Arbeitstagung Programmiersprachen*, Paderborn.

[Ferdinand et al., 1997] Ferdinand, C., Martin, F., and Wilhelm, R. (1997). Applying compiler techniques to cache behavior prediction. In *LCTRTS '97: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, pages 37–46, Las Vegas, Nevada. ACM SIGPLAN.

[Ferdinand and Wilhelm, 1999] Ferdinand, C. and Wilhelm, R. (1999). Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181.

[Fraguela et al., 1999] Fraguela, B. B., Doallo, R., and Zapata, E. L. (1999). Automatic analytical modeling for the estimation of cache misses. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 221, Washington, DC, USA. IEEE Computer Society.

[Freescale Semiconductor Inc., 2002] Freescale Semiconductor Inc. (2002). MPC750 RISC Microprocessor User Manual, Section 3.5.1. `http://www.freescale.com/files/32bit/doc/ref_manual/MPC750UM.pdf`.

[Gebhard and Altmeyer, 2007] Gebhard, G. and Altmeyer, S. (2007). Optimal task placement to improve cache performance. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 259–268, New York, NY, USA. ACM.

[Ghosh et al., 1997] Ghosh, S., Martonosi, M., and Malik, S. (1997). Cache miss equations: an analytical representation of cache misses. In *ICS '97: Proceedings of the 11th International Conference on Supercomputing*, pages 317–324, New York, NY, USA. ACM Press.

[Ghosh et al., 1998] Ghosh, S., Martonosi, M., and Malik, S. (1998). Precise miss analysis for program transformations with caches of arbitrary associativity. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 228–239, New York, NY, USA. ACM Press.

[Ghosh et al., 1999] Ghosh, S., Martonosi, M., and Malik, S. (1999). Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21(4):703–746.

[Grund et al., 2008] Grund, D., Gebhard, G., and Reineke, J. (2008). Timing analysis and predictability of a branch target buffer. (to be submitted).

[Grund and Reineke, 2008] Grund, D. and Reineke, J. (2008). Estimating the performance of cache replacement policies. In *MEMOCODE '08: Proceedings of the 6th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, pages 101–111.

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

[Hennessy and Patterson, 2003] Hennessy, J. L. and Patterson, D. A. (2003). *Computer Architecture: A Quantitative Approach, 3rd Edition*. Morgan Kaufmann.

[Hergenhan and Rosenstiel, 2000] Hergenhan, A. and Rosenstiel, W. (2000). Static timing analysis of embedded software on advanced processor architectures. In *DATE '00: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 552–559, New York, NY, USA. ACM Press.

[Jacob et al., 2007] Jacob, B., Ng, S. W., and Wang, D. T. (2007). *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers.

[Kam and Ullman, 1977] Kam, J. B. and Ullman, J. D. (1977). Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317.

[Kildall, 1973] Kildall, G. A. (1973). A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, New York, NY, USA. ACM.

[Koutsoupias and Papadimitriou, 1994] Koutsoupias, E. and Papadimitriou, C. H. (1994). Beyond competitive analysis. In *FOCS '94: Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 394–400.

[Langenbach et al., 2002] Langenbach, M., Thesing, S., and Heckmann, R. (2002). Pipeline modeling for timing analysis. In *SAS '02: Proceedings of the Static Analyses Symposium*, volume 2477, Madrid, Spain.

[Lawler, 1966] Lawler, E. (1966). Optimal cycles in doubly weighted linear graphs. In *Int'l Symp. Theory of Graphs*, pages 209–213.

[Li et al., 1996] Li, Y.-T. S., Malik, S., and Wolfe, A. (1996). Cache modeling for real-time software: beyond direct mapped instruction caches. In *RTSS '96: Proceedings*

of the 17th IEEE Real-Time Systems Symposium (RTSS '96), page 254, Washington, DC, USA. IEEE Computer Society.

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

[Malamy et al., 1994] Malamy, A., Patel, R., and Hayes, N. (1994). Methods and apparatus for implementing a pseudo-LRU cache memory replacement scheme with a locking feature. United States Patent 5029072.

[Mattson et al., 1970] Mattson, R. L., Gecsei, J., Slutz, D. R., and Traiger, I. L. (1970). Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117.

[Mueller et al., 1994] Mueller, F., Whalley, D. B., and Harmon, M. (1994). Predicting instruction cache behavior. In *LCTRTS '94: Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*.

[Nielson et al., 1999] Nielson, F., Nielson, H. R., and Hankin, C. (1999). *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Peir et al., 1998] Peir, J., Hsu, W. W., and Smith, A. J. (1998). Implementation issues in modern cache memory. Technical report, University of California at Berkeley, Berkeley, CA, USA.

[Petters, 2002] Petters, S. M. (2002). *Worst Case Execution Time Estimation for Advanced Processor Architectures*. PhD thesis, Technische Universität München, Munich, Germany.

[Puaut and Decotigny, 2002] Puaut, I. and Decotigny, D. (2002). Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, page 114, Washington, DC, USA. IEEE Computer Society.

[Ramaprasad and Mueller, 2005] Ramaprasad, H. and Mueller, F. (2005). Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 148–157, Washington, DC, USA. IEEE Computer Society.

[Reineke and Grund, 2008a] Reineke, J. and Grund, D. (2008a). Relative competitive analysis of cache replacement policies. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 51–60, New York, NY, USA. ACM.

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

[Reineke et al., 2007] Reineke, J., Grund, D., Berg, C., and Wilhelm, R. (2007). Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122.

[Reineke et al., 2006] Reineke, J., Wachter, B., Thesing, S., Wilhelm, R., Polian, I., Eisinger, J., and Becker, B. (2006). A definition and classification of timing anomalies. In *WCET '06: Proceedings of 6th International Workshop on Worst-Case Execution Time Analysis*.

