A Survey on Cache Analysis for Real-Time Systems

MINGSONG LV, Northeastern University, China
NAN GUAN, Northeastern University, China
WANG YI, Uppsala University, Sweden
JAN REINEKE, Saarland University, Germany
REINHARD WILHELM, Saarland University, Germany

Real-time systems are reactive computer systems that must produce their reaction to a stimulus within given time bounds. A vital verification requirement is to estimate the Worst-Case Execution Time (WCET) of programs. These estimates are then used to predict the timing behavior of the overall system. The execution time of a program heavily depends on the underlying hardware, among which the cache has the biggest influence. Analyzing the cache behavior is very challenging due to the versatile cache features and complex execution environments. This article provides a survey on cache analysis for real-time systems. We first present the challenges and analysis techniques for independent programs with respect to different cache features. Then, the discussion is extended to cache analysis in complex execution environments, followed by a survey of existing tools based on static cache analysis. An outlook for future research is provided at last.

Categories and Subject Descriptors: J.7 [**Computers in Other Systems**]; C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems

General Terms: Verification, Reliability

Additional Key Words and Phrases: Hard real-time, cache analysis, worst-case execution time

ACM Reference Format:
Mingsong Lv, Nan Guan, Wang Yi, Jan Reineke, and Reinhard Wilhelm. 2015. A Survey on Cache Analysis for Real-Time Systems. *ACM Comput. Surv.* 0, 0, Article 0 (0), 45 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Real-time embedded systems not only exist in industry domains, such as automotive electronics, avionics, telecommunication, medical systems, etc., but are deeply immersed in our everyday life due to the rapid progress of mobile and embedded technology. A real-time system should not only provide logically correct functions, but moreover, it must meet *timing requirements* as stated in the system specification [Buttazzo 2004]. In hard real-time systems, such as aerospace systems, a timing error may result in catastrophic consequences. A major task in real-time system verification is to analyze the timing behavior of the system before deployment in order to guarantee that no timing violation occurs at run time.

A real-time system is typically composed of many tasks which cooperate to provide the required functionality. To verify the satisfaction of the timing requirements of the system, one must first know how long each task (or program) may execute. However,

Author's addresses: M. Lv and N. Guan, Northeastern University, China; email: {lvmingsong, guannan}@ise.neu.edu.cn; W. Yi, Uppsala University, Sweden; email: yi@it.uu.se; J. Reineke and R. Wilhelm, Saarland University, Germany; email: {reineke, wilhelm}@cs.uni-saarland.de.

this is not an easy problem, because the execution time of a program may vary widely as a result of many complex factors, such as data inputs, hardware features, execution contexts, etc. Among all the possible execution times (represented by the yellow range in Fig. 1), the minimum and the maximum are called the Best-Case Execution Time (BCET) and the Worst-Case Execution Time (WCET), respectively. The main objective of program-level timing analysis is to estimate the WCET [Wilhelm et al. 2008], which is then used in system-level timing analysis, such as a schedulability analysis [Davis and Burns 2011].

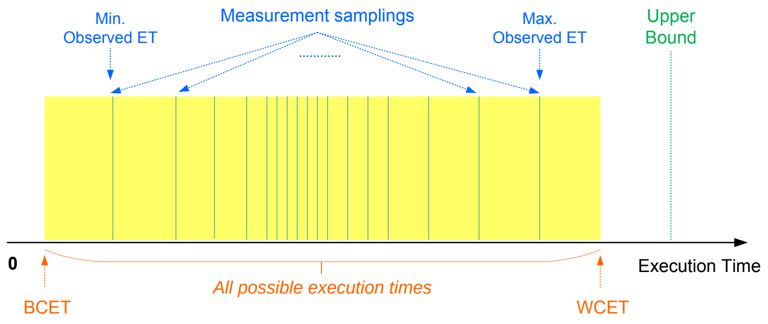


Fig. 1. Distribution of execution times of a program

The common practice in industry has been, and partly still is, to *measure* the end-to-end execution latency of a task [Wenzel et al. 2008], by sampling its executions in different scenarios (depicted as the blue vertical lines in Fig. 1). The maximal observed execution time increased by a safety margin is used as the WCET estimate of the measured program. This approach is called *dynamic timing analysis* in the real-time community. However, the worst case is not guaranteed to be covered by measurements. Thus, the observed WCET is in general an *underestimation* of the actual WCET. Analytical methods that cover all possible execution scenarios (without executing the analyzed program) and provide safe upper bounds on the WCET are desirable for hard real-time systems. They are usually called *static timing analysis*.

Unfortunately, such upper bounds cannot be easily estimated due to both the complexity of the program itself and the uncertainty from the execution environment. The program may execute different control-flow paths depending on input, and these different paths may need different execution times. The execution platform may exhibit a dependence of the execution time of instructions on the execution state of the platform. This *execution state* consists of the occupancy of the platform resources. For example, an instruction may exhibit very different execution times depending on whether instruction or operand fetch hit or miss the cache. The execution environment, finally, may interfere with a program's execution by preempting its execution and thereby increase the program's response time. Hence, these three factors all have impacts on the program's execution time.

Exhaustive exploration of the combined space of control-flow paths and paths through the platform architecture is infeasible due to the size of this space. A safe abstraction of the execution platform is typically used in static timing analysis to increase efficiency. This abstraction may adversely lead to an *overestimated* WCET. Efforts have to be exerted to reduce the overestimation as much as possible, so as to avoid over-provisioning of system resources.

All approaches to static timing analysis compute bounds on the execution times of a program starting with bounds on the execution times of individual instructions occurring at some point in the program. Their execution time typically depends on the execution state of the platform. Depending on this state, an instruction's execution may suffer from *timing accidents*, which may increase the execution time by their associated *timing penalties*. For example, a memory access may suffer from a cache miss, which increases its execution time by the cache miss penalty. The actual execution state is the result of the execution history. Different control-flow paths through the program, in general, result in different execution states and may thus exhibit different execution times. A classification of an occurrence of a memory access as a cache hit or a cache miss must hold for all executions of this memory access.[1]

Static timing analysis methods determine an invariant for each program point that describes all execution states that are possible when control reaches this program point. Such an invariant allows excluding many timing accidents, such as cache misses, pipeline stalls etc., and safely allows to subtract the associated timing penalties from the worst-case upper bound on the execution times.

Of the hardware features to consider in timing analysis, *cache* has the biggest influence on the execution time [Hennessy and Patterson 2011]. A precise analysis of the cache behavior does therefore have a great impact on the precision of any overall WCET estimation.

Cache is a small on-chip memory to bridge the speed gap between the processing unit and the much slower off-chip memory by storing a portion of the content from main memory. If a data request *hits* in the cache, it takes only very few processor cycles to deliver the data from the cache to the processing unit; otherwise, in the case of a *miss*, the CPU has to fetch the data from main memory, which nowadays consumes hundreds of processor cycles.

The role of cache analysis for WCET estimation is to predict the behavior of a program on the platform's caches. For example, cache analysis may provide a safe bound on the number of cache hits or misses when a program executes on some given platform; it may also categorize the accesses to memory blocks in programs as definite hits or misses.

In [Wilhelm et al. 2008], WCET analysis techniques and tools are surveyed. Due to its importance in timing analysis and its complexity, cache analysis alone deserves an in-depth discussion.

The rest of the article is organized as follows. First, background knowledge on WCET estimation and caches are given in Sec. 2. Then, we present the research problems and solutions for the intensively researched LRU caches in Sec. 3. A survey of the most recent research on non-LRU caches are provided in Sec. 4. In Sec. 5, the discussions are extended to cache analysis in multi-tasking and multi-core environments where programs interfere with each other on shared caches. A summary of WCET analysis tools based on static cache analysis is given in Sec. 6. We present an outlook for future research at last.

2. BACKGROUND KNOWLEDGE

We first present an established static WCET analysis framework to exhibit its main work flow and where cache analysis steps in. Then, the basic concepts on cache organization, behavior and analysis are introduced.

2.1. A Classical WCET Analysis Framework

The objective of static timing analysis is to compute safe lower and upper bounds on the execution times of programs. These are also called BCET and WCET estimates,

[1]This distinction between the occurrence of a memory access or an instruction and one, several, or all of their executions is of utmost importance.

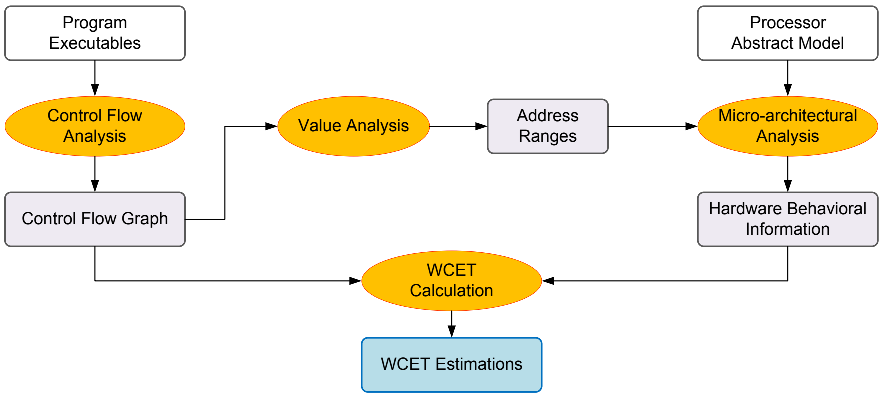


Fig. 2. The separated path and cache analyses framework for WCET estimation

respectively. The WCET is observed in a particular execution scenario with some execution context, such as data input and initial hardware state. Theoretically, the WCET is not computable; otherwise, one could solve the halting problem. In this article, we assume that all real-time programs terminate so that their WCET can be computed.

Most static analyses are performed on the binary code rather than the source code of the program, because the two need not have the same control flow due to compiler optimizations, and the source code does not determine the precise location of instructions and program variables in memory, which are needed for instruction- and data-cache analysis.

A naive, straightforward analysis would enumerate all possible executions to find the largest execution time. However, this method does not scale. Consider a loop with a conditional branch inside. If we do not know whether or not the branch is taken in each iteration, the number of program paths to be explored is exponential with respect to the number of loop iterations. To tackle the complexity, the state-of-the-art analysis techniques adopt the framework in Fig. 2.

The first step is to reconstruct the Control Flow Graph (CFG) of the program. A CFG is a directed graph, with each vertex representing an instruction and each edge representing the control flow. We say there is a *program point* right before each vertex in the following discussions. A CFG typically has a single entry and a single exit corresponding the start and the end of a program. The analysis procedures are then conducted on the CFG.

This step is followed by a *Value Analysis*, which computes enclosing intervals for all potential values of registers and local variables and also determines loop bounds. This is a more or less standard *Interval Analysis* as invented by P. and R. Cousot [Cousot and Cousot 1977]. The next step is to compute an upper bound on the execution time of each instruction (C_i in Equation 1), which heavily depends on the underlying hardware features, such as pipelines [Li et al. 2006], branch predictors [Colin and Puaut 2000; Burguière and Rochange 2005] and caches. *Cache analysis* is an important part of this step, which is often referred to as *micro-architectural analysis* or *low-level analysis*.

With the above results, the final step is to find the execution path that exhibits the longest execution time, typically referred to as WCET calculation. An established approach is the *Implicit Path Enumeration Technique* (IPET) [Li and Malik 1995], whose main idea is to transform the problem of searching the worst-case execution path into searching the execution counts for each instruction such that the execution time is the largest. This can be formally modeled as an integer linear programming

(ILP) problem, in which the execution time of a program is represented by the sum of execution latencies of all instructions. Thus, the WCET can be obtained by maximizing the execution time (the objective function of the ILP problem):

$$WCET = max \sum_{i=1}^{N} C_i \times X_i \tag{1}$$

In Equation (1), N refers to the total number of instructions which is a constant obtained from the CFG; C_i is the WCET for the i^{th} instruction, which has been computed in the second step; the variable X_i stands for the execution count of the i^{th} instruction. X_i is subject to constraints induced by the program structure. In the following, the variable d_{i_j} captures how often the edge from instruction i to instruction j is taken. Then, X_i must be equal to both the total execution counts of all its incoming edges and those of all outgoing edges[2], which can be expressed as follows.

$$\forall i, X_i = \sum_{all\ incoming\ edges} d_{*\_i} = \sum_{all\ outgoing\ edges} d_{i\_*} \tag{2}$$

Other program behavior can be constrained as well. For example, the loop iterations should be bounded in advance either manually or by automatic analysis [Gustafsson et al. 2006]. They can be modeled as linear functions relating the execution counts of the loop body and the loop entry. All available constraints are expressed in one ILP problem, whose maximal solution bounds the WCET from above. To improve analysis efficiency, sequences of instructions (with no branch along the path) are combined into *basic blocks* and represented by a single vertex in the CFG.

The key feature of this framework is the separation of micro-architectural analysis from WCET calculation. In general, this approach is pessimistic. However, the sacrifice of precision is rewarded by significant improvement in analysis efficiency.

2.2. Cache Organization, Behavior and Analysis

— *Cache Organization and Behavior*

A cache is a small, high-speed memory residing on the processor chip (shown in Fig. 3) that stores a copy of a portion of the instructions and/or data in main memory. Each access to the cache results in either a *hit* or a *miss*. One can distinguish two types of cache misses. A *cold miss* occurs when a data element, absent from the cache, is loaded for the first time. If the cache is full and a cache miss occurs, a data element needs to be evicted. A *replacement miss* occurs when an evicted element is reused. Cache hits are the result of *memory reuse*. In most processors, a cache line (the unit for cache access) contains multiple data elements. An access to one element causes the whole cache line to be loaded into the cache. As a result, a following access to another element of the same cache line also results in a cache hit. Besides, consecutive accesses to the same data element result in cache hits as well, an example of which is the execution of a loop. The above two types of reuses are commonly referred to as *spatial reuse* and *temporal reuse* respectively, the pervasiveness of which is expressed by the well-known *locality principle*.

Today, some processors are equipped with two or more levels of caches, as a fine-grained trade-off between cost and speed. The lowest level[3] (namely L1 cache) is usually divided into a private instruction cache and a private data cache, each of which is

[2]For either the entry or the exit instruction, one can simply constraint the execution count to be exactly 1.
[3]A cache level is lower if it is closer to the processing unit; the highest level is typically called the last level.

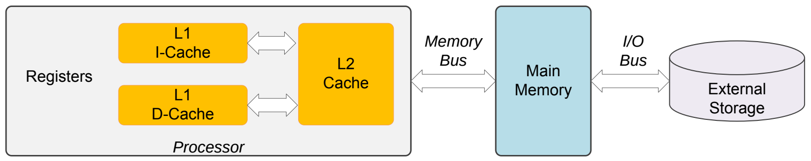


Fig. 3. A common memory architecture

typically no larger than 32 KB and has an access latency of $1-2$ cycles. If a memory access misses in the L1 cache, the L2 cache is queried. The capacity of L2 caches may range from hundreds of KB to several MB, with an access latency of around 10 cycles. In some high performance multi-core processors, an L3 cache may also be deployed to further expand cache capacity. Misses in the last level cache trigger accesses to the main memory via the *off-chip* memory bus, causing a delay in the order of hundreds of cycles.

Like other storage devices, addressing is an important feature of the cache design. Some processors adopt the *set-associative* organization, in which the address space is partitioned into independent *sets*. Every set has a fixed number of *ways*, each of which refers to a single cache line in every cache set. The total number of ways within a cache set is called *associativity*. To load a memory block, the processor first determines which cache set the block maps to. Then a lookup into the target set is performed for a free cache way. If all the cache ways are occupied, the *replacement policy* determines which old block to evict to make room for the new block. In this article, we consider four common policies illustrated in Figure 4, assuming a 4-way cache set.

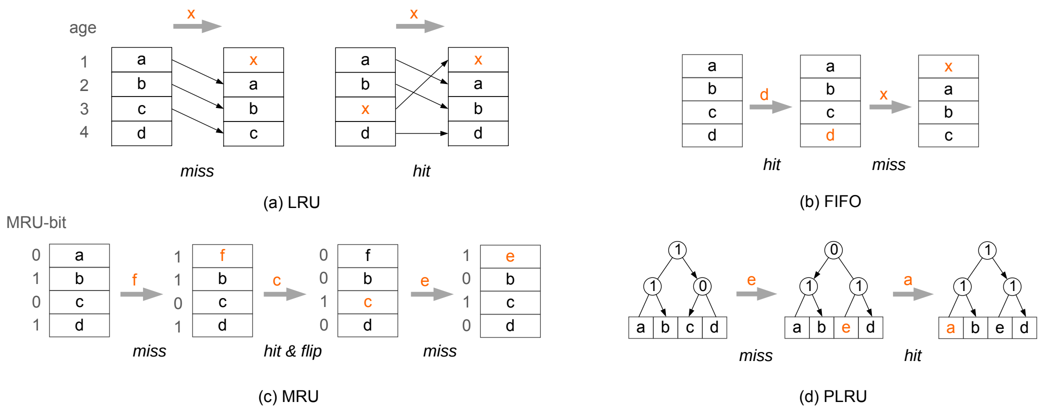


Fig. 4. Common cache replacement policies

The least-recently-used (LRU) policy replaces the block that has been used least recently. The illustration of LRU in Figure 4(a) is a first abstraction from the actual hardware implementation. Each cache way in an LRU cache set is associated with a fixed age, which is received by the block in the corresponding cache way. Figure 4(a) illustrates how the positions of the blocks are reordered upon a cache hit and a cache miss.

However, most commercial processors do not employ LRU, because it requires complex hardware implementation and further leads to higher power consumption. Non-LRU replacement policies, such as First-In-First-Out (FIFO), Most-Recently-Used

(MRU) and Pseudo-LRU (PLRU), are adopted instead since they are simple to implement and still have similar average performance as LRU [Heckmann et al. 2003].

Figure 4(b) shows how the FIFO replacement policy works. A cache hit does not change the cache state. Upon a cache miss, all the memory blocks shift one position downwards, evicting the block in the bottom cache way; then the new block is installed in the top-most cache way. Again, this representation is an abstraction from the actual hardware implementation, which does not shift memory blocks from one cache line to another, but rather maintains a modulo-4 counter to determine the next block to replace.

