Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches

Damien Hardy Thomas Piquet Isabelle Puaut
Université Européenne de Bretagne / IRISA/INRIA, Rennes, France

Abstract

Multi-core chips have been increasingly adopted by the microprocessor industry. For real-time systems to exploit multi-core architectures, it is required to obtain both tight and safe estimates of worst-case execution times (WCETs). Estimating WCETs for multi-core platforms is very challenging because of the possible interferences between cores due to shared hardware resources such as shared caches, memory bus, etc.

This paper proposes a compile-time approach to reduce shared instruction cache interferences between cores to tighten WCET estimations. Unlike [28], which accounts for all possible conflicts caused by tasks running on the other cores when estimating the WCET of a task, our approach drastically reduces the amount of inter-core interferences. This is done by controlling the contents of the shared instruction cache(s), by caching only blocks statically known as reused. Experimental results demonstrate the practicality of our approach.

1 Introduction

In real-time systems it is crucial to prove that the execution of a task meets its deadline in all execution situations, including the worst-case. This proof needs an estimation of the worst-case execution time (WCET) of any task in the system. WCET estimates have to be safe, i.e. larger than or equal to any possible execution time. Moreover, they have to be tight, i.e. as close as possible to the actual worst-case execution time, to correctly dimension the resources required by the system. WCET estimation methods have to account for all possible flows in a program and determine the longest execution path (so-called *high-level analysis*). They also have to account for the hardware the program is running on, through a *low-level analysis*. A number of static analysis methods have been designed in the last two decades at both levels, mainly for uni-core processors [26].

Multi-cores have been increasingly adopted for both desktop and now embedded applications. In most multi-core architectures, the different cores have private caches and also shared on-chip caches useful for code and data sharing between cores. However, for multi-core architectures to be used in hard real-time systems, it is required to obtain both tight and safe estimates of WCETs. This is a very challenging task because of the possible interferences between cores due to shared hardware resources such as shared caches. WCET estimation for multi-core platforms has been the subject of very few studies. We present in this paper a new WCET estimation method for multi-core platforms with shared instruction caches. The proposed method provides tight WCET estimates, through a control of the contents of the shared instruction cache(s), more precisely by caching only the blocks statically known to be reused and bypassing from shared caches the other blocks.

Related work. Many WCET estimation methods have been designed in the last two decades (see [26] for a survey). The mostly used static WCET computation technique, called IPET (Implicit Path Enumeration Technique) such as [12] estimates the WCET through the resolution of an Integer Linear Programming (ILP) problem constraining the execution frequencies of the program basic blocks.

Static WCET estimation methods need a low-level analysis phase to determine the worst-case timing behavior of the microarchitectural components: pipelines and out-of-order execution [4, 10], branch predictors [1] and caches. Regarding cache memories on uni-core architectures, two main classes of approaches have been proposed: *static cache simulation* [14, 15], based on dataflow analysis, and the methods described in [6, 24, 7], based on abstract interpretation. Both classes of methods provide a classification of the outcome of every memory reference in the worst-case execution scenario (e.g. *always-hit*, *always-miss*, *first-miss*, etc.). These methods were originally designed for code only, and for direct-mapped or set-associative caches with a Least Recently Used (LRU) replacement policy. They have been later extended to other replacement policies [9], data and unified caches [25], and caches hierarchies [8]. Related techniques estimate the worst-case delay to reload the cache contents after a preemption by a higher priority task [16, 22].

Very few studies have considered WCET analysis for multi-core platforms so far. The method described in [28] estimates the WCET in presence of shared caches on multi-cores by analysing *inter-thread cache conflicts*. They assume a dual-core architecture with a private L1 instruction cache and a shared L2 instruction cache. The method aims at analysing the WCET of a real-time task (*rtt*), running on one core, during the execution of a non real-time task (*nrtt*) running on the other core. The method detects conflicts in the shared L2 in-

struction cache, and integrates such conflicts by changing the cache classification of the concerned program blocks of the *rtt*. The classification of a program block in the *rtt* located in a loop, which would have been classified as a *hit* if the task was executed alone, is changed to *always-except-one* if none of the conflicting blocks in the *nrtt* is located in a loop. This method might underestimate the WCET of the *rtt* when different cache blocks of the *nrtt* evict a *rtt* cache block several times during the execution of the loop, or if the *nrtt* executes several times while the *rtt* executes. More generally, the method described in [28] is expected to lack scalability with respect to the task size and number of tasks, because every conflict with every other task is considered. Our proposed method, like [28] identifies all inter-task interferences due to cache sharing for the sake of safety. However, it is complemented by a *bypass* method of the shared instruction cache(s), caching in the shared cache(s) only the blocks statically known to be reused, allowing to drastically reduce the amount of inter-core interferences.

A very different approach for multi-cores with shared instruction caches is proposed in [23] and is based on the combined use of cache *locking*, i.e. temporarily disabling cache replacement, and *partitioning*, i.e. partitioning the cache among the tasks or cores. The objective of such a joint use of locking and partitioning is to completely avoid intra-task and inter-task conflicts, which then do not need to be analysed. Different combinations of partitioning (per-core/per-task) and locking (static/dynamic) are experimented. With partitioning approaches, interferences caused by shared caches are avoided, thus having a positive impact on the WCET of every task; on the other hand, partitioning comes at the cost of a lower volume of cache available per task/core, having a negative impact on the WCET. In contrast to [23], our approach does neither lock nor partition the shared instruction cache(s). Experiments would be required to assess the respective merits of [23] compared with our approach.