[Roy, 2007] Roy, B. V. (2007). A short proof of optimality for the min cache replacement algorithm. *Inf. Process. Lett.*, 102(2-3):72–73.

[Schlickling and Pister, 2007] Schlickling, M. and Pister, M. (2007). A framework for static analysis of VHDL code. In *WCET '07: Proceedings of 7th International Workshop on Worst-Case Execution Time (WCET) Analysis*.

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

[Sleator and Tarjan, 1985] Sleator, D. D. and Tarjan, R. E. (1985). Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208.

[Smaragdakis, 2004] Smaragdakis, Y. (2004). General adaptive replacement policies. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 108–119, New York, NY, USA. ACM.

[Smaragdakis et al., 2003] Smaragdakis, Y., Kaplan, S., and Wilson, P. (2003). The EELRU adaptive replacement algorithm. *Perform. Eval.*, 53(2):93–123.

[Sudarshan et al., 2004] Sudarshan, T., Mir, R. A., and Vijayalakshmi, S. (2004). Highly efficient lru implementations for high associativity cache memory. In *ICACC 04: Proceedings of the 12th IEEE International Conference on Advanced Computing and Communications*, pages 87–95, Ahemdabad, Gujarat, India. Allied Publishers Pvt. Ltd.

[Tarski, 1955] Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications. *Pac. J. Math.*, 5:285–309.

[Theiling, 2003] Theiling, H. (2003). *Control Flow Graphs for Real-Time System Analysis*. PhD thesis, Saarland University.

[Theiling et al., 2000] Theiling, H., Ferdinand, C., and Wilhelm, R. (2000). Fast and precise WCET prediction by separate cache and path analyses. *Real-Time Systems*, 18(2/3).

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

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

[Vera et al., 2003] Vera, X., Lisper, B., and Xue, J. (2003). Data cache locking for higher program predictability. *SIGMETRICS Perform. Eval. Rev.*, 31(1):272–282.

[Vogler, 2008] Vogler, W. (2008). Another short proof of optimality for the min cache replacement algorithm. *Inf. Process. Lett.*, 106(5):219–220.

[Wenzel, 2006] Wenzel, I. (2006). *Measurement-Based Timing Analysis of Superscalar Processors*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria.

[White et al., 1997] White, R. T., Healy, C. A., Whalley, D. B., Mueller, F., and Harmon, M. G. (1997). Timing analysis for data caches and set-associative caches. In *RTAS '97: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, page 192, Washington, DC, USA. IEEE Computer Society.

[Wilhelm et al., 2008] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenström, P. (2008). The worst-case execution-time problem—overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53.

Index

A

Relative Competitiveness Results

Running our tool on a Core 2 Duo E6750 at 2.66GHz with 2GB of RAM, we have obtained a vast amount of competitiveness results for LRU, FIFO, MRU PLRU, and OPT at associativities ranging from 2 to 8. To avoid any paging activity, we have limited the heap space to be allocated by our tool to 1.5GB. If the computation required more than 30 minutes or exceeded the heap space limit of 1.5GB, we report MEM or TIME, respectively.

Miss-Competitiveness

PLRU \ LRU	2	3	4	5	6	7	8
2	$1, 0$	∞	∞	∞	∞	∞	∞
4	$1, 0$	$1, 0$	∞	∞	∞	∞	∞
8	$1, 0$	$1, 0$	$1, 0$	∞	∞	∞	∞

(a) Miss-Competitiveness: PLRU vs LRU

PLRU \ FIFO	2	3	4	5	6	7	8
2	$2, 1$	∞	∞	∞	∞	∞	∞
4	$1, 0$	$2, 2$	∞	∞	∞	∞	∞
8	$1, 0$	$\frac{4}{3}, 1$	$2, 3$	∞	∞	∞	∞

(b) Miss-Competitiveness: PLRU vs FIFO

PLRU \ MRU	2	3	4	5	6	7	8
2	$1, 0$	∞	∞	∞	∞	∞	∞
4	$1, 0$	$2, 1$	∞	∞	∞	∞	∞
8	$1, 0$	$1, 0$	$2, 1$	∞	∞	MEM	MEM

(c) Miss-Competitiveness: PLRU vs MRU

PLRU \ OPT	2	3	4	5	6	7	8
2	$2, 1$	∞	∞	∞	∞	∞	∞
4	$\frac{4}{3}, 1$	$2, 2$	∞	∞	∞	∞	∞
8	$\frac{8}{7}, 1$	MEM	MEM	MEM	MEM	MEM	MEM

(d) Miss-Competitiveness: PLRU vs OPT

Figure A.1: Miss-Competitiveness results relating PLRU with LRU, FIFO, MRU, and OPT. The first component of each cell denotes the competitive ratio of the policy specified in the row relative to the policy specified in the column. The second component shows the additive constant. As an example, PLRU(8) is $(\frac{4}{3}, 1)$-miss-competitive relative to FIFO(3).