The MRU cache (shown in Fig. 4(c)) maintains a bit for each cache way (called MRU-bit) to approximate the recency of access. Bit 1 means the block was visited recently. Upon a hit, the MRU-bit of the hit block is set to 1. Upon a miss, the top-most way with MRU-bit 0 is taken by the new block, and its MRU-bit is set to 1. Eventually there is only one cache way with MRU-bit 0. When this way is visited (MRU-bit turned to 1), all the other MRU-bits are set to 0 immediately, called a *global flip*.

PLRU is a tree-based approximation of LRU (Fig. 4(d)). It arranges the cache ways at the leaves of a binary tree with $k-1$ bits, where k is the cache associativity. Bit 0 and 1 on the branches indicate the left and the right subtrees respectively. Following the bits downwards from the root, the cache line to be replaced or refilled can be found. After an access (either hit or miss) to a cache way, all tree bits along the path from it to the root are set to point away. It is possible that a cache set contains invalid cache lines. We assume the *tree-fill* policy, by which the line to be filled or replaced is always determined by the tree bits.

In most architectures, cache sets are completely independent of each other. This makes the independent analysis of programs' behavior on different cache sets possible. Throughout this article, we focus on the cache behavior in one set, and may use *cache* to refer to a cache set for simple presentation when appropriate.

— *Cache Analysis*

The objective of cache analysis is to statically determine the cache behavior of a program, the results of which can be used for performance analysis and optimization. The results may be of several types: one is the *classification of individual memory accesses in a program as hits or misses*. Such a classification of memory accesses can be used in a cooperating pipeline analysis to determine whether the pipeline may have to stall on an instruction or operand fetch. Another is a *bound on the number of cache loads in a segment of the program*. This allows, under certain conditions, to just add an accumulated penalty to the execution-time bound for memory accesses that could be neither classified as cache hits or misses. The former type of cache analysis could be called a *classifying cache analysis*, one instance of the latter, relevant for practice, is known as *persistence analysis*.

A typical use of the results of a cache analysis in real-time systems is for computing the BCET and WCET of programs. The bigger the percentage of hits that will happen during execution it can predict, the tighter the WCET estimation is. In contrast, predicting a higher percentage of actual misses leads to tighter BCET estimation.

The designer of a cache analysis faces several questions: the first one is whether the analysis is to be a classifying or a persistence analysis. The second question is by which method cache behavior should be analyzed. Associated to the second question are the questions of the granularity of the analysis and the representation of the cache-behavior properties.

Let us illustrate this rather abstract discussion at the example of classifying cache analyses. The most precise analysis would predict each *executed memory access* to be either a hit or a miss—here it is vital to make the difference between an *executed* *memory access* and the *occurrence of an instruction involving a memory access* in a program. We have assumed that all real-time programs terminate. Hence any program would execute only finitely many memory accesses, so that such an analysis would in principle be possible. However, the corresponding analysis would, in general, not scale. On the other end are cache analyses that would classify *occurrences of memory accesses* as *always hit* or *always miss*, where *always* means for all executions of this occurrence of the memory access. However, experience has shown that the actual execution times of memory accesses associated with one occurrence of a memory access may vary widely. This means that just taking their upper bounds may largely overestimate the memory-access costs. Precise and efficient analyses should attempt to classify subsets of executed memory accesses corresponding to one occurrence, such that the accesses in the subsets have some homogeneous timing behavior. The subsets would be characterized through control-flow criteria, in the following called *contexts*. The most important examples for contexts are different iterations of loops.

To approach the second question raised above, one needs to identify the information about cache contents—in the following mostly called *concrete cache states*—to be computed by a classifying cache analysis. This information would provide answers to the question, *are all memory accesses belonging to this occurrence (in this context) hits or are they all misses?* One solution would be to collect at each program point the set S of all concrete cache states that are possible when program control reaches this program point (in this context). Such an analysis would again not scale. Instead, one can represent sets of concrete cache states by *abstract cache states*. Each abstract cache state (compactly) represents a set of concrete cache states. As we will later see, two types of such abstract cache states are of interest. Consider the set S of all concrete cache states that are possible at a program point. One might represent by an *abstract Must cache state* the information, whose memory blocks will be in each of the possible concrete cache states in S. This is obtained by some kind of intersection applied to the elements in S. Likewise, one might represent in an *abstract may cache state* the information, which memory blocks may be in one of the possible cache contents. This is obtained by some kind of union applied to the elements in S.

3. ANALYSIS OF LRU CACHES

For decades, a majority of research on cache analysis has focused on caches with LRU replacement strategy. In this section, we survey the main analysis techniques with an emphasis on the approach based on Abstract Interpretation (AI) [Ferdinand and Wilhelm 1999]. This technique is realized in the aiT tool of AbsInt [Heckmann and Ferdinand 2014], which is widely used in industry. Since programs spend most of their execution time in loops, a sub-section is dedicated to the analysis of the cache behavior in loops. The big picture on LRU caches is completed with further discussions on data cache and multi-level cache analyses.

3.1. Abstract-Interpretation-Based Approaches

The first cache analyses based on abstract interpretation (AI) were proposed by Ferdinand and Wilhelm in the 1990's [Alt et al. 1996; Ferdinand and Wilhelm 1999].

The overall approach works in two phases:

(1) An AI-based cache analysis computes abstract cache states at all program points as part of a fixed-point solution;
(2) These abstract cache states are queried in order to classify memory accesses.

A Short Introduction to Abstract Interpretation. Abstract interpretation [Cousot and Cousot 1977] is a static program analysis method based on a semantics of the considered programming language. Instead of executing the program on the concrete domain of values, it executes an abstracted version of the program on an abstract domain of descriptions of values. In the case of cache analysis, the program abstraction only describes the memory-access behavior of the program, i.e., it performs all memory accesses that the program would execute. This abstracted program works on *abstract cache states*, which are descriptions of sets of concrete cache states. One abstract cache state is associated with each program point. Whenever the analysis encounters a memory access, it updates the abstract cache state in a way induced by the update that the processor would perform on the concrete cache states. Whenever the control flow of the program merges, e.g. at the end of a conditional or at the header of a loop, it combines the incoming abstract cache states in a sound way.

Gary Kildall [1973] has recognized that the abstract domains of typical data-flow analyses were lattices, i.e. partially ordered sets where all subsets have least upper bounds. The partial order reflects the relative information content of two lattice elements. By convention, elements lower in the lattice represent more information than information higher in the lattice, i.e., an element a below or equal to an element b in the lattice, $a \sqsubseteq b$, contains no worse information than b. The domain of abstract cache states together with a partial order reflecting the amount of knowledge about cache contents, in this sense, also forms a lattice.

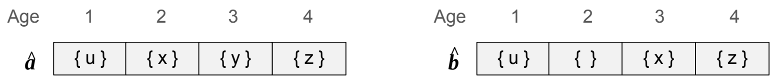


Fig. 5. An example to show the $\sqsubseteq$ relation w.r.t. the Must domain

Consider two abstract Must cache states $\hat{a}$ and $\hat{b}$ (as shown in Fig. 5). Abstract cache state $\hat{a}$ represents just one concrete cache state, containing memory blocks $\{u, x, y, z\}$ with the ages 1, 2, 3, 4. Abstract cache state $\hat{b}$ represents concrete cache states with memory block u having age 1, x having age at most 3, z having age at most 4, and possibly one more (unknown) block at age 2, 3, or 4. $\hat{a} \sqsubseteq \hat{b}$ means that all the cache states represented by $\hat{a}$ are also represented by $\hat{b}$. In particular, this implies that all the memory blocks known to be contained in the concrete cache states described by $\hat{b}$, in the example above u, x, z, are also known to be contained in the concrete cache states described by $\hat{a}$. Furthermore, $\hat{a}$ additionally tells us that, (1) block y is guaranteed to be in the cache while $\hat{b}$ does not; (2) the age upper bound estimated for block x in $\hat{a}$ is smaller that that in $\hat{b}$. Clearly, the abstract cache state $\hat{a}$ contains better information than $\hat{b}$. As stated above, at control-flow merge points cache analysis must combine the incoming information in a sound way. The operation applied to the incoming abstract cache states is the least upper bound, $\sqcup$, of the lattice. This is shown in Fig. 6. As said above and made more precise later, it is some form of intersection. It determines safe information holding for all incoming paths.

The lowest element in the lattice of abstract cache states, $\perp$, called *bottom*, describes the empty set of concrete cache states. It is the initial analysis information at all program points but program entry. If we do not have any information about the cache contents at program entry, the highest lattice element, $\top$, called *top*, is used as initial analysis information. It describes the set of all concrete cache states, and thus the absence of information about cache contents.

The update functions are, in general, monotone, so that information, once computed, is not lost again. A fixed-point iteration over the control-flow graph of the program is guaranteed to terminate and deliver the least fixed point as solution. Essential for termination is the finiteness of the lattice.

Let us summarize this short introduction to AI by listing the main ingredients of a particular abstract interpretation. The designer needs to choose or define an *abstract domain*, a lattice of abstract values, which are descriptions of (sets of) concrete values. The *partial order* defines the relative information content of two lattice elements. The *least upper bound* is the operation to join abstract values flowing to a program node through different control-flow graph edges. *Abstract update functions* for an instruction reflect the instruction's effect on the incoming abstract values.

Querying the Results. Querying an abstract cache state, resulting from a cache analysis, for an accessed memory block may yield qualitative properties as listed in Table I. To determine whether the memory access to m is *always hit* (AH), one simply checks the existence of m in the abstract must cache reaching its program point. Similarly, to determine whether the memory access to m is *always miss* (AM), it suffices to know that m is not in the abstract may cache reaching its program point. If a memory access can be neither classified as AH nor as AM, it is classified as NC. An NC classification can have two reasons: (1) some of the executions of a memory access hit in the cache and others miss in the cache, or (2) the analysis method overapproximates the set of concrete cache states and thereby fails to deliver the correct classification. Research results show that these properties are able to cover most access behaviors for LRU caches [Ferdinand and Wilhelm 1999]. The classifications can then be expressed as linear constraints on the execution cost of each instruction (basically each instruction generates a single memory access) and later integrated into WCET computation. In architectures without timing anomalies [Reineke et al. 2006], if the classification of a memory access is NC, it is safe to treat it as AM. The properties AH and AM in Table I are explored by independent analyses, which are now described in detail.

Table I. Cache Hit/Miss Classification

Classification	Cache Access Behaviors Described	Analysis
Always Hit (AH)	Block is guaranteed to be in the cache upon each memory access	Must
Always Miss (AM)	Block is guaranteed not to be in the cache upon each memory access	May
Not Classified (NC)	Cannot be classified by any of the above classifications	/

Must Cache Analysis. The objective of Must analysis is to compute a Must-ACS at each program point, *which represents the common cache contents in all possible executions leading to this program point*. An age is associated with each memory block in the Must-ACS, which is an upper bound of its ages in all CCS. We use a graphical representation to show a Must-ACS, in which blocks are grouped according to their ages, e.g., in Fig. 6. The set of CCS represented by a given Must-ACS is formally defined by the *concretization function* below, where c and $\hat{c}$ denote concrete and abstract cache state respectively, and $age(c, m)$ refers to m's age in cache state c (applies to both concrete and abstract states).

$$conc^{Must}(\hat{c}) = \{c \mid \forall m \in \hat{c} \;:\; m \in c \;\wedge\; age(c, m) \leq age(\hat{c}, m)\} \tag{3}$$

The Must-ACS at the program entry is initialized with $\top$ representing all concrete cache states if the initial cache content is unknown; all other nodes are initialized with $\bot$ representing the empty set of concrete cache states. A fixed-point computation is employed to compute the Must-ACS at each program point, during which two main operations over the Must-ACS are involved.

A *join function* combines several Must-ACS into a single Must-ACS when the control flow merges. The resulting Must-ACS takes the intersection of the sets of blocks in all the incoming states and assigns to each block its maximal age from the incoming

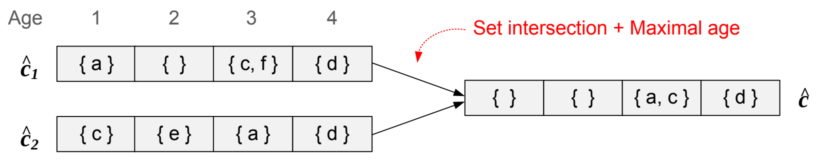


Fig. 6. An example to demonstrate the Must join function

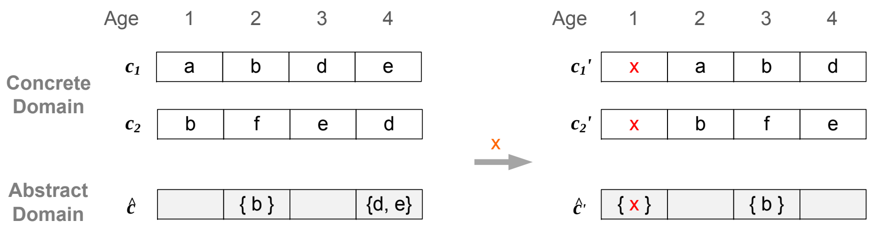


Fig. 7. An example to demonstrate the Must update function

states, as shown in Fig.6. An *update function* $\hat{U}(\hat{c}, x)$ defines how an abstract state $\hat{c}$ is changed due to an access to memory block x, specifically, how the age of each block in the ACS is updated. Fig. 7 shows an example. A correct Must update function guarantees that the age of each block in the computed ACS is a safe age upper bound for all possible represented CCS. For example in Fig. 7, the age of d in $\hat{c}$ implies that there could be a CCS represented by $\hat{c}$, in which d has an age of 4, such as c_2. By loading x, we can no longer guarantee d still stays in any resulting CCS. Thus, d has to be removed from the computed abstract state $\hat{c}'$.

May analysis. May analysis computes a May-ACS at each program point, which represents all potentially cached contents in all possible executions leading to this program point. If block m does not exist in the May-ACS at the reaching program point, we can guarantee the access to m is AM. Unlike the Must-ACS, the age of each block in a May-ACS is the lower bound of its ages in all represented CCS, as expressed by the May concretization function below, with the same notions as in function 3.

$$conc^{May}(\hat{c}) = \{c \mid \forall m \in c \ : \ m \in \hat{c} \ \wedge \ age(\hat{c}, m) \leq age(c, m)\} \tag{4}$$

The May join function takes the union of the sets of blocks in all incoming May-ACS and assigns each block the minimal age in all incoming states. The May update function is exemplified in Fig. 8, where $\hat{c}$ is the May-ACS representing the concrete states c_1 and c_2. Take memory block d for example. d's age in the May-ACS is the minimum of those in c_1 and c_2. After the access to x, d is evicted from c_2. However, d still remains in the resulting May-ACS $\hat{c}'$, because $\hat{c}'$ must soundly represent the other resulting CCS c_1' in which d remains. To determine whether the memory access to m is AM, it suffices to know that m is not in the May-ACS reaching its program point. A block m is not in the final May-ACS at a program point because either m has never been loaded or enough different blocks have been loaded to evict m from the cache. May analysis does not directly help with tighter WCET estimations, however, predicting more misses results in better estimations on BCET.

3.2. Improving Precision by Using Contexts

In practice, merely relying on classifying analyses, such as Must and May analyses, may still largely overestimate the memory-access cost and thus the WCET. Methods to improve the knowledge about the cache behavior are proposed by taking program

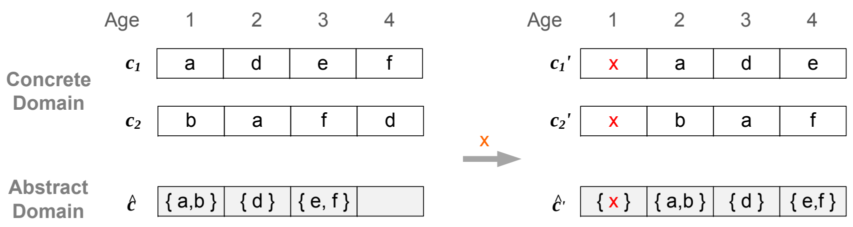
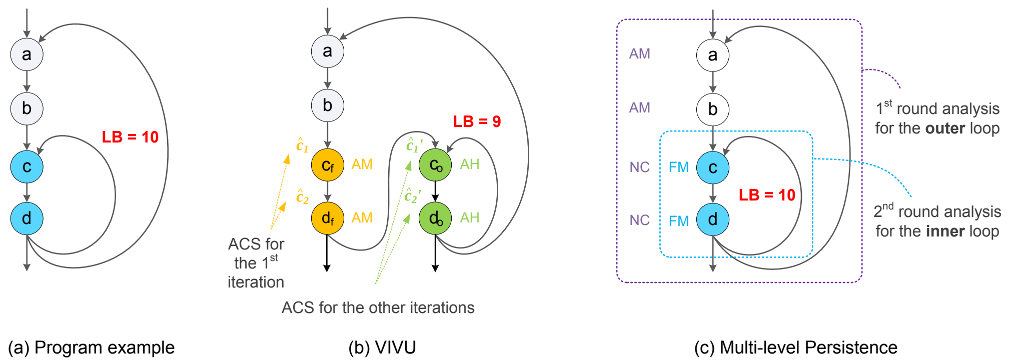


Fig. 8. An example to demonstrate the May update function

structures into consideration, in paticular loops. The cache behavior of programs in loops is somewhat special: the first iteration typically loads the contents into the cache; later iterations profit from the first iteration since accesses to the cached contents are hits. The concrete state on return to the loop header may thus be very different from that reaching the loop from the outside. This is also reflected in cache analysis where the abstract state upon return from the first iteration may be very different from that on the entrance to the loop. Naively applying the join function to these two abstract cache states would produce very bad information about the cache behavior of the loop. In such cases, a large percentage of the accesses to the memory blocks in the loop cannot be classified as either AH or AM by the previously introduced Must and May analysis. However, there are two alternative ways to tighten the WCET estimation. The first one exploits the above observation by virtually unrolling each loop followed by a Must analysis. An access to a memory block who in the first iteration would be classified as AM, and who in the other iterations would be classified as AH would then be classified as FM (first miss). The other alternative would be to bound the number of cache misses for all the accesses to a memory block within a certain program scope. These two analysis techniques will be now introduced. Note that the first analysis still is a classifying analysis for memory accesses, albeit with a new classification, FM, and the second is a bounding analysis for memory blocks in a scope.



a
b
c
d
LB = 10
(a) Program example
a
b
$\hat{c}_1$
$\hat{c}_2$
c_f AM
d_f AM
ACS for the 1st iteration
$\hat{c}_1'$
$\hat{c}_2'$
c_o AH
d_o AH
LB = 9
ACS for the other iterations
(b) VIVU
AM
AM
NC
NC
FM
FM
a
b
c
d
LB = 10
1st round analysis for the outer loop
2nd round analysis for the inner loop
(c) Multi-level Persistence

Fig. 9. The ideas of VIVU and multi-level Persistence analysis

Virtual Inlining & Virtual Unrolling (VIVU). VIVU [Martin et al. 1998] can be used to improve cache analysis precision for loops. The idea is to analyze the first loop iteration separately from all other loop iterations. This is done by virtually unrolling the first iteration of the loop body[4], so as to distinguish the behavior between the two contexts. Then, a Must analysis is applied to the program with the unrolled loop to find the AH memory accesses in all but the first loop iterations. Fig. 9(b) shows the results of unrolling the inner loop of the program in Fig. 9(a). In the new CFG, c_f and c_o refer to the first and the other iterations of the inner loop respectively (similar for d). Assume the cache is 2-way associative. Must analysis on the new CFG is able to classify c_o and d_o as AH, and thus c and d as FM.