LRU \ FIFO	2	3	4	5	6	7	8
2	$2,1$	∞	∞	∞	∞	∞	∞
3	$1,0$	$3,2$	∞	∞	∞	∞	∞
4	$1,0$	$2,1$	$4,3$	∞	∞	∞	∞
5	$1,0$	$1,0$	$2,2$	$5,4$	∞	∞	∞
6	$1,0$	$1,0$	$2,1$	$3,3$	$6,5$	∞	∞
7	$1,0$	$1,0$	$1,0$	$2,2$	$3,4$	$7,6$	∞
8	$1,0$	$1,0$	$1,0$	$2,1$	$2,3$	$4,5$	$8,7$

(a) Miss-Competitiveness: LRU vs FIFO

LRU \ MRU	2	3	4	5	6	7	8
2	$1,0$	∞	∞	∞	∞	∞	∞
3	$1,0$	$2,1$	∞	∞	∞	∞	∞
4	$1,0$	$1,0$	$3,2$	∞	∞	∞	∞
5	$1,0$	$1,0$	$\frac{3}{2},1$	$4,3$	∞	∞	∞
6	$1,0$	$1,0$	$1,0$	$2,2$	$5,4$	∞	∞
7	$1,0$	$1,0$	$1,0$	$\frac{4}{3},1$	$\frac{5}{2},3$	$6,5$	∞
8	$1,0$	$1,0$	$1,0$	$1,0$	$\frac{5}{3},2$	$3,4$	$7,6$

(b) Miss-Competitiveness: LRU vs MRU

LRU \ OPT	2	3	4	5	6	7	8
2	$2,1$	∞	∞	∞	∞	∞	∞
3	$\frac{3}{2},1$	$3,2$	∞	∞	∞	∞	∞
4	$\frac{4}{3},1$	$2,2$	$4,3$	∞	∞	∞	∞
5	$\frac{5}{4},1$	$\frac{5}{3},2$	$\frac{5}{2},3$	$5,4$	∞	∞	∞
6	$\frac{6}{5},1$	$\frac{3}{2},2$	$2,3$	MEM	TIME	∞	∞
7	$\frac{7}{6},1$	$\frac{7}{5},2$	MEM	MEM	MEM	MEM	∞
8	$\frac{8}{7},1$	MEM	MEM	MEM	MEM	MEM	MEM

(c) Miss-Competitiveness: LRU vs OPT

LRU \ PLRU	2	4	8
2	$1,0$	∞	∞
3	$1,0$	∞	∞
4	$1,0$	$2,1$	∞
5	$1,0$	$\frac{3}{2},1$	∞
6	$1,0$	$\frac{4}{3},1$	∞
7	$1,0$	$\frac{5}{4},1$	∞
8	$1,0$	$\frac{6}{5},1$	$5,4$

(d) Miss-Competitiveness: LRU vs PLRU

Figure A.2: Miss-Competitiveness results relating LRU with FIFO, MRU, OPT, and PLRU. The first component of each cell denotes the competitive ratio of the policy specified in the row relative to the policy specified in the column. The second component shows the additive constant. As an example, LRU(6) is $(3,3)$-miss-competitive relative to FIFO(5).

FIFO \ LRU	2	3	4	5	6	7	8
2	$2,1$	∞	∞	∞	∞	∞	∞
3	$\frac{3}{2},1$	$3,2$	∞	∞	∞	∞	∞
4	$\frac{4}{3},1$	$2,2$	$4,3$	∞	∞	∞	∞
5	$\frac{5}{4},1$	$\frac{5}{3},2$	$\frac{5}{2},3$	$5,4$	∞	∞	∞
6	$\frac{6}{5},1$	$\frac{3}{2},2$	$2,3$	$3,4$	$6,5$	∞	∞
7	$\frac{7}{6},1$	$\frac{7}{5},2$	$\frac{7}{4},3$	$\frac{7}{3},4$	$\frac{7}{2},5$	$7,6$	∞
8	$\frac{8}{7},1$	$\frac{4}{3},2$	$\frac{8}{5},3$	$2,4$	$\frac{8}{3},5$	$4,6$	$8,7$

(a) Miss-Competitiveness: FIFO vs LRU

FIFO \ MRU	2	3	4	5	6	7	8
2	$2,1$	∞	∞	∞	∞	∞	∞
3	$\frac{3}{2},1$	$3,3$	∞	∞	∞	∞	∞
4	$\frac{4}{3},1$	$2,3$	$4,4$	∞	∞	∞	∞
5	$\frac{5}{4},1$	$\frac{5}{3},3$	$\frac{5}{2},4$	$5,5$	∞	∞	∞
6	$\frac{6}{5},1$	$\frac{3}{2},3$	$2,4$	$3,5$	$6,6$	∞	∞
7	$\frac{7}{6},1$	$\frac{7}{5},3$	$\frac{7}{4},4$	$\frac{7}{3},5$	$\frac{7}{2},6$	MEM	∞
8	$\frac{8}{7},1$	$\frac{4}{3},3$	$\frac{8}{5},4$	$2,5$	MEM	TIME	MEM

(b) Miss-Competitiveness: FIFO vs MRU

FIFO \ OPT	2	3	4	5	6	7	8
2	$2,1$	∞	∞	∞	∞	∞	∞
3	$\frac{3}{2},2$	$3,3$	∞	∞	∞	∞	∞
4	$\frac{4}{3},2$	$2,3$	$4,4$	∞	∞	∞	∞
5	$\frac{5}{4},2$	$\frac{5}{3},3$	$\frac{5}{2},4$	MEM	∞	∞	∞
6	$\frac{6}{5},2$	$\frac{3}{2},3$	MEM	MEM	MEM	∞	∞
7	$\frac{7}{6},2$	MEM	MEM	MEM	MEM	MEM	∞
8	MEM	MEM	MEM	MEM	MEM	MEM	MEM

(c) Miss-Competitiveness: FIFO vs OPT

FIFO \ PLRU	2	4	8
2	$2,1$	∞	∞
3	$\frac{3}{2},1$	∞	∞
4	$\frac{4}{3},1$	$4,4$	∞
5	$\frac{5}{4},1$	$\frac{5}{2},4$	∞
6	$\frac{6}{5},1$	$2,4$	∞
7	$\frac{7}{6},1$	$\frac{7}{4},4$	∞
8	$\frac{8}{7},1$	$\frac{8}{5},4$	$8,8$

(d) Miss-Competitiveness: FIFO vs PLRU

Figure A.3: Miss-Competitiveness results relating FIFO with LRU, MRU, OPT, and PLRU. The first component of each cell denotes the competitive ratio of the policy specified in the row relative to the policy specified in the column. The second component shows the additive constant. As an example, FIFO(6) is $(2,3)$-miss-competitive relative to LRU(4).