Persistence Analysis. One can aim at the same analysis objective by a Persistence analysis. There are several possible notions of persistence of memory blocks one could aim at. These are:

persistence. execution causes at most one miss for the memory block,
first miss. only the first access is a miss, all others are hits,
no eviction. the block is never evicted after a possible miss.

For a memory block that is persistent in a program fragment, timing analysis can assume a bound of one cache load for all accesses within that program fragment.

The first Persistence analysis was proposed by Ferdinand and Wilhelm [Ferdinand and Wilhelm 1999]. This analysis employs abstract cache states, Per-ACS, as do the Must and May analyses. The fact that a block has been visited and already evicted from the cache is modeled by assigning an age $\top$ to the block, where $\top$ is larger than the cache associativity. The Per-ACS at each program point represents the cache contents that are potentially visited and then guaranteed to remain in the cache. If a memory block exists in the Per-ACS at the end of the scope, then one can guarantee that at most one cache miss may occur for all the accesses to this block.

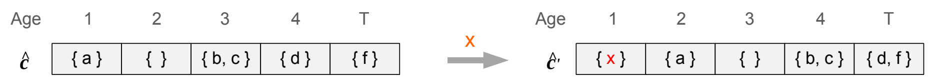
The concretization of a Per-ACS is a set of traces satisfying the persistence condition, i.e. at most one miss for each block with a non–$\top$ age in the Per-ACS. More precisely, a Per-ACS captures upper bounds on the ages of memory blocks, assuming that they have already been accessed at least once in the execution of the program.

Fig. 10(a) gives an example for Persistence update. All blocks in Per-ACS $\hat{c}$ are potentially visited during program execution. By accessing x, d is no longer guaranteed to be in the cache since it has age 4 in $\hat{c}$, which is maintained by putting d in the $\top$-age line. Block f, already evicted from the cache before accessing x, remains unchanged in the $\top$-age line. The update function for Persistence analysis mainly needs to guarantee the maximal age for each block in the ACS is soundly maintained.

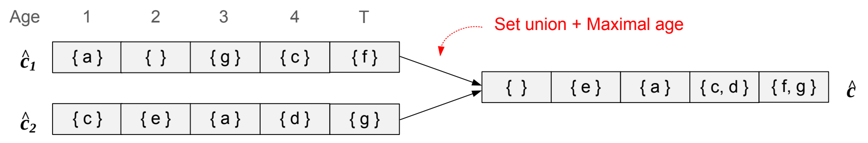
At a control flow merge point, several Per-ACS are merged by the join function, which takes the *union* of the blocks in all the incoming Per-ACS and assigns each block the *maximal* age from the incoming states, as shown in Fig. 10(b). Intuitively, the set union operation guarantees the resulting Per-ACS does not lose track of any potentially visited memory block; the maximal age ensures that we can safely predict whether a block is definitely persistent after its first access.

The Persistence analysis by [Ferdinand and Wilhelm 1999] was recently found to be unsafe, due to an error in the update function, which may incorrectly underestimate the age of a block. The error was corrected by Cullmann [Cullmann 2013] and Huynh [Huynh et al. 2011]. Several different ways were proposed to restrict the set of memory blocks in a Per-ACS to the actual capacity of a cache set. The simplest

[4](1) The unrolling is called *virtual* since it is done by maintaining separate abstract cache states for the first iteration and the remaining iterations (e.g., $\hat{c}_1$ and $\hat{c}'_1$ in Fig. 9(b)). For ease of understanding, we use a physically unrolled CFG to show the effects. (2) VIVU allows to unroll more than one iteration of the loop since iterations other than the first may have vastly different behavior and thus execution times. Here we assume only unrolling the first iteration to simplify presentation.



(a) Persistence Update

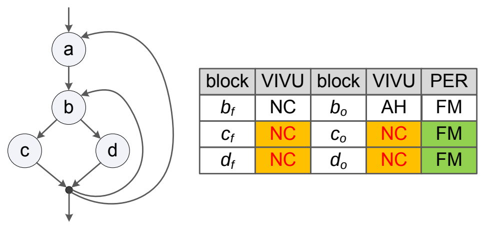


(b) Persistence Join

Fig. 10. Operations for Persistence analysis

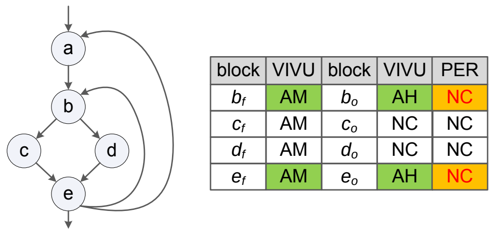
way, yielding the least precise results, marks all memory blocks in a Per-ACS as non-persistent, if the Per-ACS contains more blocks than the associativity allows. Others check the number of conflicts between members of the Per-ACS; Huynh's analysis employs fixed-point computation to collect for each block m a set of potentially conflicting blocks (blocks that are mapped to the same cache set with m and thus may age m). If the total number of conflicting blocks is no larger than $A - 1$, where A is the cache associativity, then one can safely draw the conclusion that m, once loaded, will persist in the cache. A similar idea was also applied in an earlier Persistence analysis [Mueller 2000]. Cullmann presents a number of similar persistence analyses [Cullmann 2013]. The most precise of Cullmann's analyses relies on a May analysis to make correct decisions on age update.

Analysis Scope. A bounding analysis, such as the Persistence analysis, is designed to investigate cache behavior within a *program scope*, in most cases a loop body. It is common for a program to have nested loops, where a block in an inner loop also belongs to the outer loop. A natural question would be: does the block have a different cache behavior for different loop levels? To distinguish a block's behavior, the relevant loop nest(s) (the inner loop, the outer loop, or both) is/are unrolled in the VIVU approach. For Persistence analysis, a multi-level approach was proposed by Ballabriga and Cassé [Ballabriga and Casse 2008], which applies the basic Persistence analysis on the relevant scope, here specifically the relevant loop nest, to explore local cache behavior. Fig. 9(c) illustrates the basic idea. Persistence properties regarding different loop nests for a memory block can be encoded as linear constraints (or other forms) and integrated into WCET computation for tighter estimations.



block	VIVU	block	VIVU	PER
b_f	NC	b_o	AH	FM
c_f	NC	c_o	NC	FM
d_f	NC	d_o	NC	FM

(a) Multi-level Persistence outperforms VIVU



block	VIVU	block	VIVU	PER
b_f	AM	b_o	AH	NC
c_f	AM	c_o	NC	NC
d_f	AM	d_o	NC	NC
e_f	AM	e_o	AH	NC

(b) VIVU outperforms multi-level Persistence

Fig. 11. Comparing VIVU with multi-level Persistence analysis

Comparing VIVU and Persistence Analysis. Since both VIVU and Persistence analysis are able to bound cache misses for a program scope, a straightforward question would be: which one is more precise? In fact, the two techniques are generally incomparable. For the program in Fig. 11(a), c_o and d_o cannot be classified as AH by Must analysis [Ferdinand and Wilhelm 1999] of the VIVU approach, as neither c_o nor d_o are guaranteed to be accessed in the first loop iteration. On the other hand, the multi-level Persistence analysis is successful in this example for c and d. In Fig.11(b), both b_o and e_o can be classified as AH by VIVU. However, none of b, c, d or e can be classified as FM by the multi-level Persistence analysis [Ballabriga and Casse 2008], which is based on a specific Persistence analysis proposed by [Mueller 2000]. This is because the Persistence analysis [Mueller 2000] counts the conflicting blocks in a loop (mapped to the same cache set); if the number is larger than the cache associativity, none of the blocks in the loop can be classified as FM. If, otherwise, a different Persistence domain is adopted in the multi-level analysis, such as [Cullmann 2013], b and e can be locally classified as persistent.

VIVU and Persistence analysis can also be compared in terms of analysis cost. On the one hand, VIVU may result in a more expensive micro-architectural analysis, having to distinguish multiple contexts. On the other hand, the results of Persistence analysis need to be encoded into constraints during implicit path enumeration. The influence of the two effects on analysis times have not yet been compared empirically.

3.3. Other Techniques

Mueller and Whalley proposed *static cache simulation* [Mueller and Whalley 1995]. Static cache simulation is very similar to AI-based methods discussed above: it adopts the concept of abstract cache states and leverages a data flow analysis by fixed-point iteration to compute them. Static cache simulation also tries to determine similar classifications to Ferdinand's analysis. The major difference lies in the construction of their abstract domains: the analysis employs a unified abstract domain to infer AH, AM, FM and First Hit (FH) classifications all at once.

Before AI-based approaches, Li et al. presented a technique which uses *Cache State Transition Graphs* (CSTGs) to model cache behavior [Li et al. 1996]. An CSTG, built out of the CFG, models the cache-state transitions for a given cache set. A vertex in the CSTG stands for a possible concrete cache state, and each edge in the CSTG represents a possible transition from the source state to the destination state due to a memory access in the program. Instead of exploring qualitative properties, such as AH, AM and FM, the analysis tries to find a lower bound on cache hits for each memory block. The bounds can be modeled as linear constraints and combined into the ILP to obtain the WCET for the program. By explicitly enumerating the concrete cache states, the CSTG approach can provide good analysis precision. However, it does not scale with program size. Assume that there are M memory blocks mapped to each cache set with associativity K, the number of states in an CSTG can be calculated by $\sum_{i=0}^{K} \frac{M!}{(M-i)!}$ [Li et al. 1996]. Note that the number of linear constraints is of the same scale as the number of CSTG states. In practice, the analysis efficiency is low due to the complexity of the resulting ILP problem.

Model checking [Clarke et al. 1999] is a powerful technique widely used in system-level timing analysis of real-time systems. Timed automata [Alur and Dill 1994] have been used to model the cache behavior of programs, and a model checker has been employed to find the WCET. Existing work includes the McAiT tool [Lv et al. 2011], the METAMOC approach [Dalsgaard et al. 2010a], Gustavsson et al.'s analysis [Gustavsson et al. 2010], etc.

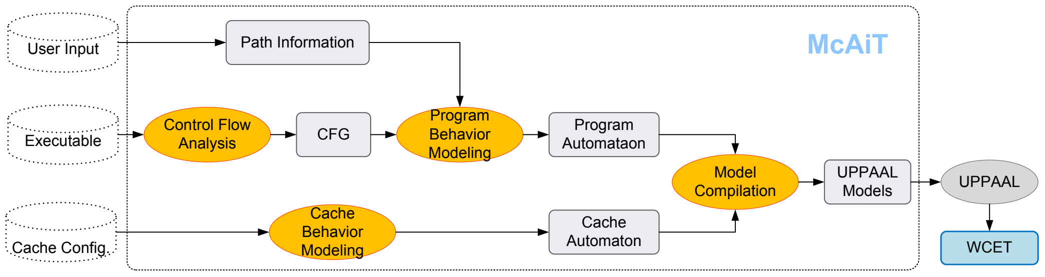


Fig. 12. The analysis framework of McAiT

Fig. 12 shows the architecture of the McAiT tool. McAiT first constructs the program automaton out of the CFG, which fully simulates the behavior of the program, such as the control flow and how the program accesses caches. For a given cache configuration, McAiT builds a timed automaton to model each cache. The execution of an instruction causes the program automaton to issue messages to the cache automaton via UPPAAL's channel mechanism, and the cache automaton updates the cache state accordingly. The timed automata models for both the program and the cache are then explored using the UPPAAL model checker to find the WCET.

Essentially, the estimated WCET by a model checker is the actual WCET of the program, since all the possible executions are explored by the model checker. Cache hits and misses for *each execution of memory accesses* is precisely reported. The major difference between this approach and the CSTG approach is that the possible cache states are not explicitly modeled in the automaton, but rather explored by the model checker. The main drawback of model checking-based approaches is their lack of scalability, since an exponential state space has to be explored.

3.4. Data Caches

Modern processors are typically equipped with *data caches* to improve the performance of data accesses. Instructions are fetched from known addresses; so instruction fetching can be accurately analyzed. In contrast, data accesses are less predictable [Vera et al. 2003; Lundqvist and Stenström 1999a].

— *Main Challenges*

Before predicting hits/misses for data accesses, the set of data addresses accessed by each instruction needs to be determined, referred to as value analysis or address analysis [White et al. 1999; Balakrishnan and Reps 2004]. The threat to precision is that an imprecise value analysis may not be able to eliminate memory accesses that do not occur in a real execution. The problems are the following: Firstly, data manipulations using redirectable pointers make it hard to statically determine the data items actually accessed. Secondly, in the presence of dynamic data structures on the heap, the data addresses can only be determined at run-time (due to this problem, dynamic data structures are typically avoided in hard real-time systems). Lastly, value analyses may work with abstractions of memory addresses, such as intervals. As a result, the address range they compute may be over-approximated. Considering non-feasible data accesses in cache-behavior analysis increases the probability of not being able to classify memory accesses as cache hits or misses. In addition, a memory access without precisely determined address pollutes the information contained in an abstract data cache since the update function has to be applied to all potential concrete addresses.

Besides that, data cache analysis is challenged by another problem: executing an instruction may generate accesses to *multiple* data addresses. Fig. 13 depicts a program

```
1 for (i = 0; i < N; i++)
2   for (k = 0; k < N; k++)
3     for (j = 0; j < N; j++)
4       C[j][i] += A[k][i] * B[j][k];
```

Fig. 13. An example of matrix multiplication

with a matrix multiplication, in which line 4 generates accesses to different matrix elements (in different loop iterations). For this simple program, one can easily determine the data items accessed in each loop iteration. But, in general, data accesses could be very unpredictable due to input dependence of the array indexes. Think of accesses to array $A[x][y]$: if the values of x or y are not clear, one has to conservatively assume that any address in the whole array could be accessed. Furthermore, classifications of memory accesses as used for instruction-cache analysis (AH, AM, FM) may not be sufficient to describe data cache behavior.

—*Analysis without Input Dependence*

Early work, such as the Cache Miss Equation (CME) framework [Ghosh et al. 1999], focused on analyzing programs with predictable data accesses. The underlying idea is to set up mathematical formulas (Linear Diophantine equations specifically) to precisely capture both spatial and temporal memory reuses by relating data addresses, loop induction variables and cache parameters. From the solution of the equations, one can check if a memory block is evicted from the cache before it can be reused. An upper bound on the number of misses can thus be obtained for WCET estimation.

However, only a small set of programs can be analyzed by the CME framework: (1) loops must be rectangular loops and perfectly nested; (2) array subscript expressions and the bounds of the loop index must be affine combinations of the enclosing loop indices; (3) no data/input-dependent conditions may exist. The CME framework has been later extended to allow function calls [Vera and Xue 2002], conditionals only depending on the loop induction variables [Vera and Xue 2002], and multiple loop nests [Ramaprasad and Mueller 2005]. Unfortunately, none of these methods can deal with input dependence. Clauss presented an approach of solving cache miss equations through the mapping to Ehrhart polynomials [Clauss 1996]. Still, the complexity of solving these polynomials is high. Another approach is the Presburger Arithmetic framework [Chatterjee et al. 2001], which has similar restrictions and is computationally expensive.

—*Analysis with Input Dependence*

Earlier research to handle input dependence focused on direct-mapped caches. Kim et al. proposed an analysis method based on the pigeonhole principle [Kim et al. 1996]. Fig. 14 shows 3 iterations in the execution of a loop in which a, b, c, and d can be accessed. If in total 9 memory accesses are generated, then at least $9-4=5$ among them must be cache hits. Apparently, the 4 cache misses are simply cold misses. This work was later extended by Staschulat and Ernst to handle programs with unpredictable input dependency [Staschulat and Ernst 2006], in which cache misses are bounded according to data access types: (1) cache misses from predictable accesses are bounded by the pigeonhole principle; (2) cache misses from unpredictable accesses are tracked down by a miss counter and expressed with linear constraints. Unfortunately, these methods are still too restrictive. First, they only work for direct-mapped caches. Second, the loops must fit into the cache to utilize the pigeonhole principle. Essentially, these approaches correspond to simple Persistence analyses for the special case where programs fits into the cache.