MRU \ LRU	2	3	4	5	6	7	8
2	$1,0$	∞	∞	∞	∞	∞	∞
3	$1,0$	$2,1$	∞	∞	∞	∞	∞
4	$1,0$	$\frac{3}{2},1$	$3,2$	∞	∞	∞	∞
5	$1,0$	$\frac{4}{3},1$	$2,2$	$4,3$	∞	∞	∞
6	$1,0$	$\frac{5}{4},1$	$\frac{5}{3},2$	$\frac{5}{2},3$	$5,4$	∞	∞
7	$1,0$	$\frac{6}{5},1$	$\frac{3}{2},2$	$2,3$	$3,4$	$6,5$	∞
8	$1,0$	$\frac{7}{6},1$	$\frac{7}{5},2$	$\frac{7}{4},3$	$\frac{7}{3},4$	$\frac{7}{2},5$	$7,6$

(a) Miss-Competitiveness: MRU vs LRU

MRU \ FIFO	2	3	4	5	6	7	8
2	$2,1$	∞	∞	∞	∞	∞	∞
3	$1,0$	$4,3$	∞	∞	∞	∞	∞
4	$1,0$	$3,2$	$6,5$	∞	∞	∞	∞
5	$1,0$	$\frac{3}{2},1$	$4,4$	$8,7$	∞	∞	∞
6	$1,0$	$\frac{11}{10},\frac{7}{5}$	$2,6$	$5,6$	$10,9$	∞	∞
7	$1,0$	$1,1$	$\frac{5}{3},2$	$3,5$	$6,8$	MEM	∞
8	$1,0$	$1,1$	$\frac{32}{27},\frac{70}{27}$	$2,5$	MEM	TIME	MEM

(b) Miss-Competitiveness: MRU vs FIFO

MRU \ OPT	2	3	4	5	6	7	8
2	$2,1$	∞	∞	∞	∞	∞	∞
3	$2,1$	$4,3$	∞	∞	∞	∞	∞
4	$\frac{3}{2},\frac{3}{2}$	$3,3$	$6,5$	∞	∞	∞	∞
5	$\frac{4}{3},\frac{5}{3}$	$2,3$	MEM	MEM	∞	∞	∞
6	$\frac{5}{4},\frac{7}{4}$	MEM	MEM	MEM	MEM	∞	∞
7	MEM	MEM	MEM	MEM	MEM	MEM	∞
8	MEM	MEM	MEM	MEM	MEM	MEM	MEM

(c) Miss-Competitiveness: MRU vs OPT

MRU \ PLRU	2	4	8
2	$1,0$	∞	∞
3	$1,0$	∞	∞
4	$1,0$	$4,3$	∞
5	$1,0$	$2,3$	∞
6	$1,0$	$\frac{5}{3},3$	∞
7	$1,0$	$\frac{3}{2},3$	∞
8	$1,0$	$\frac{7}{5},3$	TIME

(d) Miss-Competitiveness: MRU vs PLRU

Figure A.4: Miss-Competitiveness results relating MRU with LRU, FIFO, OPT, and PLRU. The first component of each cell denotes the competitive ratio of the policy specified in the row relative to the policy specified in the column. The second component shows the additive constant. As an example, MRU(6) is $(\frac{5}{3},2)$-miss-competitive relative to LRU(4).

Hit-Competitiveness

Online policies can be miss-competitive relative to OPT. In contrast, no online policy can be hit-competitive relative to OPT. Therefore, we have only computed hit-competitiveness among LRU, PLRU, FIFO, and MRU.

PLRU \ LRU	2	3	4	5	6	7	8
2	$1, 0$	0	0	0	0	0	0
4	$1, 0$	$1, 0$	$\frac{1}{2}, 1$	0	0	0	0
8	$1, 0$	$1, 0$	$1, 0$	$\frac{2}{3}, \frac{4}{3}$	$\frac{1}{2}, \frac{3}{2}$	$\frac{2}{5}, \frac{8}{5}$	$\frac{1}{4}, \frac{3}{2}$

(a) Hit-Competitiveness: PLRU vs LRU

PLRU \ FIFO	2	3	4	5	6	7	8
2	0	0	0	0	0	0	0
4	$1, 0$	0	0	0	0	0	0
8	$1, 0$	$\frac{3}{4}, 1$	$\frac{1}{2}, \frac{3}{2}$	0	0	0	0

(b) Hit-Competitiveness: PLRU vs FIFO

PLRU \ MRU	2	3	4	5	6	7	8
2	$1, 0$	0	0	0	0	0	0
4	$1, 0$	$\frac{1}{2}, \frac{1}{2}$	0	0	0	0	0
8	$1, 0$	$1, 0$	$\frac{2}{3}, \frac{4}{3}$	$\frac{1}{3}, 1$	0	MEM	MEM

(c) Hit-Competitiveness: PLRU vs MRU

Figure A.5: Hit-Competitiveness results relating PLRU with LRU, FIFO, and MRU. The first component of each cell denotes the competitive ratio of the policy specified in the row relative to the policy specified in the column. The second component shows the subtractive constant. As an example, PLRU(8) is $(\frac{2}{3}, \frac{4}{3})$-hit-competitive relative to LRU(5).