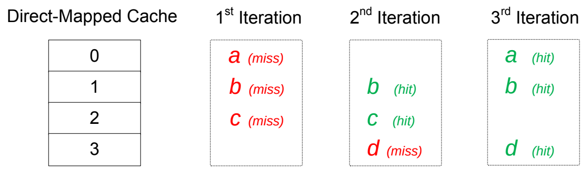
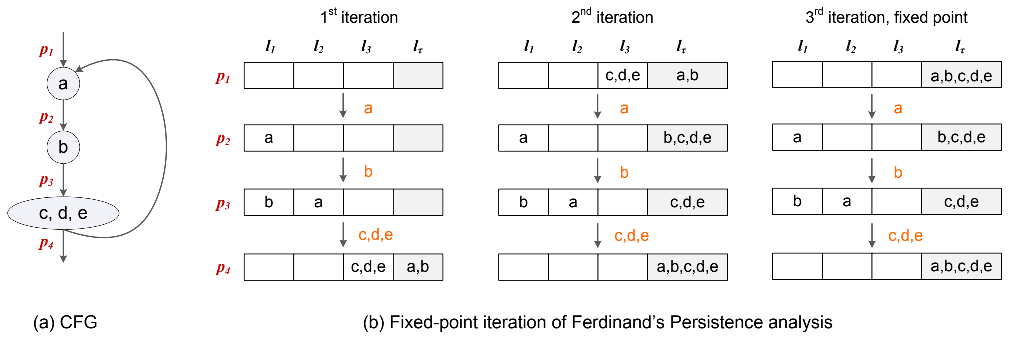


Fig. 14. Data cache analysis based on the pigeonhole principle

AI-based analysis techniques are extended to analyze set-associative data caches with input dependency. Ferdinand extended the Persistence analysis [Ferdinand 1997] with a new update function to handle multiple memory accesses by one instruction. The basic idea and its drawback can be explained by the example in Fig. 15.



p1
a
p2
b
p3
c, d, e
p4
(a) CFG
1st iteration
2nd iteration
3rd iteration, fixed point
l1
l2
l3
lτ
c,d,e
a,b
a,b,c,d,e
b,c,d,e
(b) Fixed-point iteration of Ferdinand's Persistence analysis

Fig. 15. Ferdinand's Persistence analysis for data caches

Fig. 15(a) gives the CFG of a loop in which p_1 to p_4 are program points. Note that each time the instruction after p_3 is executed, one of the blocks from $\{c, d, e\}$ could be accessed. Fig. 15(b) shows the fixed-point iteration process, given a cache size of 3. The last line l_τ of each abstract state is used to collect the blocks that have been evicted from the cache (a common structure for most Persistence abstract domains). On the transitions from p_3 to p_4, since it is not clear which of the three blocks (or their combination) is actually accessed, they pessimistically assume that all blocks in $\{c, d, e\}$ could be accessed and cause other blocks to age. Thus, both a and b are evicted from the cache (collected in l_τ). Moreover, since there is no knowledge about the access sequence of c, d and e, they receive an age of 3 when they are brought into the cache state in the 1^{st} iteration. As a result, no cache hit can be predicted for this loop. As only one of the blocks c, d, e may be accessed in every iteration of the loop, slightly better results would be possible with a different transfer function.

Sen and Srikant developed a Must analysis for data caches [Sen and Srikant 2007]. The analysis can be combined with VIVU to discover the persistence property of data accesses. Despite some small differences, the age manipulation in Sen's Must analysis are similar to Ferdinand's Persistence analysis [Ferdinand 1997], and thus may lead to very pessimistic estimations.

Ferdinand's and Sen's analyses show that without modeling data access patterns, the abstract domain has to do very conservative age maintenance. Again, for the program in Fig. 15(a), if by some means we know that the lifetime of c, d, and e do not overlap, then the analysis can be improved. For example, if the loop iterates for 30 times, c is only accessed in iterations 1 to 10, d only in iterations 11 to 20, and e only in iterations 21 to 30, then c, d and e cannot evict each other in their lifetime. They are actually *persistent* (given a cache size of 3) once they are loaded into the cache, since any of them can only be aged by a and b. Based on this observation, Huynh et al. proposed scope-aware data cache analysis [Huynh et al. 2011]. Each memory access is now associated with a *temporal scope* to model its lifetime, as an augmentation to the traditional AI-based analysis. In the update function, memory accesses that have no overlapping temporal scopes do not cause each other to age. In consequence, some non-existing access conflicts are excluded, and more persistent data accesses can be identified (such as c, d and e).

Hahn and Grund observe that cache analysis does not require knowledge of *absolute addresses* of memory accesses. Instead it is sufficient to know about the *relation* between the addresses of different memory accesses: do they refer to the same cache block, a different cache block but the same cache set, or different cache blocks in different cache sets? Based on this insight they developed *relational cache analysis* [Hahn and Grund 2012], which can classify accesses as cache hits even if the absolute address of the access is unknown.

To summarize, input dependence makes data cache analysis a challenging task. The pessimism mainly comes from competition of memory accesses in the analysis that are infeasible in real executions. The main causes of this problem are imprecise address analysis and the inability to model and analyye data access patterns. The success of data cache analysis depends on whether *temporal and spatial locality* of data accesses can be precisely captured and analyzed.

3.5. Multi-Level Caches

Most modern processors adopt a multi-level cache design. Upon a memory access, the processor queries the memory hierarchy from the L1 cache down to main memory until the requested data or instruction are found. Regardless of the number of levels, the highest level cache is generally much faster than main memory, since the latter is accessed via the *off-chip* memory bus. To produce precise WCET estimations, cache analysis should be conducted all the way to the highest level, instead of merely on the L1 cache.

— *Separate vs. Integrated Approaches*

Two major analysis frameworks for multi-level caches are the *separate analysis*, which analyzes caches level by level, and the *integrated analysis*, which deals with the cache hierarchy as a whole.

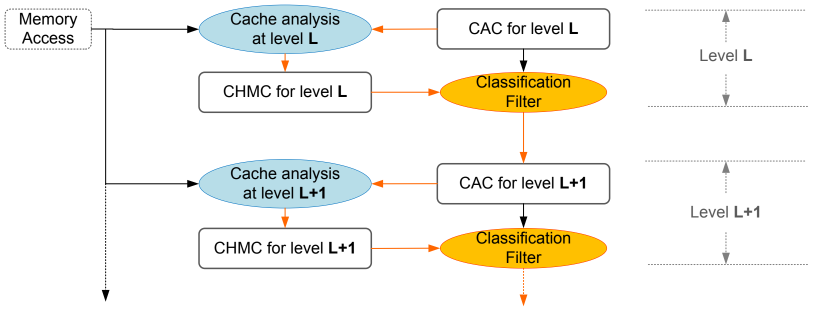


Fig. 16. The separate analysis architecture

The separate-analysis framework was first proposed by Mueller [Mueller 1997] and refined by Hardy and Puaut [Hardy and Puaut 2008], who corrected a soundness problem. Fig. 16 [Hardy and Puaut 2008] shows the main work flow. In the analysis of all but the L1 cache, a key information is whether a data request actually leads to an access to this level. For example, if a memory access is predicted *always hit* at L1, then the L2 cache will not be visited. An interface across cache levels, called *Cache Access Classification* (CAC) is introduced to describe this information. The notion $CAC_{r,L}$ denotes the access property of block r to level L, which is evaluated to one of the following three cases:

— N (Never): the access to r is never performed at level L;
— A (Always): the access to r is always performed at level L;
— U (Uncertain): the access to r at level L can neither be excluded nor predicted.

The CAC values for level L are computed from both the CHMC and the CAC for level $L-1$, which is shown in Table II.

Table II. Computing CAC for level L and memory block r [Hardy and Puaut 2008]

$CAC_{r,L-1}$ \ $CHMC_{r,L-1}$	AM	AH	FM	NC
A	A	N	U	U
U	U	N	U	U
N	N	N	N	N

Trivially, if $CAC_{r,L-1} = A$ (or N), then the access to memory block r is always (or never) considered in the analysis of level L. However, handling the U classification requires special attention. In Mueller's multi-level analysis [Mueller 1997], memory accesses with U classification are "conservatively" treated as *always access* in the analysis of the current cache level. However, this treatment is demonstrated to be unsafe [Hardy and Puaut 2008], since it may underestimate block ages, and thus incorrectly predict cache misses as hits. Hardy and Puaut corrected the problem by considering both possibilities for the U accesses in the update function (shown in Fig. 17), which guarantees that the worst-case scenario is never missed.

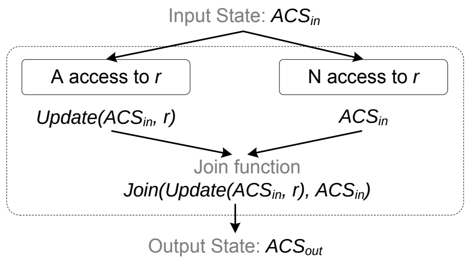


Fig. 17. Update function for U access [Hardy and Puaut 2008]

However, Hardy and Puaut's analysis may suffer from precision problems. Note that in the above CAC computation, the FM classification is treated the same as NC, which means the information obtained by Persistence analysis at level $L-1$ is never leveraged in the analysis of level L. Actually, a block classified as FM at level $L-1$ causes at most one access to level L on its first access. Mueller in an earlier practice tried to solve this problem by unrolling the loop bodies [Mueller 1997], but this approach does not scale.

The component-wise separate analysis has several advantages. First, the analyzer has the flexibility to apply different analysis methods for each cache level, as long as the methods produce hit/miss classifications as the interface across adjacent levels. This is desirable for architectures with different replacement policies for different cache levels, such as in the IBM Power 5 processor. Second, the overall analysis is scalable as long as the adopted single-level analysis is scalable.

However, the separate analysis may be pessimistic due to imprecise transfer of cache access information across cache levels. In contrast, *integrated analysis* [Sondag and Rajan 2010] tries to build a holistic abstract domain for all cache levels, aiming to collect the information lost by the separate analysis.

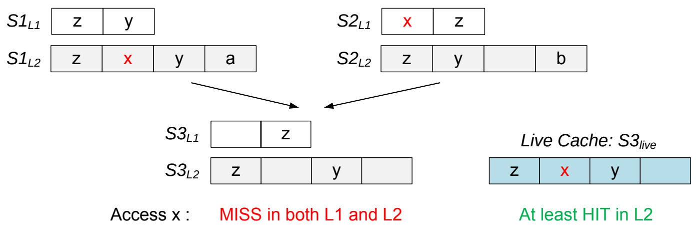


Fig. 18. How a live cache helps to obtain a more precise join operation [Sondag and Rajan 2010]

Consider a Must join operation of a separate analysis with a 2-way L1 cache and a 4-way L2 cache, shown in Fig. 18. Si_{Lj} represents the abstract state Si at Level j. Consider block x, which appears in $S1_{L2}$ on the left branch and in $S2_{L1}$ on the right branch. By a separate analysis, x does not appear in the joined state $S3$ of any level. If a subsequent access to x occurs, a cache hit can not be predicted. However, by evaluating both levels together, it can be seen that x does show up in *every incoming path*, so a subsequent access to x should be a cache hit, either in L1 or in L2 depending on the execution history. This is to say, x is guaranteed in the cache hierarchy at the joined program point. Unfortunately, this information is lost in the separate analysis.

Based on this observation, Sondag and Rajan introduced a new component called *live cache* into the traditional abstract domains. At a join, a block is added to the live cache if it appears in some cache level in every input cache state, as depicted by $S3_{live}$[5] in Fig. 18. With live-cache information, one can now safely predict that x at least hits in the L2 cache. As reported in [Sondag and Rajan 2010], the extra overhead by introducing live cache is acceptable for a 2-level cache hierarchy. However, analysis overhead increases with the number of cache levels, since an independent live cache is maintained for every pair of cache levels.

— *The Impact of Inclusiveness*

The relationship between cache levels is a key design feature. In some processors, all data in level L must be contained in level $L+1$. Such caches are called *(strictly) inclusive* caches. For example in Fig. 19, the access to e causes a to be evicted from L2, so a is forced to be removed from L1 to guarantee inclusion. Inclusive caches are favored in multi-cores: any data update in the shared L2 cache is automatically synchronized to the private L1 caches of all cores due to inclusion enforcement, which also achieves data coherency. Other processors adopt *exclusive* caches, in which data is guaranteed to be in at most one cache level. Exclusive caches are desirable for resource-limited systems since there is no data duplication in the cache hierarchy. The type adopted in previous discussions is called *mainly-inclusive*, which neither enforces inclusiveness nor exclusiveness. Inclusive caches are harder to analyze than exclusive caches,

[5]x has an older age in $S1_{L2}$ than in $S2_{L1}$. This is because Sondag's analysis assumes the write back policy. If x is evicted from the L1 cache, it is installed in the youngest position of the L2 cache. This means x can still suffer another $k-1$ evictions in L2, where k is the cache associativity. The age of a block in the live cache describes how long it can stay in the *whole* cache hierarchy.

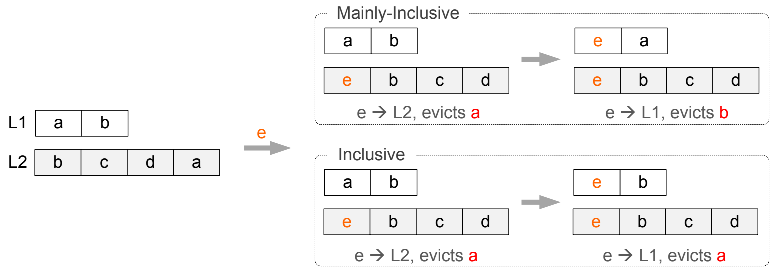


Fig. 19. Handling new behavior of inclusive and exclusive caches

because the update to one cache level may cause further changes to both lower and higher cache levels. Such behavior may preclude the separate-analysis flow.

Hardy et al. adapted the separate analysis [Hardy and Puaut 2008], originally designed for mainly-inclusive caches, to both inclusive and exclusive caches [Hardy and Puaut 2011]. The main idea is to first conduct an analysis assuming a mainly-inclusive cache, and to then modify the CHMC results to guarantee that the effects of cross-level cache updates are safely considered. For example in Fig. 19, the analysis assuming mainly-inclusive cache reports that a is AH at L1 but may be evicted from L2. To adapt this result to an inclusive cache, one must consider the possibility that a can be removed from the L1 cache due to an update in the L2 cache. As a result, a's CHMC at L1 is modified from AH to NC. Similar problems exist in the May analysis as well. As a consequence to CAC computation, accesses to all cache levels except L1 are changed to Uncertain ($CAC_{r,L} = U$ for $L \geq 2$). All these modifications severely degrade the analysis precision.

Sondag and Rajan extended their integrated analysis to both inclusive and exclusive caches [Sondag and Rajan 2010]. The resulting update and join functions for inclusive caches are very complex since once a block is accessed on some level L, the corresponding changes in other cache levels must be correctly considered. Analysis of exclusive caches has similar problems. To summarize, the inter-dependent updates among cache levels to enforce inclusion/exclusion brings new difficulties regardless of the analysis framework.

— *The Impact of Write Operations*

A write to the cache occurs when a data variable receives a new value. Two levels of policies determine when and where to conduct the writing of data back to memory. The *write-through* policy requires that the new value is updated synchronously both in the cache and in main memory. In contrast, the *write-back* policy only marks the modified data as *dirty*, and performs the actual update of memory only when the data is evicted from the cache. A *write miss* occurs if the data to write are not in the cache. Under the *write allocate* policy, missed data are first loaded into the cache, and then updated with the new value, resulting in a cache miss followed by a cache hit. For the *non-write allocate* policy, the data are directly written to main memory, bypassing the caches.

The write-through policy is generally easy to handle in cache analysis, since data writes to a certain cache level incur no change to other cache levels. However, for the write-back policy, evicted dirty data are written to higher cache levels. Second, for the write allocate policy, a write operation always causes cache accesses regardless of hit or miss, which makes no difference from the read operation. However, for non-write allocate caches, a write miss never causes a cache access, so one cannot simply assume that each write operation changes the cache state, as is the case for reads. Like the inclusiveness enforcement, complex write operations cause cross-level cache updates, which is a challenge to the analysis.

Hardy's separate analysis framework has been extended to multi-level data caches by Lesage et al. [Lesage et al. 2009] and to multi-level unified caches by Chattopadhyay and Roychoudhury [Chattopadhyay and Roychoudhury 2009]. However, both analyses assume a write-through policy. The abstract domains adopted in these methods are not able to handle write-back. Sondag and Rajan modeled write-back behavior in their integrated analysis [Sondag and Rajan 2010]. It is shown that modeling write-back is easier by an integrated abstract domain, but one still has to carefully distinguish *possibly evicted blocks* from *definitely evicted blocks* at any level during the analysis to guarantee soundness. Definitely evicted blocks are identified from the May information in Sondag's method. Due to the access uncertainty of possibly evicted blocks, conservative update and join operations have to be employed, causing a loss in precision.

4. ANALYSIS OF NON-LRU CACHES

In the past two decades, most research on cache analysis in the real-time domain was focused on the LRU replacement policy. The analysis of non-LRU replacement policies, i.e., those widely adopted in real-life processors, is still immature. In this section, we look into the challenges for non-LRU analysis and survey existing techniques.

4.1. Why Are Non-LRU Replacement Policies Hard to Analyze

To answer this question we explore why it is hard to design *precise* and *efficient* abstract domains and the corresponding operations for non-LRU caches. We identify multiple challenges discussed in the following paragraphs.

— *Unsuitability of AH, AM, and FM Classifications*

Under LRU replacement most memory accesses can be classified as AH, AM, or FM. Are these classifications equally suitable for non-LRU replacements? If not, what are alternatives that are better suited to characterize other policies' behavior?

Unfortunately, these classifications are not as suitable for non-LRU replacements as they are for LRU. As shown in Guan et al.'s analysis [Guan et al. 2013], under FIFO replacement memory accesses may exhibit alternative hit and miss behavior so that none of the traditional classifications, i.e., AH, AM or FM. Similarly, under MRU [Guan et al. 2012] many cache accesses exhibit the *K-Miss* property. An access classified as *K-Miss* suffers several misses (bounded by $K \leq \mathit{cache\ associativity}$) upon the first few accesses, and then persists in the cache. This kind of persistence property, however, is not captured by the FM classification. This demonstrates that one needs to better understand the specific cache behavior under different policies to come up with proper classifications.

— *Irregular and Non-monotone Cache Update Behavior*

The abstract domains and the corresponding transfer functions are designed to compute cache behavior invariants. An abstract domain is precise and efficient if (1) abstract states can compactly represent many concrete states, while preserving the information required for classification, and at the same time (2) transfer functions precisely capture the effect of a memory access on the concrete cache states. For example in the AI-based analyses for LRU, the block age bounds in the abstract states capture precisely the information required to classify blocks as cached or not. Further, this information can be precisely maintained by the transfer functions due to LRU's regular cache update: a) whether or not a block ages depends solely on its age relative to the accessed block's age. Upper and lower bounds (in Must and May analyses) on the ages of blocks can be precisely updated due to the monotonicity of the operation, b) regardless of its previous age, and whether it was cached or not, the accessed block is always assigned the youngest age.

Unfortunately, most non-LRU replacement policies do not possess such monotone behavior. Take the FIFO replacement in Fig. 4(a) for example. After a hit to d, d remains in the original position and is immediately evicted by the next access to x. The fact that d is *recently* accessed is not reflected by the update rules. An example of irregular behavior under MRU is shown in Fig. 4(c). After f is installed into the cache, a subsequent hit to c followed by a miss to e evicts f out of the cache. However, block b, which is older than f, remains in the cache even after f is evicted. The problem of PLRU is shown in Fig. 4(b). In the state before a is accessed, the oldest block that will be evicted next is b. However, after a hit to a, the block to be evicted is changed to d.

To build efficient abstract domains for such replacement policies is very difficult. For example, in a Must analysis for FIFO [Grund and Reineke 2009], early determination of cache misses is helpful to better predict cache hits later. However, a very complex May analysis has to be designed to determine miss information as early as possible. For PLRU, a precise analysis must model the tree bits. The Must analysis for PLRU [Grund 2011] by Grund employs a far more complex abstract domain, compared to LRU, to express information in the tree and predict cache hits.

— *The Influence of Initial States*

Cache behavior heavily depends on the execution history. In [Reineke and Grund 2013], it is shown that program performance under non-LRU replacement policies are very sensitive to the initial cache state, i.e., what remains in the cache before a program starts. This presents a challenge to obtain precise estimations. To illustrate the problem, assume currently m is accessed in a FIFO cache and we want to precisely estimate m's lifetime. There are many possible situations: Case 1: m hits, but it has been in the cache for the longest time among the cached blocks, and thus it will be evicted upon the next cache miss. Case 2: the access to m is a miss and due to first-in, first-out behavior m will withstand another $k-1$ (where k is cache associativity) cache misses without being evicted. Obviously, m's remaining "life expectancy" in the two cases is rather different. If no knowledge on the initial cache states is available, a safe analysis (to predict hits) has to assume the worst case, i.e., *Case 1*. To be more precise, one may try to distinguish *Case 2* from *Case 1* by investigating whether the current access to m is a miss or not. Then, the analysis needs to know there are enough cache misses to evict any previously accessed m out of the cache, which again relies on the initial states. If a replacement policy can remove uncertainty from the initial states quickly, it will be easier to analyze.

To analytically model the effects of unknown initial states, Reineke et al. proposed a metric, *evict*, for a replacement policy [Reineke et al. 2007], as listed in Table III[6]. Intuitively, the value of $evict(k)$ tells us after how long a sequence of pairwise different memory accesses, we can conclude that the cache only contains blocks from the access sequence, or, in other words, how long a sequence of pairwise different accesses is needed to evict unknown cache contents from the cache. The $evict(k)$ results show that generally longer sequences have to be observed for non-LRU replacement policies. This property directly corresponds to the achievable precision by a May analysis to predict misses, and indirectly affects Must analysis, the precision of which partly depends on how much may information can be obtained during the analysis [Grund and Reineke 2009]. Similarly, the $fill$ metric captures the number of pairwise different memory

[6]Table III extracts the HM case for $evict$ and $fill$ with $k > 2$ from the full results in [Reineke et al. 2007], where k is cache associativity.

Table III. Predictability metrics [Reineke et al. 2007]

Policy	$evict(k)$	$fill(k)$
LRU	k	k
FIFO	$2k-1$	$3k-1$
MRU	$2k-2$	$3k-4$
PLRU	$\frac{k}{2}log_2k+1$	$\frac{k}{2}log_2k+k-1$

Table IV. Generalized predictability metrics [Reineke et al. 2007].

Policy	$mls(k)=mls'(k)$	$evict'(k)$
LRU	k	k
FIFO	1	$2k-1$
MRU	2	$2k-2$
PLRU	$\log_2 k+1$	∞

accesses required to reach a single cache state independently of the initial cache state. As can be seen in Table III, the gap between LRU and other policies is even bigger for the $fill$ metric.

The two metrics $evict$ and $fill$ discussed above relate to the precision of *classifying* analyses. In other work, Reineke and Grund [Reineke and Grund 2013] determine how strongly the *number* of cache misses may vary depending on the initial state, which is related to the precision of *bounding* analyses in the presence of uncertainty about the initial state. Their analysis demonstrates that the number of cache misses may vary strongly depending on the initial state under FIFO, PLRU, and MRU, while it may not vary much under LRU replacement. Further, it is shown that the empty cache state is not necessarily the worst initial state for non-LRU policies. This presents severe problems for measurement-based WCET analysis approaches.

4.2. Predicting Cache Hits

The more hits can be predicted the better the WCET bound. Must and Persistence analyses discussed earlier are used for this purpose. To predict a hit for a memory access to m an analysis needs to ensure m has not been evicted since its last access. Intuitively, the more pairwise different blocks have been accessed since the last access to m, the higher the chance that m has been evicted from the cache. To capture this information, we introduce the following definition:

DEFINITION 1 (**STACK DISTANCE**). *Let p be a memory access sequence that ends with an access to memory block m. The* stack distance *of m, denoted by $\mathbf{sd}_p(m)$, is the number of distinct blocks accessed along p since the previous access to m in p.*

The notion of stack distances coincides with that of ages of memory blocks that is used in the analysis of LRU caches. For example, let $p_1 = \langle beabcda \rangle$ and $p_2 = \langle abccdcccba \rangle$, we have $\mathbf{sd}_{p_1}(a) = \mathbf{sd}_{p_2}(a) = 4$. Due to branches or input-dependent memory accesses, there can be multiple access sequences leading to the same access to block m in a program. So we define the *maximal stack distance*, denoted by $\widehat{\mathbf{sd}}(m)$, as the maximal value of $\mathbf{sd}(m)$ over all possible memory access sequences leading to a particular access. We can evaluate analysis techniques by the maximal stack distances for which they can predict cache hits.

Reineke et al. explored the *minimal life-span* [Reineke et al. 2007] for different replacement policies, which is the minimal length of a sequence of pairwise different memory accesses necessary to evict a block that has just been accessed from the cache. The minimal life-span values are given in the $mls(k)$ column of Table IV. A slight variation of $mls(k)$ is $mls'(k)$, which considers the minimal number of pairwise different memory blocks required to evict a block that has just been accessed from the cache.

Notice, the slight difference between the two notions: $mls(k)$ considers only sequences consisting of pairwise different accesses, whereas $mls'(k)$ allows multiple accesses to the same block. For all the considered policies, $mls(k)$ is equal to $mls'(k)$. The metric $evict'$ also listed in Table IV will be discussed in Sec. 4.3. The $mls'(k)$ metric tells us how many of the most recently accessed blocks are guaranteed to be in the cache. By this result, a Must analysis can be constructed as follows: for any memory access to m, one can check if $\widehat{\text{sd}}(m) \leq mls'(k)$ holds for the given replacement policy. If yes, the memory access to m is guaranteed to be a hit. We say that such an analysis explores *Level I*, which is illustrated in Fig. 20.

Notice that to predict cache hits, the Must analysis for LRU presented in Sec. 3.1 computes upper bounds on the maximal stack distances of memory blocks. Similarly, the May analysis computes lower bounds on the minimal stack distance of memory blocks to predict cache misses. As the notion of stack distances is replacement policy-independent, these LRU Must analysis can thus safely be reused to predict hits for other policies, by relying on the policies value of $mls'(k)$.

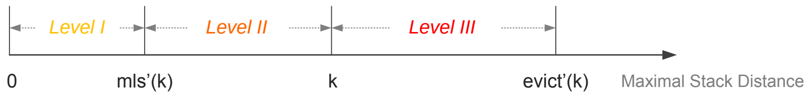


Fig. 20. Levels to explore cache hits

However, the $mls'(k)$ values for non-LRU replacements are commonly small compared to cache associativity k, because they consider worst-case scenarios. In practice, it is unlikely that the worst case occurs at every program point. Thus, analyses tailored to a particular replacement policy can often go beyond *Level 1* in predicting hits.

For a program that can fit into the cache of size k, there is a strong intuition that each block of the program eventually persists in the cache, i.e., after some misses, the remaining accesses to each block are definitely cache hits. For such programs, the maximal stack distance of any access is no larger than k (*Level II* in Fig. 20). This property is attractive as it enables an efficient Persistence analysis that simply collects the set of different blocks accessed by a program. However, such a Persistence analysis is not correct for every replacement policy: it work for LRU, MRU and FIFO, but not for PLRU.

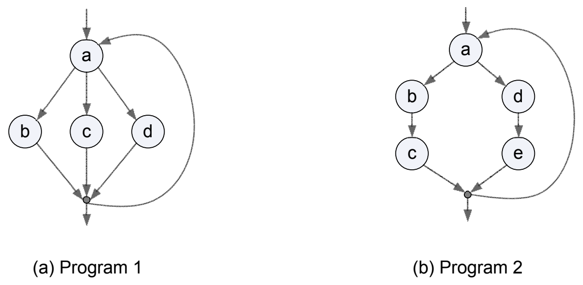


Fig. 21. Example programs

Fig. 22 illustrates the cache state transitions when the loop in Fig. 21(a) is executed, alternating between the three branches in the loop body on a 4-way PLRU cache. Each time a is accessed, the root bit points to the right subtree, so b, c, and d have to compete for the two cache lines on the right. Even though the loop can fit into the 4-way cache

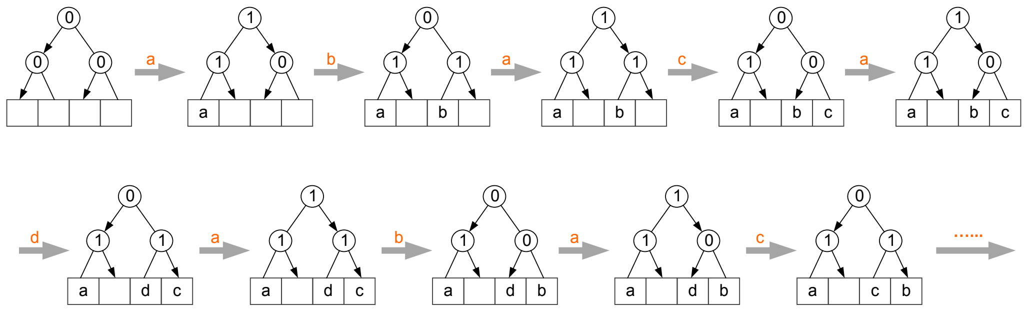


Fig. 22. An example that demonstrates that PLRU does not always use the cache's entire capacity

set, only block a is persistent. This example unveils a negative property of PLRU: it does not always make use of all of its capacity [Berg 2006].

It is crucial to investigate what process a block may go through before it finally persists in the cache. This is important to safely bound the number of misses that may occur.

For MRU, Guan et al.'s results [Guan et al. 2012] show that if for a block m, $\widehat{\mathbf{sd}}(m) \leq k$ holds, m will eventually persist in the cache. However, m may suffer more than one miss before reaching the stable state. The result is strong in that bounds on the number of misses are determined for all stack distances in *Level II* for MRU. Consider the program in Fig. 21(b): even if the program cannot fit into a 4-way MRU cache, block a's number of misses can be bounded by a constant since $\widehat{\mathbf{sd}}(a) \leq 4$.

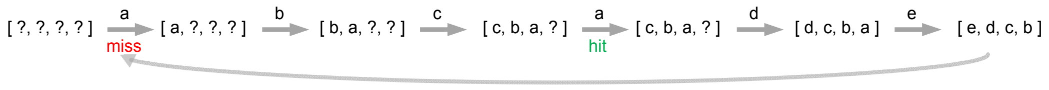


Fig. 23. Alternative hit and miss behavior on FIFO

For FIFO replacement, Grund and Reineke [Grund and Reineke 2010a] show that if a loop entirely fits into the cache, each block suffers at most one miss and then persists in the cache[7]; otherwise, no guarantee is given. Guan et al. further explored this problem, and found that if a block m satisfies $\widehat{\mathbf{sd}}(m) \leq k$, then even though m is not eventually persistent, it is still guaranteed to enjoys cache hits, which can be expressed by a bound on the number of misses that accesses to m may suffer [Guan et al. 2013]. For example in Fig. 21(b), $\widehat{\mathbf{sd}}(a) \leq 4$ holds for a 4-way FIFO cache. In the worst case, the loop alternatively takes the two branches, and a may suffer cache misses repeatedly. To evict a from the cache, both branches have to be taken, which causes a to enjoy a cache hit in the execution of one of the branches (shown in Fig. 23). It can be shown that a suffers at most $\lfloor \frac{1}{2} \cdot x \rfloor + y$ misses, where x is the execution count of a, and y is the total number of times the loop is entered.

The only analysis to predict hits for PLRU for maximal stack distances in *Level II* is a Must analysis proposed in [Grund and Reineke 2010b]. The analysis presented is based on the following observation: to evict a block with the fewest possible accesses to distinct memory blocks, the three bits (assuming an associativity of 8) that are on the path from a cache line to the root of the tree need to be flipped in a particular order.

[7]Note that blocks do not necessarily encounter their misses in the first loop iteration. It may take several iterations for all the blocks to stabilize in the cache.

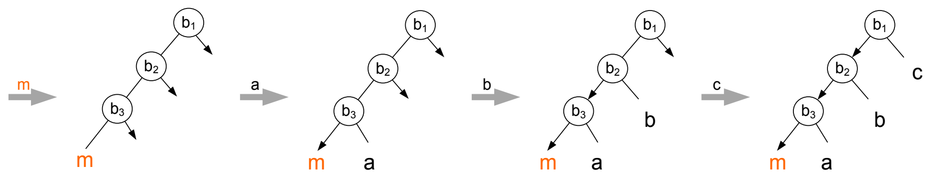


Fig. 24. Scenario in which block m is evicted with the minimal $mls'(k) = \log_2 k + 1$ accesses

Namely from the bottom to the top. This is illustrated in Fig. 24 for block m. Notice that flipping bits near the root before flipping all bits closer to the leaves does not contribute to evicting m, as these bits will eventually be flipped back before evicting m. The basic idea behind the analysis in [Grund and Reineke 2010b] is to track two properties: a) the number of bits that already point towards a block (counting from the leaf of the tree), b) the so-called "sub-tree distance" between pairs of blocks. The sub-tree distance between a and b captures which bits on the path from a to the root an access to b may flip. By analyzing these key properties, it is sometimes possible to predict that a block stays in the cache even if more than $mls'_{PLRU}(k) = \log_2 k + 1$ other blocks have been accessed.

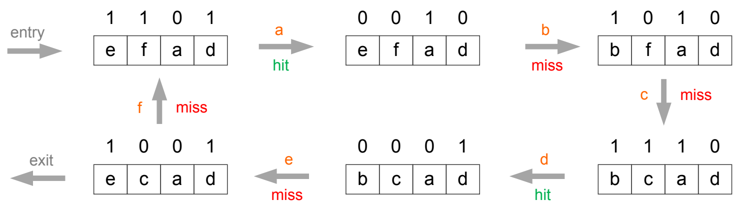


Fig. 25. Cache behavior under MRU in *Level III*

To go beyond *Level II* means to explore whether blocks with maximal stack distance larger than k still have cache hits. One needs to first show that the above fact does occur for some replacement policy, and second propose an analysis to discover the cache hits. Take MRU for example, Fig. 25 shows that even if we have $\widehat{\mathrm{sd}}(a) = \widehat{\mathrm{sd}}(d) > 4$ for a 4-way cache, accessing a and d is always hit. But this phenomenon relies on the initial state at the entry of the loop. To explore such behavior, the abstract domain must be able to preserve very detailed information on cache states. This requirement makes it very hard to explore cache hits in *Level III* by abstract analysis methods. The abstractions introduced by Grund and Reineke for FIFO [Grund and Reineke 2009] are in principle able to predict *Level III* hits, however, they usually require a highly context-sensitive analysis to do so. *Level III* ends at $evict'(k)$, after which no more hits are possible.

4.3. Predicting Cache Misses

Predicting more misses tightens the estimated BCET, but it can also indirectly help with the analyses for some non-LRU replacement policies to predict more hits [Grund and Reineke 2009]. A concrete example is the case of FIFO explained in the discussion of the influence of initial states in Sec. 4.1. Furthermore, for multi-level cache analysis, predicting more misses for level L reduces the uncertainty of cache accesses on level $L + 1$, which leads to more precise overall estimations. Lastly, in microarchitectures with timing anomalies, if a memory access cannot be classified as a cache hit, both the cache hit and the cache miss case need to be explored. Predicting cache misses may in such cases drastically reduce analysis times, as it allows to explore only the cache miss case.

To predict cache misses requires to show that memory blocks are not in the cache right before they are being accessed. Thus a May analysis, i.e., an analysis that over-approximates cache contents, is required to safely predict cache misses. May analyses can be constructed based on a variation of the $evict$ metric [Reineke et al. 2007], which we denote by $evict'$. The difference between $evict$ and $evict'$ is the same as the difference between mls and mls': any sequence s containing $evict'(k)$ distinct memory blocks is guaranteed to evict any prior cache contents not contained in the sequence s. In contrast, $evict$ refers only to sequences that never access the same memory block twice. Values of $evict'$ for common policies are listed in Table IV. For example, for FIFO, $evict'(k) = 2k-1$. This means that after accesses to $2k-1$ pairwise different blocks, the cache only contains elements from the accessed sequence. Then, the May analysis only needs to observe a sequence with $2k-1$ pairwise different blocks other than m precluding the current access to m. On the other hand, for PLRU, $evict'(k) = \infty$. In other words, there are sequences of memory accesses containing an arbitrary number of distinct memory blocks that do not evict all prior cache contents. We have seen an example of such a sequence in the previous section, which is illustrated in Fig. 22.