LRU \ FIFO	2	3	4	5	6	7	8
2	0	0	0	0	0	0	0
3	$1,0$	0	0	0	0	0	0
4	$1,0$	0	0	0	0	0	0
5	$1,0$	$1,0$	0	0	0	0	0
6	$1,0$	$1,0$	0	0	0	0	0
7	$1,0$	$1,0$	$1,0$	0	0	0	0
8	$1,0$	$1,0$	$1,0$	0	0	0	0

(a) Hit-Competitiveness: LRU vs FIFO

LRU \ MRU	2	3	4	5	6	7	8
2	$1,0$	0	0	0	0	0	0
3	$1,0$	0	0	0	0	0	0
4	$1,0$	$1,0$	0	0	0	0	0
5	$1,0$	$1,0$	0	0	0	0	0
6	$1,0$	$1,0$	$1,0$	0	0	0	0
7	$1,0$	$1,0$	$1,0$	0	0	0	0
8	$1,0$	$1,0$	$1,0$	$1,0$	0	0	0

(b) Hit-Competitiveness: LRU vs MRU

LRU \ PLRU	2	4	8
2	$1,0$	0	0
3	$1,0$	0	0
4	$1,0$	$\frac{1}{2},1$	0
5	$1,0$	$\frac{2}{3},1$	0
6	$1,0$	$\frac{3}{4},1$	0
7	$1,0$	$\frac{4}{5},1$	0
8	$1,0$	$\frac{5}{6},1$	$\frac{1}{8},\frac{15}{8}$

(c) Hit-Competitiveness: LRU vs PLRU

Figure A.6: Hit-Competitiveness results relating LRU with FIFO, MRU, and PLRU. The first component of each cell denotes the competitive ratio of the policy specified in the row relative to the policy specified in the column. The second component shows the subtractive constant. As an example, LRU(8) is $(\frac{5}{6},1)$-hit-competitive relative to PLRU(4).

FIFO \ LRU	2	3	4	5	6	7	8
2	$\frac{1}{2}, \frac{1}{2}$	0	0	0	0	0	0
3	$\frac{2}{3}, \frac{2}{3}$	$\frac{1}{2}, 1$	0	0	0	0	0
4	$\frac{3}{4}, \frac{3}{4}$	$\frac{1}{2}, 1$	$\frac{1}{2}, \frac{3}{2}$	0	0	0	0
5	$\frac{4}{5}, \frac{4}{5}$	$\frac{2}{3}, \frac{4}{3}$	$\frac{1}{2}, \frac{3}{2}$	$\frac{1}{2}, 2$	0	0	0
6	$\frac{5}{6}, \frac{5}{6}$	$\frac{2}{3}, \frac{4}{3}$	$\frac{1}{2}, \frac{3}{2}$	$\frac{1}{2}, 2$	$\frac{1}{2}, \frac{5}{2}$	0	0
7	$\frac{6}{7}, \frac{6}{7}$	$\frac{3}{4}, \frac{3}{2}$	$\frac{2}{3}, 2$	$\frac{1}{2}, 2$	$\frac{1}{2}, \frac{5}{2}$	$\frac{1}{2}, 3$	0
8	$\frac{7}{8}, \frac{7}{8}$	$\frac{3}{4}, \frac{3}{2}$	$\frac{2}{3}, 2$	$\frac{1}{2}, 2$	$\frac{1}{2}, \frac{5}{2}$	$\frac{1}{2}, 3$	$\frac{1}{2}, \frac{7}{2}$

(a) Hit-Competitiveness: FIFO vs LRU

FIFO \ MRU	2	3	4	5	6	7	8
2	$\frac{1}{2}, \frac{1}{2}$	0	0	0	0	0	0
3	$\frac{2}{3}, \frac{2}{3}$	0	0	0	0	0	0
4	$\frac{3}{4}, \frac{3}{4}$	$\frac{1}{2}, \frac{3}{2}$	0	0	0	0	0
5	$\frac{4}{5}, \frac{4}{5}$	$\frac{1}{2}, \frac{3}{2}$	0	0	0	0	0
6	$\frac{5}{6}, \frac{5}{6}$	$\frac{1}{2}, \frac{3}{2}$	$\frac{1}{2}, 2$	0	0	0	0
7	$\frac{6}{7}, \frac{6}{7}$	$\frac{2}{3}, 2$	$\frac{1}{2}, 2$	0	0	MEM	0
8	$\frac{7}{8}, \frac{7}{8}$	$\frac{2}{3}, 2$	$\frac{1}{2}, 2$	$\frac{1}{2}, \frac{5}{2}$	MEM	TIME	MEM

(b) Hit-Competitiveness: FIFO vs MRU

FIFO \ PLRU	2	4	8
2	$\frac{1}{2}, \frac{1}{2}$	0	0
3	$\frac{2}{3}, \frac{2}{3}$	0	0
4	$\frac{3}{4}, \frac{3}{4}$	$\frac{1}{4}, \frac{5}{4}$	0
5	$\frac{4}{5}, \frac{4}{5}$	$\frac{3}{7}, \frac{13}{7}$	0
6	$\frac{5}{6}, \frac{5}{6}$	$\frac{1}{2}, 2$	0
7	$\frac{6}{7}, \frac{6}{7}$	$\frac{3}{5}, \frac{13}{5}$	0
8	$\frac{7}{8}, \frac{7}{8}$	$\frac{2}{3}, \frac{8}{3}$	$\frac{1}{11}, \frac{19}{11}$

(c) Hit-Competitiveness: FIFO vs PLRU

Figure A.7: Hit-Competitiveness results relating FIFO with LRU, MRU, and PLRU. The first component of each cell denotes the competitive ratio of the policy specified in the row relative to the policy specified in the column. The second component shows the subtractive constant. As an example, FIFO(7) is $(\frac{3}{4}, \frac{3}{2})$-hit-competitive relative to LRU(4).

MRU \ LRU	2	3	4	5	6	7	8
2	$1, 0$	0	0	0	0	0	0
3	$1, 0$	0	0	0	0	0	0
4	$1, 0$	$\frac{1}{2}, \frac{1}{2}$	0	0	0	0	0
5	$1, 0$	$\frac{1}{2}, \frac{1}{2}$	$\frac{1}{3}, 1$	0	0	0	0
6	$1, 0$	$\frac{2}{3}, \frac{2}{3}$	$\frac{1}{2}, 1$	$\frac{1}{4}, \frac{5}{4}$	0	0	0
7	$1, 0$	$\frac{2}{3}, \frac{2}{3}$	$\frac{1}{2}, 1$	$\frac{2}{5}, \frac{8}{5}$	$\frac{1}{5}, \frac{7}{5}$	0	0
8	$1, 0$	$\frac{3}{4}, \frac{3}{4}$	$\frac{2}{3}, \frac{4}{3}$	$\frac{1}{2}, \frac{3}{2}$	$\frac{1}{3}, 2$	$\frac{1}{6}, \frac{3}{2}$	0

(a) Hit-Competitiveness: MRU vs LRU

MRU \ FIFO	2	3	4	5	6	7	8
2	0	0	0	0	0	0	0
3	$1, 0$	0	0	0	0	0	0
4	$1, 0$	0	0	0	0	0	0
5	$1, 0$	$\frac{1}{2}, 1$	0	0	0	0	0
6	$1, 0$	$\frac{3}{4}, \frac{3}{4}$	0	0	0	0	0
7	$1, 0$	$1, 1$	$\frac{1}{3}, \frac{4}{3}$	0	0	MEM	0
8	$1, 0$	$1, 1$	$\frac{2}{3}, 2$	0	MEM	MEM	MEM

(b) Hit-Competitiveness: MRU vs FIFO

MRU \ PLRU	2	4	8
2	$1, 0$	0	0
3	$1, 0$	0	0
4	$1, 0$	0	0
5	$1, 0$	$\frac{1}{4}, 1$	0
6	$1, 0$	$\frac{1}{2}, \frac{3}{2}$	0
7	$1, 0$	$\frac{1}{2}, \frac{3}{2}$	0
8	$1, 0$	$\frac{3}{5}, \frac{9}{5}$	MEM

(c) Hit-Competitiveness: MRU vs PLRU

Figure A.8: Hit-Competitiveness results relating MRU with LRU, FIFO, and PLRU. The first component of each cell denotes the competitive ratio of the policy specified in the row relative to the policy specified in the column. The second component shows the subtractive constant. As an example, MRU(8) is $(\frac{2}{3}, \frac{4}{3})$-hit-competitive relative to LRU(4).

B

Non-Distributivity of Ferdinand's LRU Analysis

It has been an open question, whether Christian Ferdinand's LRU analysis, described in Chapter 4, is distributive. We present two small counter-examples that show that both the *may* and the *must* analysis are not distributive.

For the non-distributivity of the *may* and the *must* analysis consider the following two abstract states:

$$\begin{aligned} \widehat{p} &= [\{a\},\{c\},\{b\}] \\ \widehat{q} &= [\{b\},\{a\},\{c\}] \end{aligned}$$

Accessing b in $\widehat{p}$ and $\widehat{q}$, respectively, yields

$$\begin{aligned} update^{\mathrm{LRU}}_{must}(\widehat{p},b) = update^{\mathrm{LRU}}_{may}(\widehat{p},b) &= [\{b\},\{a\},\{c\}] \\ update^{\mathrm{LRU}}_{must}(\widehat{q},b) = update^{\mathrm{LRU}}_{may}(\widehat{q},b) &= [\{b\},\{a\},\{c\}] \end{aligned}$$

whose join is

$$\begin{aligned} update^{\mathrm{LRU}}_{must}(\widehat{p},b) \sqcup^{\mathrm{LRU}}_{must} update^{\mathrm{LRU}}_{must}(\widehat{q},b) &= \\ update^{\mathrm{LRU}}_{may}(\widehat{p},b) \sqcup^{\mathrm{LRU}}_{may} update^{\mathrm{LRU}}_{may}(\widehat{q},b) &= [\{b\},\{a\},\{c\}]\,. \end{aligned}$$

The *may* and *must* joins of $\widehat{p}$ and $\widehat{q}$ are

$$\begin{aligned} \widehat{p} \sqcup^{\mathrm{LRU}}_{must} \widehat{q} &= [\{\},\{a\},\{b,c\}] \\ \widehat{p} \sqcup^{\mathrm{LRU}}_{may} \widehat{q} &= [\{a,b\},\{c\},\{\}] \end{aligned}$$

Accessing b in these states yields

$$\begin{aligned} update^{\mathrm{LRU}}_{must}([\{\},\{a\},\{b,c\}],b) &= [\{b\},\{\},\{a,c\}] \\ update^{\mathrm{LRU}}_{may}([\{a,b\},\{c\},\{\}],b) &= [\{b\},\{a,c\},\{\}] \end{aligned}$$

which proves the non-distributivity of Ferdinand's LRU *may* and *must* analyses. This is unfortunate, as the *Abstract collecting path semantics* induced by the abstract transformers $update^{\mathrm{LRU}}_{must}$ and $update^{\mathrm{LRU}}_{may}$ and therefore the *MOP* solution are exact, i.e., the abstract domain does not introduce additional uncertainty beyond that inherent to the collecting semantics. Due to the non-distributivity this does not carry over to the *MFP* solution.