May analyses based on $evict'$ can be constructed by determining *lower bounds* on the stack distances of memory blocks. The LRU May analysis presented in Sec. 3.1 does exactly that.

The $evict$ metric suggests that less than $evict(k)$ accesses to pairwise different memory blocks do not allow to predict any misses, thereby constituting a limit on how much information a May analysis can obtain. However, note that this conclusion is built on the assumptions that the initial state is *completely unknown*, and that the access sequence consists of *pairwise different* memory accesses. There is thus hope that by tailoring an abstract domain to a specific replacement policy, more precise May analyses can be achieved. So far, such abstractions have only been built for the FIFO replacement policy [Grund and Reineke 2009; Grund 2011]. Due to limited space, we only explain the main intuitions and the key constituents of the two existing FIFO abstract domains.

Consider a 4-way FIFO cache and the access sequence $x \circ s \circ x$, where the first access to x is a miss and installs x into the "first-in" cache way, and then a sequence s that does not contain x is accessed followed by another access to x. To predict a miss for the second access to x, it suffices to check whether either of the following two properties holds:

- *Property 1*: The accesses in s result in at least 4 misses;
- *Property 2*: Before the second access to x, every cache way is occupied by a memory block from s.

The abstract domain proposed in [Grund and Reineke 2009], which we call $FIFO^{\alpha}$, checks *Property 1*. The key information to be maintained in the abstract domain is the number of *definite misses* after the first access to x, denoted by $dm(x)$. When $dm(x)$ reaches 4 during the analysis, one can predict misses for a future access to x. To better maintain definite misses, the number of *cache ways covered* by blocks accessed after the last access to x is maintained as auxiliary information.

A disadvantage of $FIFO^{\alpha}$ is that it only starts to predict misses after $2k-1$ pairwise different memory blocks have been accessed, which is in line with the $evict$ metric. Therefore, Grund and Reineke proposed another FIFO domain, which we denote by $FIFO^{\beta}$, so that cache misses can be predicted even if fewer than $2k-1$ pairwise different blocks are accessed between two different accesses to the same block [Grund and Reineke 2010a; Grund 2011]. The domain $FIFO^{\beta}$ checks *Property 2* to predict cache misses. For the above example, it explores if memory blocks accessed in sequence s

eventually cover all the 4 cache ways. The exploration is based on a more powerful result (Lemma 4 in [Grund and Reineke 2010a]):

If a sequence s contains l distinct blocks, then $l - k + 1$ cache ways must be occupied by the contents of s, regardless of the initial cache state.

Importantly, the effect of consecutive sequences adds up. For example, let $s = s_1 \circ s_2 = \langle a, b, c, d, e\rangle \circ \langle a, b, c, d, e\rangle$. The accesses to s_1 cover the $5–4+1 = 2$ most-recently-used ways in the cache set. Similarly, the accesses to s_2 contribute another $5–4+1 = 2$ to the covered positions. Then we can guarantee that access to s finally covered all the 4 cache ways[8]. This means the execution of s actually evicts x out of the cache and a miss on the second access to x can safely be predicted.

So far, no May analysis is known for PLRU. For MRU the best known May analysis is based on $evict'$. Precisely predicting misses for these two policies is still a challenge.

4.4. The Relative Competitiveness Framework

Besides the above research, Reineke and Grund proposed the Relative Competitiveness framework [Reineke and Grund 2008] which allows to translate analysis results for one replacement policy to another policy. The promise is then to apply known LRU analyses to non-LRU caches.

A policy P is (k, c)-*hit-competitive relative* to policy Q if the number of cache hits $h_P(s)$ of P on sequence s is bounded from *below* by the number of cache hits $h_Q(s)$ of Q as follows: $h_P(s) \geq k \cdot h_Q(s) - c$. Similarly, a policy P is (k, c)-*miss-competitive relative* relative to policy Q if the number of cache misses $m_P(s)$ of P on sequence s is bounded from *above* by the number of cache misses $m_Q(s)$ of Q as follows: $m_P(s) \leq k \cdot m_Q(s) + c$.

By monotonicity of the two inequalities, they can also be applied to lower bounds on the number of hits and upper bounds on the number of misses: For example, given a lower bound on the number of hits of Q using hit-competitiveness a lower bound on the number of hits of P can be derived.

For $(k, c) = (1, 0)$ the notions of (k, c)-hit- and miss-competitiveness coincide. In this case, P "dominates" Q. In other words, P never incurs more misses than Q. In such a case we simply say that P is $(1, 0)$-competitive relative to Q. Then, a Must analysis for Q is a valid Must analysis for P; conversely, a May Analysis for Q is a valid May Analysis for P.

In [Reineke and Grund 2008] it is shown how to automatically compute the best values for (k, c) such that policy P is (k, c)-hit/miss-competitive relative to policy Q, for fixed associativities of the two policies. Depending on the similarity of P and Q this computation scales to associativities between 8 and 256.

The most interesting cases are those in which either P or Q is LRU, as precise analyses for LRU are known. Examples for hit-competitiveness results derived in this way are [Reineke and Grund 2008; Reineke 2008]:

- An 8-way FIFO cache is $(\frac{1}{2}, \frac{7}{2})$-hit-competitive relative to an 8-way LRU cache.
- An 8-way FIFO cache is $(\frac{2}{3}, 2)$-hit-competitive relative to an 4-way LRU cache.
- An 8-way PLRU cache is $(\frac{1}{2}, \frac{3}{2})$-hit-competitive relative to an 6-way LRU cache.
- An 8-way MRU cache is $(\frac{2}{3}, \frac{4}{3})$-hit-competitive relative to a 4-way LRU cache.

To use this relation, assume 100 hits are predicted for a program on an 4-way LRU cache, then $\lceil \frac{2}{3} \times 100 - 2 \rceil = 65$ hits are guaranteed on an 8-way FIFO cache.

The metrics $mls'(k)$ and $evict'(k)$ are strongly related to $(1, 0)$-competitiveness relative to LRU. In particular, let $mls'_P(k)$ and $evict'_P(k)$ denote the values of the two

[8]The effectiveness of this analysis depends on how a long sequence is partitioned. Grund has a systematic method to explore different partitionings for optimization [Grund 2011].

metrics under policy P. Then, P is $(1, 0)$-competitive relative to $LRU(mls'_P(k))$ and $LRU(evict'_P(k))$ is $(1, 0)$-competitive relative to P. For example:

- $LRU(2k - 1)$ is $(1, 0)$-competitive relative to $FIFO(k)$.
- $LRU(2k - 2)$ is $(1, 0)$-competitive relative to $MRU(k)$.
- $PLRU(k)$ is $(1, 0)$-competitive relative to $LRU(\log_2 k + 1)$.

Cache analyses based on relative competitiveness can be pessimistic, because the relation holds for *any* possible workload. Moreover, the framework provides bounds on hits (or misses) for the whole program or alternatively program fragments rather than classifying independent memory access, except for the case of $(1, 0)$-competitiveness. This makes it difficult to apply the approach in multi-level cache analysis, or in integrated analyses considering both caches and pipelines.

5. EXECUTION ENVIRONMENTS

Discussions so far have focussed on analyzing an independent program. Cache analysis is severely challenged in the presence of complex execution environments, such as multi-tasking systems or shared-cache multi-cores, where extra time delay due to interference on caches from other co-scheduled/running programs must be taken into account.

5.1. Cache-Related Preemption Delay

An essential feature of real-time systems is preemption, which allows a higher priority task to preempt a lower priority task so that the higher priority one meets its deadline. However, preemptions may lead to extra cache misses: the execution of the preempting task may alter the cache state, so that once resumed, the preempted task needs to bring data back into the cache that was evicted as a consequence of the preemption. The extra delay due to cache reloading is commonly referred to as the *Cache-Related Preemption Delay* (CRPD). Empirical results [Liu and Solihin 2010] show that CRPD contributes signficantly to the execution time, so it must be precisely estimated to obtain tight estimations of response times. Furthermore, it has also been shown that with CRPD, the synchronous release of all higher priority tasks does not represent the critical instance of single-core preemptive scheduling [Yomsi and Sorel 2007]. Clearly, preemptions introduce a new dimension of complexity into timing analysis.

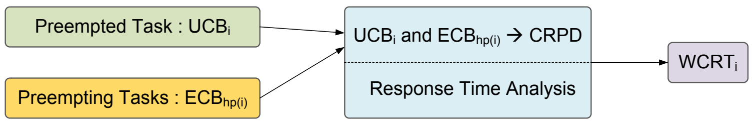

Preempted Task : UCB$_i$
Preempting Tasks : ECB$_{hp(i)}$
UCB$_i$ and ECB$_{hp(i)}$ → CRPD
Response Time Analysis
WCRT$_i$

Fig. 26. Separate CRPD analysis framework

The most intensively studied framework is *separate* CRPD analysis, in which the CRPD is treated as a separate overhead rather than as a part of the WCET of the preempted task. To bound the CRPD under LRU replacement, two approaches have been proposed, which are illustrated in Fig. 26:

(1) By analyzing the preempted task [Lee et al. 1998; Negi et al. 2003; Tan and Mooney 2004; Staschulat and Ernst 2007; Altmeyer and Burguière 2009]: Additional misses can only occur for *useful cache blocks* (UCBs), i.e., blocks that may be cached and that may be reused later, resulting in cache hits. For LRU, the number of such UCBs is a bound on the number of additional misses due to preemptions. Static analyses have been proposed to safely approximate the set of UCBs.

(2) By analyzing the preempting task [Tomiyama and Dutt 2000; Negi et al. 2003; Tan and Mooney 2004; Staschulat and Ernst 2007]: The preempting task may only cause additional cache misses in those cache sets that it modifies. Thus, analyses to compute bounds on the number of *evicting cache blocks* (ECBs) have been developed. A memory block is an ECB if it may be accessed during the preempting task's execution. However, for set-associative caches, approaches based purely on ECBs have so far been either imprecise [Burguière et al. 2009] or unsound [Tan and Mooney 2004], as shown in [Burguière et al. 2009].

The CRPD is computed as the total time delay of all preemption-related cache misses. The final step is to take into account the computed CRPD bounds in a schedulability analysis framework, so that the Worst-Case Response Time (WCRT) of the preempted task can be obtained.

— *Computing UCBs and ECBs*

Since a preemption must happen before some instruction, here we first consider what happens at a particular program point. Most existing work adopts the UCB definition in [Lee et al. 1998]. A cache block m is useful at a given program point p, if:

(1) m may be cached at p;
(2) m may be reused at some program point reachable from p without being evicted along the corresponding path.

To determine memory blocks that satisfy Condition (1), one needs to collect the set of blocks that may be cached by any possible program path from the starting point of the CFG to p, referred to as Reaching Cache Blocks (RCBs) and denoted by RCB_p. This corresponds to a May analysis as discussed in Sec. 3.1. To determine memory blocks that satisfy Condition (2), a set of Live Cache Blocks (LCBs), denoted by LCB_p, is computed similarly to RCB_p, however, by a *backward analysis*. Then, an overapproximation of the set of useful cache blocks at point p, UCB_p is obtained by the intersection of RCB_p and LCB_p. A bound on the CRPD is then obtained by taking the maximum size of UCB_p over all program points.

For direct-mapped caches, two major techniques exist: set-based analysis [Lee et al. 1998] and state-based analysis [Negi et al. 2003]. Both techniques rely on dataflow analyses to collect the RCBs and LCBs at each program point. State-based analysis maintains all possible concrete cache states at a program point. The analysis is precise, but does not scale to large programs. In contrast, set-based analysis maintains one abstract state at each program point, which collects the set of all possible cached blocks for each cache line. Staschulat and Ernst [Staschulat and Ernst 2007] proposed a scalable precision analysis that presents a trade-off between the above two analyses. The main idea is to pose a bound on the number of cache states maintained at each program point. Whenever the number of states goes beyond the limit, cache states are merged.

Regarding the analysis of the preempting task, note that (a) what matters is the size of the set of evicting cache blocks and not its actual contents, and (b) sizes that exceed the associativity of the cache do not have to be distinguished, as they will evict all prior cache contents anyway. For those reasons bounds on the number of ECBs can be obtained from bounds on the number of reaching cache blocks at the end of program execution, i.e., RCB_{end}.

— *CRPD Computation for Direct-Mapped Caches*

For direct-mapped caches, the CRPD can be estimated by only considering the preempted task, which pessimistically assumes that each UCB of the preempted task could be evicted by the preempting task [Lee et al. 1998]. These techniques are classified as the UCB-Only approach by [Altmeyer et al. 2012]. The CRPD can also be computed by only considering the preempting tasks [Busquets-Mataix et al. 1996; Tomiyama and Dutt 2000], which assumes any ECB of a preempting task may cause a preemption related cache miss (ECB-Only by [Altmeyer et al. 2012]). Clearly, more precise CRPD can be computed by evaluating both the preempting and the preempted tasks. Specifically, the ECB-Only approaches have been improved by considering the preempted tasks, resulting in the UCB-Union class [Tan and Mooney 2007]; similarly, the UCB-Only approaches have been extended into the ECB-Union class [Altmeyer et al. 2012].

Schedulability analysis needs to take the CRPD into account. Consider a widely adopted schedulability analysis shown in Equation (5), where R_i is the response time, C_i is the WCET of a task, and T_j is the activation period. Equation (5) can be interpreted in the following way: the preemption cost of task τ_i preempted by τ_j, denoted by $\gamma_{i,j}$, is seen as an *extra* part of the execution time of the *preempting task* τ_j.

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (C_j + \gamma_{i,j}) \tag{5}$$

In the presence of nested preemptions, as shown in Fig. 27, the response time of τ_3 includes both the indirect cost of τ_1 preempting τ_2 (γ_a) and the direct cost of τ_2 preempting τ_3 (γ_b). The main problem is how to safely account for γ_a. Actually, γ_a can be considered in $\gamma_{3,1}$. Note that γ_a may be larger than γ_b, so a safe $\gamma_{3,1}$ needs to account for the maximal cost of τ_1 preempting any lower priority task, however not lower than τ_3. Note that the ECB-Only approaches do no suffer from such nested preemption problems since they do not consider the preempted task.

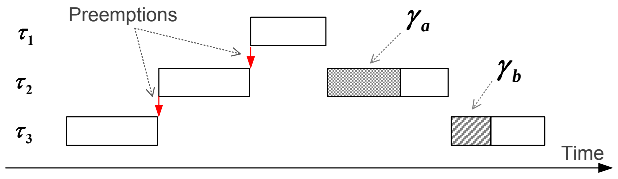


Fig. 27. An example of nested preemptions

A disadvantage of analyses by Equation (5) is: the worst-case delay $\gamma_{i,j}$ is always assumed for each preemption (τ_j preempting τ_i). As a result, some cache evictions can be included multiple times. To reduce this pessimism, other approaches [Staschulat et al. 2005; Altmeyer et al. 2012] adopted the schedulability test of Equation (6) instead, which evaluates the total cost of multiple preemptions of τ_j preempting τ_i as a whole. The computation of $\gamma_{i,j}^{sta}$ differentiates preemption scenarios, and thus can avoid unnecessary inclusion of cache evictions.

$$R_i = C_i + \sum_{\forall j \in hp(i)} (\left\lceil \frac{R_i}{T_j} \right\rceil C_j + \gamma_{i,j}^{sta}) \tag{6}$$

Altmeyer et al. provide a detailed classification of different approaches to bound the CRPD for direct-mapped caches and their relationship [Altmeyer et al. 2012].

— *CRPD Computation for Set-Associative Caches*

The computation of UCBs and ECBs can be solved by existing May analyses for set-associative caches. The main challenge is how to precisely and safely compute the "intersection" between the sets of UCBs and ECBs.

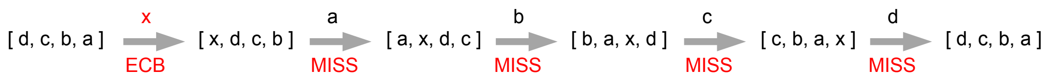


Fig. 28. An example of reordered misses

Let us first discuss a safety problem, given LRU replacement. Consider the case in Fig. 28. Blocks a, b, c and d are all useful blocks. A preemption installs x into the cache set and thereby evicts a. The subsequent accesses of the preempted task to a, b, c and d are all cache misses even though the preempting task only evicted one cache block. This illustrates that there are two types of context-switch misses [Liu and Solihin 2010; Burguière et al. 2009]. The miss to a is a *replaced miss*, as a direct result of the preemption. In contrast, the misses to b, c and d are an indirect result of *reordering* of blocks by the LRU replacement policy. This example shows that even a single ECB can lead to a chain of misses to multiple UCBs, which cannot happen for direct-mapped caches. An example of an unsafe analysis is [Tan and Mooney 2007], which overlooked reordered misses.

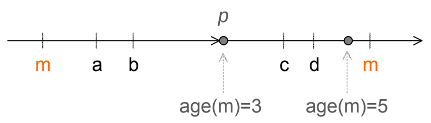


Fig. 29. The notion of resilience

One way to cope with this problem was proposed in [Burguière et al. 2009]. As soon as there is a single ECB that maps to a particular cache set, all the UCBs that map to the same cache set are assumed to contribute to context-switch misses. This is obviously conservative, and it can be improved by obtaining more detailed information about the useful cache blocks. This information is captured by the notion of *resilience* introduced by [Altmeyer et al. 2010]. The resilience $res(m)$ of a useful cache block m is the amount of "disturbance", i.e. its ECBs, by a preempting task that the block may endure before becoming useless to the preempted task. Consider a useful cache block m for an 8-way LRU cache in Fig. 29, where all blocks are mapped to the same cache set. The maximal age of m before its second access is 5. If the program is preempted at any point between the two accesses to m, for example at program point p, m will not be evicted from the cache as long as at most 3 ECBs from the preempting task map to the same set. So m's resilience is 3.