C Mathematical Foundations

This chapter introduces the mathematical notions and fundamental theorems underlying abstract interpretation and data-flow analysis. It is loosely based on [Nielson et al., 1999]. These foundations are mainly used in the introduction to abstract interpretation in Chapter 3. Where possible, we try to intuitively describe the utility of the notions in the course of introducing them.

Analysis domains are usually partially ordered sets:

Definition C.1 (Partial order, partially ordered set)**.** *A binary relation $\sqsubseteq \subseteq L \times L$ is called a* partial order*, if and only if it is*

- *reflexive:* $\forall l \in L : l \sqsubseteq l$,
- *antisymmetric:* $\forall l_1, l_2 \in L : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$,
- *transitive:* $\forall l_1, l_2, l_3 \in L : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$.

A set with a partial order $(L, \sqsubseteq)$ is called a partially ordered set*.*

When used as an analysis domain, the partial order $\sqsubseteq$ usually models precision: If $a \sqsubseteq b$, then a is more precise than b, i.e. b represents more concrete states than a.

Definition C.2 (Upper bounds, lower bounds)**.** *A subset Y of L has $l \in L$ as an* upper bound *if $\forall l' \in Y : l' \sqsubseteq l$. Y has $l \in L$ as a* lower bound *if $\forall l' \in Y : l \sqsubseteq l'$.*

A least upper bound *l of Y is an upper bound of Y that is smaller than all other upper bounds of Y. Similarly, a* greatest lower bound *l of Y is a lower bound of Y that is greater than all other lower bounds of Y.*

If a least upper bound of a subset Y of L exists, it is unique (due to the antisymmetry of $\sqsubseteq$) and denoted by $\bigsqcup Y$. Similarly, if Y has a greatest lower bound, it is unique and denoted by $\bigsqcap Y$. We also call $\bigsqcup$ join *and $\bigsqcap$* meet*.*

Least upper bounds and greatest lower bounds of pairs $l, l' \in L$ are denoted $l \sqcup l' = \bigsqcup\{l, l'\}$ and $l \sqcap l' = \bigsqcap\{l, l'\}$, respectively.

Analyses need to safely combine information. $a \sqcup b$ is the most precise element in the partially ordered set that represents all concrete states that a and b represent. However, in partially ordered sets, $a \sqcup b$ need not exist. There may be several upper bounds on a and b, none of which is the smallest. To always be able to safely and – within the

limits of the domain – precisely combine analysis information, analysis domains should be *complete lattices*:

Definition C.3 (Complete lattice)**.** *A* complete lattice $L = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ *is a partially ordered set* $(L, \sqsubseteq)$ *such that all subsets have least upper as well as greatest lower bounds. Furthermore,* $\bot = \bigsqcup \varnothing = \bigsqcap L$ *is the* least element. $\top = \bigsqcap \varnothing = \bigsqcup L$ *is the* greatest element.

Sometimes, partial orders are only closed under least upper bounds or greatest lower bounds:

Definition C.4 (Semilattice)**.** *A* join-semilattice $L = (L, \sqsubseteq, \bigsqcup, \top)$ *is a partially ordered set* $(L, \sqsubseteq)$ *such that all subsets have least upper bounds and* $\top = \bigsqcup L$ *is the* greatest element. *A* meet-semilattice $L = (L, \sqsubseteq, \bigsqcap, \bot)$ *is a partially ordered set* $(L, \sqsubseteq)$ *such that all subsets have greatest lower bounds and* $\bot = \bigsqcap L$ *is the* least element.

Example (Powerset lattice)**.** *An example of a* complete lattice *is the* powerset lattice $L = (2^S, \subseteq, \bigcup, \bigcap, \varnothing, S)$. *The subset relation* $\subseteq$ *is a partial order. In the* $\subseteq$ *relation,* $\varnothing$ *is the least element, and* S *is the greatest element. Furthermore, the least upper bound of a set of subsets is simply the union* $\bigcup$ *of these sets. Intersection* $\bigcap$ *determines the greatest lower bound of a set of subsets of* S.

We associate analysis results in some complete lattice L with program points in a set S. This induces another complete lattice:

Definition C.5 (Total function space)**.** *Let* S *be a set and* $L = (L, \sqsubseteq_L, \bigsqcup_L, \bigsqcap_L, \bot_L, \top_L)$ *be a complete lattice. Define* M *as the* total function space *from* S *to* L*:*

$$M = S \to L.$$

Then, $M = (M, \sqsubseteq_M, \bigsqcup_M, \bigsqcap_M, \bot_M, \top_M)$ *is a complete lattice, with*

$$f \sqsubseteq_M f' \Leftrightarrow \forall s \in S : f(s) \sqsubseteq_L f'(s),$$

$$\bigsqcup_M Y = \lambda s. \bigsqcup_L \{f(s) \mid f \in Y\},$$

and $\bot_M := \lambda s. \bot_L$ *and similarly for* $\bigsqcap_M$ *and* $\top_M$.

Functions are used to represent the concrete and abstract semantics and to relate concrete and abstract domains. The following properties of functions are often necessary or desirable:

Definition C.6 (Properties of functions)**.** *A function* $f : L_1 \to L_2$*, where* $(L_1, \sqsubseteq_1)$ *and* $(L_2, \sqsubseteq_2)$ *are partially ordered sets, is*

- monotone *(or isotone) if* $\forall l, l' \in L_1 : l \sqsubseteq_1 l' \Rightarrow f(l) \sqsubseteq_2 f(l')$,
- distributive *(or additive) if* $\forall l, l' \in L_1 : f(l \sqcup_1 l') = f(l) \sqcup_2 f(l')$,

The semantics or approximations of the semantics of programs can be formulated as fixed points:

Definition C.7 (Fixed points)**.** *Given a a complete lattice* $L = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ *and a monotone function* $f : L \to L$ *on* L, $l \in L$ *is a* fixed point of f *if and only if* $f(l) = l$. *We write*

$$Fix(f) := \{l \mid f(l) = l\},$$

for the set of fixed points. The function f *is* reductive at l *if and only if* $f(l) \sqsubseteq l$ *and we write*

$$Red(f) := \{l \mid f(l) \sqsubseteq l\},$$

for the subset of L *on which* f *is reductive. The function* f *is* extensive at l *if and only if* $l \sqsubseteq f(l)$. *We write*

$$Ext(f) := \{l \mid l \sqsubseteq f(l)\}.$$

for the set of elements on which f *is extensive.*

Since L *is a complete lattice, every subset of* L *has a greatest lower bound and a least upper bound in* L, *in particular* $Fix(f)$. *We denote the greatest lower bound of* $Fix(f)$ *by* $lfp(f)$

$$lfp(f) = \bigsqcap Fix(f)$$

and the least upper bound of $Fix(f)$ *by* $gfp(f)$

$$gfp(f) = \bigsqcup Fix(f).$$

Tarski [Tarski, 1955] showed that $lfp(f)$ and $gfp(f)$ are the *least fixed point of* f and the *greatest fixed point of* f, respectively:

Theorem C.8 (Tarski's fixed point theorem, [Nielson et al., 1999])**.** *Let* $L = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ *be a complete lattice and* $f : L \to L$ *be a monotone function. Then* $lfp(f)$ *and* $gfp(f)$ *are themselves fixed points:*

$$lfp(f) = \bigsqcap Red(f) \in Fix(f)$$

$$gfp(f) = \bigsqcup Ext(f) \in Fix(f).$$

Proof. See [Nielson et al., 1999] □

Tarski's fixed point theorem guarantees the existence of least and greatest fixed points. To determine abstract semantics, one needs to *compute* least fixed points. The least fixed point of a function $f : L \to L$ can be computed if the complete lattice L satisfies the *ascending chain condition*:

Definition C.9 (Chains)**.** *A subset $Y \subseteq L$ of a partially ordered set $(L, \sqsubseteq)$ is a* chain *if $\forall l_1, l_2 \in Y : (l_1 \sqsubseteq l_2) \vee (l_2 \sqsubseteq l_1)$. In other words, a* chain *is a totally ordered subset of L.*

A sequence (l_n) of elements in L is an ascending chain *if $n \leq m \Rightarrow l_n \sqsubseteq l_m$. Similarly, a sequence (l_n) of elements in L is a* descending chain *if $n \leq m \Rightarrow l_m \sqsubseteq l_n$.*

A sequence eventually stabilizes *if and only if $\exists n_0 \in \mathbb{N} : \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow l_n = l_{n_0}$.*

A partially ordered set $(L, \sqsubseteq)$ satisfies the ascending chain condition *if all ascending chains eventually stabilize. Similarly, a partially ordered set $(L, \sqsubseteq)$ satisfies the* descending chain condition *if all descending chains eventually stabilize.*

The ascending chain condition transfers to total function spaces:

Lemma C.10 (Total function space, ascending chain condition)**.** *Let S be a* finite *set and $L = (L, \sqsubseteq_L, \bigsqcup_L, \bigsqcap_L, \bot_L, \top_L)$ be a complete lattice. If L satisfies the ascending chain condition, then so does the total function space from S To L, $M = S \to L$.*

Proof. Let n be the length of the longest ascending chain in L and $|S|$ be the size of S. Then, $n \cdot |S|$ is a bound on the length of ascending chains in M. In every "step" of any ascending chain, one of the functions' values must have increased. After $n \cdot |S|$ "steps" each of the functions' values must have increased at least n times and no further increases are possible. □

Kleene's fixed point theorem provides a way to iteratively compute least and greatest fixed points:

Theorem C.11 (Kleene's fixed point theorem)**.** *If a lattice $L = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ satisfies the ascending chain condition, one can compute the least fixed point of a monotone function $f : L \to L$ by repeated application of f to $\bot$:*

$$\exists i \in \mathbb{N} : lfp(f) = f^i(\bot)$$

The greatest fixed point can similarly be computed by repeated application of f to $\top$, if the lattice satisfies the descending chain condition:

$$\exists i \in \mathbb{N} : gfp(f) = f^i(\top)$$

Proof. Trivially, $\bot \sqsubseteq f(\bot)$. Due to the monotonicity of f, also $f^1(\bot) \sqsubseteq f^2(\bot)$ and by induction $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$. As $\sqsubseteq$ is transitive by definition, the sequence $(f^n(\bot))_n$ is an ascending chain. As L satisfies the ascending chain condition, this sequence must eventually stabilize, i.e. there is an n_0 such that $f(f^{n_0}(\bot)) = f^{n_0}(\bot)$ is a fixed point. $f^{n_0}(\bot)$ is also the least fixed point: trivially $\bot \sqsubseteq lfp(f)$ and by monotonicity of f and induction, $f^i(\bot) \sqsubseteq f^i(lfp(f)) = lfp(f)$ for all i, in particular for $i = n_0$.
Similar arguments prove $gfp(f) = f^i(\top)$ for some i. □