By computing lower bounds on the resilience of useful cache blocks, one can exclude many cache misses compared with the conservative assumption in [Burguière et al. 2009]. However, nested preemptions must be very carefully handled. The ECBs from nested preempting tasks may accumulate to age a useful block. In this case, the ECBs of all possible preempting tasks must be considered, which may adversely introduce some pessimism.

The problem of reordered misses is rooted in LRU. A new policy called Selfish-LRU [Reineke et al. 2014] has been proposed to eliminate reordered misses. The idea is to first evict cache blocks that do not belong to the currently active task.

For other replacement policies, such as FIFO and PLRU, the number of additional misses can even be greater than the number of UCBs, the number of cache ways, and the number of ECBs [Burguière et al. 2009]. This makes it difficult to obtain precise CRPD bounds for these policies. Am approach based on relative competitiveness [Reineke and Grund 2008] was sketched in [Burguière et al. 2009] that allows bounding the total misses (intra- and inter-task misses) of a non-LRU policy from the results of LRU. Due to the generic nature of the relative competitiveness framework, the analysis results can be imprecise.

— *Further Approaches*

It has been observed [Altmeyer and Burguière 2009] that some pessimism is introduced by independently computing bounds on the CRPD and on the WCET. Consider the treatment of memory accesses to blocks that have been classified as useful cache blocks during the WCET analysis. If such accesses cannot be guaranteed to result in cache hits, a sound WCET analysis will also cover the cache miss case. However, in that case, while a preemption-related cache miss may occur in reality, it has already been accounted for in the computed WCET bound. Motivated by this observation, a notion of *definitely-cached* UCBs has been proposed in [Altmeyer and Burguière 2009], which excludes such blocks from the CRPD computation. This approach relies on a coupling of WCET and CRPD analysis and may improve precision significantly.

CRPD analysis under dynamic priority scheduling has also been studied [Ju et al. 2007; Lunniss et al. 2013]. The difference lies in the CRPD calculation for different schedulability tests of new scheduling policies. [Lunniss et al. 2014] compares the effectiveness of fixed priority scheduling and EDF in the presence of CRPD.

5.2. Shared Caches in Multi-Cores

Nowadays, multiple processing cores are deployed on a single die to fully exploit the real estate of the processor chip and to achieve high performance with low power consumption. A commonality among modern multi-core processors is the sharing of on-chip resources among multiple cores, such as the last-level cache (Fig. 30), so that each core can potentially make use of the entire resource. However, tasks running in parallel on different cores compete for the shared resource, resulting in *inter-core conflicts*, also referred to as *inter-core interference*. Due to this interference, the execution time of a program now also depends on the resource-access behavior of the tasks running in parallel [Yan and Zhang 2008].

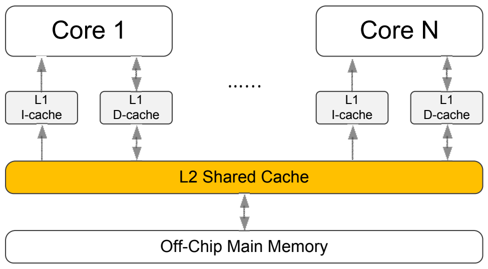


Fig. 30. A Common Shared Cache Design in Multi-Cores

Inter-core interference on a shared cache is different from inter-task interference due to preemption. First, in a single-core preemptive system, a higher priority task does not suffer from interference by a lower priority task, while in multi-core systems, all tasks running in parallel on different cores interfere with each other independently of their priority level. Second, in a single-core preemptive system, a task can only suffer from interference by preempting tasks a small number of times, no more than the total number of releases of the higher priority tasks; in contrast, on multi-cores, interference on a shared cache may come between any two consecutive cache accesses of a task. Precisely analyzing all possible interleaving cache accesses on a shared cache is notoriously difficult due to the huge number of cases to consider.

One approach is to extend the AI-based analysis to take into account the interference on the shared cache [Liang et al. 2012]. The basic idea is similar to the resilience analysis in CRPD analysis. Assume that two tasks, A and B, run in parallel and share a k-way L2 cache. To estimate A's WCET, multi-level analyses for A are first conducted without considering the interference from task B. Then, a second step analyzes task B to see whether its interference could cause the blocks of A that are guaranteed to hit when A runs in isolation, to be evicted from the cache. Consider a block m of A: its maximal age, $age(m)$, in the cache can be extracted from a Must analysis. Then task B is analyzed to determine a bound on the number of interfering memory blocks that map to the same cache set as m , denoted by $\mathcal{M}$. If $\mathcal{M} \leq k - age(m)$ holds, m willl remain in the cache even in the presence of B's interference. Otherwise, m could be evicted from the cache due to B's execution, and its classification needs to be changed accordingly.

A major drawback of the above approach is that the timing of cache conflicts is not considered, i.e., all potential cache conflicts computed from cache mapping are included. However, if by some means we know that the lifetimes of two conflicting tasks (or cache accesses) do not overlap, some cache conflicts can be safely excluded. This is a key property to tighten the estimations. Zhang and Yan proposed a technique to exclude infeasible conflicts by exploring conflicting pairs of cache accesses [Zhang and Yan 2009]. Liang et al. in [Liang et al. 2012] explore the overlapping of the lifetimes of co-running programs. The timing of cache conflicts can also be precisely captured by model checking. Gustavsson et al. used timed automata to model the behavior of programs on shared caches [Gustavsson et al. 2010]. Infeasible conflicts can be precisely excluded when the UPPAAL model checker explores the system model. However, due to state space explosion, model checking based analysis can hardly scale beyond 2 cores. Another model checking based method was proposed in [Wu and Zhang 2012]. The SPIN model checker was adopted to exclude the infeasible cache conflicts, but the models did not explore the exact timing of the cache conflicts.

Another major analysis obstacle is that uncertainty, introduced by particular analysis technique or inherent to a hardware feature, may be amplified in the presence of shared caches. For example, in AI-based analyses, pessimistic age prediction makes the blocks of the interfered task less resilient; similarly, pessimistic age prediction for the interfering task leads to an overestimated number of conflicting blocks. Another source of uncertainty is the separation of cache behavior analysis and path analysis [Theiling et al. 2000], which adversely introduces "architecturally-infeasible" paths. Pruning such infeasible paths can help to tighten WCET estimations. Banerjee et al. proposed a finer-grained abstract domain, which associates path information into the traditional Must and May abstract states to exclude non-existent cache states due to infeasible paths [Banerjee et al. 2013]. Chattopadhyay and Roychoudhury propose another technique which improves the prediction for NC blocks by excluding infeasible paths using model checking [Chattopadhyay and Roychoudhury 2011]. Both techniques can be integrated into the analysis framework of [Liang et al. 2012] to more precisely estimate shared cache interference.

Even with the above techniques, real-time system design still faces a problem: if the shared cache is freely used, the worst-case performance of the tasks also degrades. Therefore, recent research tried to employ mechanisms that provide temporal isolation on shared caches, which both simplifies cache analysis and at the same time reduces the WCET. Cache partitioning [Suhendra and Mitra 2008; Liu et al. 2010; Ungerer

Table V. WCET analysis tools supporting static cache analysis

Tools	Instruction Cache	Data Cache	Multi-Level Cache	Non-LRU Cache	CRPD	Shared Cache
aiT	AI	AI		Pseudo-RR, PLRU, FIFO		
OTAWA	AI, ML-PER					
Chronos		Scope-Aware	Separate Framework			
SymTA/P	Enhanced Pigeonhole				Separate Analysis	
Heptane	AI	AI	Separate Framework			AI
WCA	Model Checking			FIFO		
SWEET	AI					
METAMOC	Model Checking	Model Checking		Round Robin		
McAiT	AI		Separate Framework			Model Checking
Florida	SCS	SCS, CME	Separate Framework			
Chalmers	Symbolic Execution	Symbolic Execution				

et al. 2010] partitions the cache space among tasks by controlling page allocation to completely avoid cache conflicts among tasks on different cores. Cache locking [Suhendra and Mitra 2008; Liu et al. 2010] locks the frequently used data in the cache so that hit/miss behavior is totally predictable. Another approach [Hardy et al. 2009] is to bypass the shared cache upon accesses to memory blocks with little reuse. This reduces cache interference. On system level, some further issues have to be solved. In multitasking systems, different tasks may try to lock the same cache segment, so scheduling of the lockings must be considered [Ward et al. 2013]. Regarding cache partitioning, the partitions assigned to the tasks may overlap in cache space. A task can only start execution if both the CPU and the cache partition are available. The schedulability tests must consider both the CPU and the cache constraints [Guan et al. 2009]. However, partitioning and locking have a side-effect of reducing the cache space available for each task. New techniques are expected for more intelligent resource allocation and arbitration, so that the WCET of the tasks can be further reduced[9], and the schedulability of the overall system is improved.

6. STATIC ANALYSIS TOOLS

In the past decades, a number of WCET analysis tools have been developed in both industry and academia. Table V lists the tools that support static cache analysis.

aiT [Heckmann and Ferdinand 2014] is the only successful WCET analysis tool in industry. It uses the AI-based analyses [Ferdinand and Wilhelm 1999; Cullmann 2013] for both instruction and data caches. Besides LRU, the aiT tool can analyze three non-

[9]Unlike the general-purpose computing domain [Zhang et al. 2009], cache management in real-time systems [Mancuso et al. 2013] optimizes the worst-case rather than the average-case performance.

LRU replacement policies: Pseudo-Round-Robin [Heckmann et al. 2003] as well as, PLRU and FIFO based on the analyses described in [Reineke and Grund 2008] and [Grund et al. 2011].

The OTAWA tool [Ballabriga et al. 2010] developed by the University of Toulouse, France is an open framework for WCET analysis. OTAWA provides instruction cache analysis based on abstract interpretation [Ferdinand and Wilhelm 1999] with the improvement of multi-level Persistence analysis [Ballabriga and Casse 2008].

Chronos [Li et al. 2007] is a static WCET analysis tool from the National University of Singapore. It was originally designed with a highlight on pipeline analysis using the SimpleScalar simulator. The latest version, Chronos 4.2, now supports the recent contributions of the group: scope-aware data cache analysis [Huynh et al. 2011] and unified cache analysis [Chattopadhyay and Roychoudhury 2009].

SymTA/P [SymTA/P 2010] is a timing analysis tool developed by TU Braunschweig, Germany. The tool generally uses measurement-based approaches for instruction and data cache analysis. It also supports static analysis for data caches [Staschulat and Ernst 2006] and CRPD analysis based on [Staschulat et al. 2005].

Heptane [Heptane 2013] is a static WCET analysis tool developed by IRISA, France. The highlight of the tool is the separate analysis of multi-level caches [Hardy and Puaut 2008]. It also support shared cache analysis by the technique extended from AI-based analysis [Hardy et al. 2009].

WCA [Schoeberl et al. 2010] from Vienna University of Technology and DTU is a WCET analysis tool for a Java processor, JOP [Schoeberl 2008], which uses *method cache* to store the instructions of a whole Java method. A method is fully loaded into the cache upon invocation and enjoys cache hits during its execution. On exit, the content of the caller function is reloaded into the method cache. The method cache is organized like a fully-associative FIFO cache with N blocks. The tool uses model checking to analyze the method cache, and it also provides a simple persistence analysis given that a code region can fit into the cache.

SWEET [SWEET 2012] is a WCET analysis tool currently maintained by Mälardalen University of Sweden. Although mainly focused on flow analysis, it supports AI-based analysis for instruction caches [Ferdinand and Wilhelm 1999].

The METAMOC tool [Dalsgaard et al. 2010b] from Aalborg University of Denmark employs model checking for both instruction and data caches. It can analyze the round-robin replacement policy used by the ARM920T processor.

McAiT [Lv et al. 2011] is a WCET analysis tool jointly developed by Uppsala Unviersity of Sweden and Northeastern University of China. The tool supports L1 instruction cache analysis by the AI-based approaches [Ferdinand and Wilhelm 1999; Cullmann 2013], and shared L2 cache analysis by model checking.

Other research prototypes include a tool from Florida State, North Carolina State, and Furman Universities, which adopts Static Cache Simulation [Mueller and Whalley 1995] for both instruction and data cache analysis, and also supports data cache analysis using cache miss equations [Ramaprasad and Mueller 2005]. Another prototype from Chalmers University of Technology uses symbolic execution [Lundqvist and Stenström 1999b] for cache analysis.

The data provided in Table V might be imprecise, because the information can only be inferred from the publications instead of the tools in some cases. More comprehensive knowledge on existing WCET analysis tools can be found in [Wilhelm et al. 2008] and the reports for the WCET Tool Challenge in 2011 [Hanxleden et al. 2011], 2008 [Holsti et al. 2008] and 2006 [Gustafsson 2007].

7. FUTURE RESEARCH DIRECTIONS

WCET estimation is a key task in timing analysis of real-time systems. Since caches may significantly affect execution time, the quality of cache analysis determines the precision of the estimated WCET. This article surveys the main challenges and analysis techniques for vast cache architectures. For decades, the LRU replacement policy has been well studied. The most valuable asset is that a comprehensive understanding of cache behavior and cache analysis were established by the ingenious researchers in related communities. However, the use of existing techniques in real-life systems is still limited. Several future directions can be explored to bridge the gap.

—*Non-LRU cache analysis*

Although LRU is highly predictable, it is practically more important to analyze non-LRU replacement policies since they are actually adopted in real-life processors. Must, May and Persistence analyses needs to be established to fully characterize the cache behavior. Currently, the missing pieces are Persistence analysis for FIFO, Must and May analyses for MRU, Persistence and May analyses for PLRU. Furthermore, there are no techniques to analyze non-LRU data caches and multi-level caches, which are actually required to cover the whole cache hierarchy. For policies other than the abovementioned ones, similar analysis targets should be fulfilled. However, we still lack a systematic way to construct abstract analyses for new replacement policies.

—*Application of cache analysis in other domains*

So far the use of cache analysis has mostly been confined to WCET analysis. However, there is at least one more domain in which cache analysis can deliver valuable insights, namely security. *Side-channel attacks* recover secret inputs to programs from physical characteristics of the computation. Typical goals of such attacks are the recovery of cryptographic keys and private information about users. Characteristics that have been exploited for that purpose include execution time, cache behavior, memory and power consumption, and electromagnetic radiation. Doychev et al. have demonstrated that static cache analyses based on abstract interpretation can be used to derive guarantees on the amount of information leaked to an attacker [Doychev et al. 2013].

—*Design and analysis of timing-predictable embedded systems*

Preemption delay analysis and multi-core shared cache analysis have to consider the interactions among tasks running in parallel. It is commonly acknowledged that inter-core interference not only harms cache analysis, but also degrades overall system performance. Academia has gradually come to a consensus [Axer et al. 2014; Ungerer et al. 2010; Consortium 2011]: the solution to this problem should be to regulate both the hardware [Paolieri et al. 2009; Wilhelm et al. 2009] and the software [Pellizzoni et al. 2011; Falk and Kotthaus 2011; Maksoud and Reineke 2014] so that the system behaves in a timely predictable manner. The grand challenge is to obtain predictability without sacrificing the performance provided by future powerful processors. Cache analysis will provide valuable insights to characterize tasks so that good design decisions can be made in resource allocation and arbitration, such as cache partitioning and cache-aware scheduling.

One important step in this direction would be to understand how *timing compositionality* [Hahn et al. 2013] can be achieved. Due to complex interactions between caches and other microarchitectural components, such as branch predictors or out-of-order pipelines, provably sound WCET analyses can currently only be achieved by analyzing all of these components together in an *integrated* fashion. However, such an integrated approach is very unlikely to scale to multi-tasking systems or even to the parallel execution of multiple tasks on a multi-core processor. In these scenarios, to limit analysis complexity, interference costs are better analyzed separately and then taken into account during schedulability analysis. Timing compositionality has, however, not been formally proven for models of *any* modern microarchitecture, leaving much of the recent work unapplicable to real systems.

REFERENCES

Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. 1996. Cache Behavior Prediction by Abstract Interpretation. In *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings (Lecture Notes in Computer Science)*, Radhia Cousot and David A. Schmidt (Eds.), Vol. 1145. Springer, 52–66. DOI:http://dx.doi.org/10.1007/3-540-61739-6_33

Sebastian Altmeyer and Claire Burguière. 2009. A New Notion of Useful Cache Block to Improve the Bounds of Cache-Related Preemption Delay. In *21st Euromicro Conference on Real-Time Systems (ECRTS '09)*. 109–118.

Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. 2012. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems* 48, 5 (2012), 499–526.

Sebastian Altmeyer, Claire Maiza (Burguière), and Jan Reineke. 2010. Resilience Analysis: Tightening the CRPD bound for set-associative caches. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems*. 153–162.

Rajeev Alur and David L. Dill. 1994. A theory of timed automata. *Theoretical Computer Science* 126, 2 (1994), 183 – 235.

Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard Von Hanxleden, Reinhard Wilhelm, and Wang Yi. 2014. Building Timing Predictable Embedded Systems. *ACM Transactions on Embedded Computing Systems* 13, 4, Article 82 (2014), 37 pages.

Gogul Balakrishnan and Thomas Reps. 2004. Analyzing Memory Accesses in x86 Executables. In *Compiler Construction (Lecture Notes in Computer Science)*, Vol. 2985. 5–23.

Clément Ballabriga and Hugues Casse. 2008. Improving the First-Miss Computation in Set-Associative Instruction Caches. In *Proceedings of the 2008 Euromicro Conference on Real-Time Systems (ECRTS '08)*. 341–350.

Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. 2010. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Software Technologies for Embedded and Ubiquitous Systems*. Lecture Notes in Computer Science, Vol. 6399. 35–46.

Abhijeet Banerjee, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2013. Precise Micro-Architectural Modeling for WCET Analysis via AI and SAT. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*. 87–96.

Christoph Berg. 2006. PLRU Cache Domino Effects. In *Proceedings of the 6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*.

Claire Burguière, Jan Reineke, and Sebastian Altmeyer. 2009. Cache-Related Preemption Delay Computation for Set-Associative Caches - Pitfalls and Solutions. In *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*. 1–11.

Claire Burguière and Christine Rochange. 2005. A Contribution to Branch Prediction Modeling in WCET Analysis. In *Proceedings of 2005 Design, Automation and Test in Europe*. 612–617.

Josè V. Busquets-Mataix, Juan J. Serrano, Rafael Ors, Pedro Gil, and Andy Wellings. 1996. Adding Instruction Cache Effect to Schedulability Analysis of Preemptive Real-Time Systems. In *Proceedings of IEEE Real-Time Technology and Applications Symposium*. 204–212.

Giorgio C. Buttazzo. 2004. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA.

Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon, and Alvin R. Lebeck. 2001. Exact Analysis of the Cache Behavior of Nested Loops. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. 286–297.

Sudipta Chattopadhyay and Abhik Roychoudhury. 2009. Unified Cache Modeling for WCET Analysis and Layout Optimizations. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*. 47–56.

Sudipta Chattopadhyay and Abhik Roychoudhury. 2011. Scalable and Precise Refinement of Cache Timing Analysis via Model Checking. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*. 193–203.

Edmund M. Clarke, Orna Grumberg, and Doron Peled. 1999. *Model Checking*. MIT Press, Cambridge, MA, USA.

Philippe Clauss. 1996. Counting Solutions to Linear and Nonlinear Constraints Through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs. In *Proceedings of the 10th International Conference on Supercomputing*. 278–285.

Antoine Colin and Isabelle Puaut. 2000. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Real-Time Systems* 18, 2/3 (2000), 249–274.

PREDATOR Consortium. 2011. The PREDATOR project page. (2011). Retrieved 2014-08-10 from http://www.predator-project.eu/consortium.htm

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. 238–252.

Christoph Cullmann. 2013. Cache Persistence Analysis: Theory and Practice. *ACM Transactions on Embedded Computing Systems* 12, 1s, Article 40 (March 2013), 25 pages.

Andreas E. Dalsgaard, Mads Chr. Olesen, Martin Toft, René R. Hansen, and Kim G. Larsen. 2010a. METAMOC: Modular Execution Time Analysis using Model Checking. In *10th International Workshop on Worst-Case Execution Time Analysis*. 113–123.

Andreas E. Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. 2010b. METAMOC: Modular Execution Time Analysis using Model Checking. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, Vol. 15. 113–123.

Robert I. Davis and Alan Burns. 2011. A Survey of Hard Real-time Scheduling for Multiprocessor Systems. *Comput. Surveys* 43, 4, Article 35 (2011), 44 pages.

Goran Doychev, Dominik Feld, Boris Kpf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security*. http://embedded.cs.uni-saarland.de/publications/CacheAuditUSENIXSecurity13.pdf

Heiko Falk and Helena Kotthaus. 2011. WCET-driven Cache-aware Code Positioning. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 145–154.

Christian Ferdinand. 1997. *Cache Behavior Prediction for Real-Time Systems*. Ph.D. Dissertation. Saarland University, Saarbruecken, Germany. ISBN: 3-9307140-31-0.

Christian Ferdinand and Reinhard Wilhelm. 1999. Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems* 17, 2-3 (Dec. 1999), 131–181.

Somnath Ghosh, Margaret Martonosi, and Sharad Malik. 1999. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Transactions on Programming Langguages and Systems* 21, 4 (July 1999), 703–746.

Daniel Grund. 2011. *Static Cache Analysis for Real-Time Systems - LRU, FIFO, PLRU*. Ph.D. Dissertation. Saarland University, Saarbruecken, Germany. ISBN: 978-3-8442-1699-8.

Daniel Grund and Jan Reineke. 2009. Abstract Interpretation of FIFO Replacement. In *Proceedings of the 16th International Symposium on Static Analysis (SAS '09)*. 120–136.

Daniel Grund and Jan Reineke. 2010a. Precise and Efficient FIFO-Replacement Analysis Based on Static Phase Detection. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS)*. 155–164.

Daniel Grund and Jan Reineke. 2010b. Toward Precise PLRU Cache Analysis. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010) (OpenAccess Series in Informatics (OASIcs))*, Björn Lisper (Ed.), Vol. 15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 23–35. DOI: http://dx.doi.org/10.4230/OASIcs.WCET.2010.23 The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7.

Daniel Grund, Jan Reineke, and Gernot Gebhard. 2011. Branch target buffers: {WCET} analysis framework and timing predictability. *Journal of Systems Architecture* 57, 6 (2011), 625 – 637. DOI: http://dx.doi.org/10.1016/j.sysarc.2010.05.013 Design and Optimization for Embedded and Real-Time Computing Systems and Applications.

Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. 2012. WCET Analysis with MRU Caches: Challenging LRU for Predictability. In *Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium*. 55–64.

Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. 2009. Cache-aware Scheduling and Analysis for Multicores. In *Proceedings of the 7th ACM International Conference on Embedded Software*. 245–254.

Nan Guan, Xinping Yang, Mingsong Lv, and Wang Yi. 2013. FIFO Cache Analysis for WCET Estimation: A Quantitative Approach. In *Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE), 2013*. 296–301.

Jan Gustafsson. 2007. *WCET Challenge 2006 - Technical Report*. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-206/2007-1-SE. http://www.es.mdh.se/publications/1020-

Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. 2006. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*. 57–66.

Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. 2010. Towards WCET Analysis of Multicore Architectures Using UPPAAL. In *10th International Workshop on Worst-Case Execution Time Analysis*, Vol. 15. 101–112.

Sebastian Hahn and Daniel Grund. 2012. Relational Cache Analysis for Static Timing Analysis. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS '12)*. 102–111. DOI:http://dx.doi.org/10.1109/ECRTS.2012.14

Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. 2013. Towards Compositionality in Execution Time Analysis – Definition and Challenges. In *6th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*. http://embedded.cs.uni-saarland.de/publications/TowardsCompositionality.pdf

Reinhard von Hanxleden, Niklas Holsti, Björn Lisper, Erhard Ploedereder, Armelle Bonenfant, Hugues Cassé, Sven Bünte, Wolfgang Fellger, Sebastian Gepperth, Jan Gustafsson, Benedikt Huber, Nazrul Mohammad Islam, Daniel Kästner, Raimund Kirner, Laura Kovacs, Felix Krause, Marianne de Michiel, Mads Christian Olesen, Adrian Prantl, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Simon Wegener, Michael Zolda, and Jakob Zwirchmayr. 2011. WCET Tool Challenge 2011: Report. In *the 11th International Workshop on Worst-Case Execution Time (WCET) Analysis (WCET 2011)*.

Damien Hardy, Thomas Piquet, and Isabelle Puaut. 2009. Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*. 68–77.

Damien Hardy and Isabelle Puaut. 2008. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *2008 Real-Time Systems Symposium*. 456–466.

Damien Hardy and Isabelle Puaut. 2011. WCET Analysis of Instruction Cache Hierarchies. *Journal of Systems Architecture* 57, 7 (Aug. 2011), 677–694.

Reinhold Heckmann and Christian Ferdinand. 2014. Worst-Case Execution Time Prediction by Static Program Analysis. *The whitepaper of aiT* (2014). http://www.absint.com/aiT_WCET.pdf

Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. 2003. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *Proc. IEEE* 91, 7 (July 2003), 1038–1054.

John L. Hennessy and David A. Patterson. 2011. *Computer Architecture: A Quantitative Approach* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Heptane. 2013. The Heptane tool page. (2013). Retrieved 2014-08-10 from http://www.irisa.fr/alf/index.php?option=com_content&view=article&id=29&Itemid=0&lang=en

Niklas Holsti, Jan Gustafsson, Guillem Bernat, Clément Ballabriga, Armelle Bonenfant, Roman Bourgade, Hugues Cassé, Daniel Cordes, Albrecht Kadlec, Raimund Kirner, Jens Knoop, Paul Lokuciejewski, Nicholas Merriam, Marianne de Michiel, Adrian Prantl, Bernhard Rieder, Christine Rochange, Pascal Sainrat, and Markus Schordan. 2008. WCET 2008 – Report from the Tool Challenge 2008. In *the 8th International Workshop on Worst-Case Execution Time Analysis (WCET'08)*, Vol. 8.

Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. 2011. Scope-Aware Data Cache Analysis for WCET Estimation. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '11)*. 203–212.

Lei Ju, Samarjit Chakraborty, and Abhik Roychoudhury. 2007. Accounting for Cache-Related Preemption Delay in Dynamic Priority Schedulability Analysis. In *Design, Automation Test in Europe Conference Exhibition, 2007*. 1–6. DOI:http://dx.doi.org/10.1109/DATE.2007.364534

Gary A Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 194–206.

Sung-Kwan Kim, Sang-Lyul Min, and Rhan Ha. 1996. Efficient Worst Case Timing Analysis of Data Caching. In *Proceedings of 1996 IEEE Real-Time Technology and Applications Symposium*. 230–240.

Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. 1998. Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling. *IEEE Transactions on Computers* 47, 6 (6 1998), 700–713.

Benjamin Lesage, Damien Hardy, and Isabelle Puaut. 2009. WCET Analysis of Multi-Level Set-Associative Data Caches. In *Proceedings of the 9th Intl. Workshop on Worst-Case Execution Time Analysis*, Vol. 10.

Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. 2007. Chronos: A Timing Analyzer for Embedded Software. *Science of Computer Programming* 69, 1-3 (2007), 56–67.

Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. 2006. Modeling out-of-order processors for WCET analysis. *Real-Time Systems* 34, 3 (2006), 195–227.

Yau-Tsun Steven Li and Sharad Malik. 1995. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference (DAC '95)*. 456–461.

Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. 1996. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*. 254–263.

Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. 2012. Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. *Real-Time Systems* 48, 6 (2012), 638–680.

Fang Liu and Yan Solihin. 2010. Understanding the Behavior and Implications of Context Switch Misses. *ACM Transactions on Architecture and Code Optimization* 7, 4, Article 21 (2010), 28 pages.

Tiantian Liu, Yingchao Zhao, Minming Li, and Chun Jason Xue. 2010. Task Assignment with Cache Partitioning and Locking for WCET Minimization on MPSoC. In *Proceedings of the 39th International Conference on Parallel Processing*. 573–582.

Thomas Lundqvist and Per Stenström. 1999a. A Method to Improve the Estimated Worst-Case Performance of Data Caching. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*. 255–262.

Thomas Lundqvist and Per Stenström. 1999b. A Method to Improve the Estimated Worst-Case Performance of Data Caching. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*.

Will Lunniss, Sebastian Altmeyer, and Robert I. Davis. 2014. A Comparison between Fixed Priority and EDF Scheduling accounting for Cache Related Pre-emption Delays. *Leibniz Transactions on Embedded Systems* 1, 1 (2014), 01:1–01:24.

Will Lunniss, Robert I. Davis, Claire Maiza, and Sebastian Altmeyer. 2013. Integrating Cache Related Preemption Delay Analysis into EDF Scheduling. In *Proceedings of the IEEE 19th Real-Time and Embedded Technology and Applications Symposium*. 75–84.

Mingsong Lv, Nan Guan, Qingxu Deng, Ge Yu, and Wang Yi. 2011. McAiT: A Timing Analyzer for Multicore Real-time Software. In *Proceedings of the 9th International Conference on Automated Technology for Verification and Analysis*. 414–417.

Mohamed Abdel Maksoud and Jan Reineke. 2014. A Compiler Optimization to Increase the Efficiency of WCET Analysis. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems (RTNS '14)*. ACM, New York, NY, USA, Article 87, 10 pages. DOI:http://dx.doi.org/10.1145/2659787.2659825

Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. 2013. Real-time Cache Management Framework for Multi-core Architectures. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*. 45–54.

Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. 1998. Analysis of Loops. In *In Proceedings of the 7th International Conference on Compiler Construction*. Springer-Verlag, 80–94.

Frank Mueller. 1997. Timing Predictions for Multi-Level Caches. In *In ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*. 29–36.

Frank Mueller. 2000. Timing Analysis for Instruction Caches. *Real-Time Systems* 18, 2/3 (2000), 217–247.

Frank Mueller and David B. Whalley. 1995. Fast Instruction Cache Analysis via Static Cache Simulation. In *Proceedings of the 28th Annual Simulation Symposium*. 105–114.

Hemendra Singh Negi, Tulika Mitra, and Abhik Roychoudhury. 2003. Accurate Estimation of Cache-Related Preemption Delay. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 201–206.

Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. 2009. Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems. *Proceedings of the 36th Annual International Symposium on Computer Architecture* 37, 3 (2009), 57–68.

Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. 2011. A Predictable Execution Model for COTS-Based Embedded Systems. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. 269–279.

Harini Ramaprasad and Frank Mueller. 2005. Bounding Worst-Case Data Cache Behavior by Analytically Deriving Cache Reference Patterns. In *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*. 148–157.

Jan Reineke. 2008. *Caches in WCET Analysis: Predictability, Competitiveness, Sensitivity*. Ph.D. Dissertation. Saarland University, Saarbruecken, Germany. ISBN: 978-3-941071-69-8.

Jan Reineke, Sebastian Altmeyer, Daniel Grund, Sebastian Hahn, and Claire Maiza. 2014. Selfish-LRU: Preemption-Aware Caching for Predictability and Performance. In *Proceedings of the 20th Real-Time and Embedded Technology and Applications Symposium (RTAS'14)*.

Jan Reineke and Daniel Grund. 2008. Relative Competitive Analysis of Cache Replacement Policies. *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* 43, 7 (June 2008), 51–60.

Jan Reineke and Daniel Grund. 2013. Sensitivity of Cache Replacement Policies. *ACM Transactions on Embedded Computing Systems* 12, 1s, Article 42 (2013), 18 pages.

Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. 2007. Timing Predictability of Cache Replacement Policies. *Real-Time Systems* 37, 2 (Nov. 2007), 99–122.

Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. 2006. A Definition and Classification of Timing Anomalies. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis*.

Martin Schoeberl. 2008. A Java Processor Architecture for Embedded Real-time Systems. *Journal of Systems Architecture* 54, 1-2 (2008), 265–286.

Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. 2010. Worst-case Execution Time Analysis for a Java Processor. *Software - Practice and Experience* 40, 6 (2010), 507–542.

Rathijit Sen and Y. N. Srikant. 2007. WCET Estimation for Executables in the Presence of Data Caches. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*. 203–212.

Tyler Sondag and Hridesh Rajan. 2010. A More Precise Abstract Domain for Multi-level Caches for Tighter WCET Analysis. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS)*. 395–404.

Jan Staschulat and Rolf Ernst. 2006. Worst case timing analysis of input dependent data cache behavior. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*. 227–236.

Jan Staschulat and Rolf Ernst. 2007. Scalable Precision Cache Analysis for Real-Time Software. *ACM Transactions on Embedded Computing Systems* 6, 4, Article 25 (2007). http://doi.acm.org/10.1145/1274858.1274863

Jan Staschulat, Simon Schliecker, and Rolf Ernst. 2005. Scheduling Analysis of real-Time Systems with Precise Modeling of Cache Related Preemption Delay. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS 2005)*. 41–48.

Vivy Suhendra and Tulika Mitra. 2008. Exploring Locking & Partitioning for Predictable Shared Caches on Multi-cores. In *Proceedings of the 45th Annual Design Automation Conference (DAC '08)*. 300–303.

SWEET. 2012. The SWEET tool page. (2012). Retrieved 2014-08-10 from http://www.mrtc.mdh.se/projects/wcet/sweet/online/content/index.php

SymTA/P. 2010. The SymTA/P project page. (2010). Retrieved 2014-08-10 from https://www.ida.ing.tu-bs.de/en/research/projects/symtap/

Yudong Tan and Vincent Mooney. 2007. Timing Analysis for Preemptive Multitasking Real-time Systems with Caches. *ACM Transactions on Embedded Computing Systems* 6, 1, Article 7 (2007).

Yudong Tan and Vincent J. Mooney. 2004. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. In *Workshop on Software and Compilers for Embedded Systems*. 182–199. http://codesign.ece.gatech.edu/publications/ydtan/paper/sc04_Yudong_Tan.pdf

Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. 2000. Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *Real-Time Systems* 18, 2/3 (May 2000), 157–179.

Hiroyuki Tomiyama and Nikil D. Dutt. 2000. Program Path Analysis to Bound Cache-related Preemption Delay in Preemptive Real-Time Systems. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign*. 67–71.

Theo Ungerer, Francisco Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quinones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Cassé, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Floria Kluge, Stefan Metzlaff, and Jorg Mische. 2010. Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability. *IEEE Micro* 30, 5 (2010), 66–75.

Xavier Vera, Björn Lisper, and Jingling Xue. 2003. Data caches in multitasking hard real-time systems. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*. 154–165.

Xavier Vera and Jingling Xue. 2002. Let's Study Whole-Program Cache Behaviour Analytically. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA '02)*. 175–.

Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. 2013. Making Shared Caches More Predictable on Multicore Platforms. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*. 157–167.

Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. 2008. Measurement-Based Timing Analysis. 17 (2008), 430–444.

Randall T. White, Frank Mueller, Chris Healy, David Whalley, and Marion Harmon. 1999. Timing Analysis for Data and Wrap-Around Fill Caches. *Real-Time Systems* 17, 2-3 (Dec. 1999), 209–233.

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems* 7, 3, Article 36 (2008), 53 pages.

Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. 2009. Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 7 (2009), 966–978.

Lan Wu and Wei Zhang. 2012. A Model Checking Based Approach to Bounding Worst-Case Execution Time for Multicore Processors. *ACM Transactions on Embedded Computing Systems* 11, S2, Article 56 (2012), 19 pages.

Jun Yan and Wei Zhang. 2008. WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*. 80–89.

Patrick Meumeu Yomsi and Yves Sorel. 2007. Extending Rate Monotonic Analysis with Exact Cost of Preemptions for Hard Real-Time Systems. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*. 280–290.

Wei Zhang and Jun Yan. 2009. Accurately Estimating Worst-Case Execution Time for Multi-core Processors with Shared Direct-Mapped Instruction Caches. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. 455–463.

Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. 2009. Towards Practical Page Coloring-based Multicore Cache Management. In *Proceedings of the 4th ACM European Conference on Computer Systems*.

Received xxxx; revised xxxx; accepted xxxx