Our approach also has some links with the uni-core approach described in [11], designed for single-level data caches. In [11] accesses to data are classified as either *hard-to-predict* or *easy-to-predict* at compile-time, and only *easy-to-predict* data are cached in the L1 data cache. Our approach also implements selective caching based on information computed at compile-time, the relevant information being the reuse of instructions in the shared instruction cache(s) in our case.

Related cache bypass techniques approaches have already been proposed in the computer architecture community [5, 3, 18], and some architectures have an ISA support for bypass (HP PA-RISC, Itanium). These studies show the presence of cache blocks that are not reused and therefore generate wasteful conflicts (e.g. 33% in average in the L2 cache on the SPEC CPU benchmarks [18]). In [18], both the detection and the bypass of single-usage blocks are dynamic and hardware implemented. In [5], they suggest a static (using profiling) and a dynamic solution to select the cache blocks to bypass. All these related studies aim at reducing the average-case execution time, while we focus on a reduction of worst-case execution time estimates. Furthermore, in our case, detection of single-usage blocks is done at compile-time.

Contributions. The first contribution in this paper is a safe WCET estimation method for multi-core architectures with shared instruction cache(s). Our method is more general than [28] in the sense that it supports multiple levels of shared caches, set-associative caches and an arbitrary number of real-time tasks and cores competing for the shared caches.

Our second and main contribution is the static identification of single-usage (not reused) program blocks in shared instruction caches, thanks to static analysis of the program code. This enables a compiler-directed bypass scheme, which, from the static knowledge of single-usage blocks, allows a drastic reduction of inter-task and intra-task interferences, and thus a tighter WCET estimate.

Paper outline. The remainder of the paper is organized as follows. Section 2 presents the assumptions our analysis is based on, regarding the target architecture and task scheduling. Section 3 presents a safe WCET estimation method for multi-core architectures with shared instruction caches, that considers all interferences for the shared instruction caches. Section 4 then presents the main contribution of the paper, a bypass technique to decrease interferences due to instruction cache sharing, and thus to tighten the WCET estimate. Experimental results are given in Section 5. Finally Section 6, conclude this paper and give directions for future work.

2 Assumptions

A multi-core architecture is assumed. Each core has a private first-level (L1) instruction cache, followed by instruction cache levels with at least one shared cache. The caches are set-associative. Each level of the cache hierarchy is non-inclusive:

- A piece of information is searched for in the cache of level ℓ if and only if a cache miss occurred when searching it in the cache of level $\ell-1$. Cache of level 1 is always accessed.
- Except if the bypass mechanism presented in Section 4 is used, every time a cache miss occurs at cache level ℓ, the entire cache block containing the missing piece of information is always loaded into the cache of level ℓ.
- There are no actions on the cache contents (i.e. invalidations, lookups/modifications) other than the ones mentioned above.

Our study concentrates on instruction caches; it is assumed that the shared caches do not contain data. This study can be seen as a first step towards a general solution for shared caches. It can also push the use of separate shared instruction and data caches instead of unified ones[1].

Our method assumes a LRU (Least Recently Used) cache replacement policy. The access time variability to main memory and shared caches, due to bus contention, is supposed to

[1]Unified caches could be partitioned at boot time for instance in a A-way instruction cache and a B-way data cache.

be bounded and known, by using for instance *Time Division Multiple Access (TDMA)* like in [21] or other predictable bus arbitration policies [17].

Figure 1 illustrates two different supported architectures.

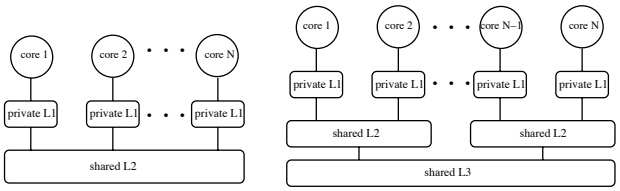


Figure 1. Two examples of supported architectures

Regarding scheduling, it is assumed that a job does not migrate between cores at run-time. Migrations are allowed between job instances only. No further assumption is made on task scheduling, implying that any part of an interfering task may be executed simultaneously with the analysed task and may thus pollute the shared cache(s). This assumption was made in a first approach to keep WCET estimation and schedulability independent activities, as traditionally done when temporally validating real-time software. We do not attempt to explore joint WCET estimation and scheduling, which is left for future work. Tasks are independent (i.e. do not synchronize with each other), but might share code, such as libraries (see paragraph 3.2).

The focus in this paper is to estimate the WCET of a hard-real time task running on a core, in isolation from the tasks running on the same core, but suffering indirect interferences because of cache sharing from tasks running on the other cores. The computation of cache-related preemption delay due to intra-core interferences is considered out of the scope of this paper.

3 WCET Analysis on multi-cores with multiple levels of instruction caches

In this section, we describe a safe, albeit pessimistic, WCET estimation method in presence of shared instruction caches. Section 3.1 first presents our base WCET estimation method for multi-level caches on uni-core processors, initially presented in [8]. Section 3.2 then extends the base method to cope with interfering tasks running on the other cores. In this section, no attempt is made to reduce the volume of such inter-task interferences, which will be the subject of Section 4.

3.1 Static multi-level cache analysis for uni-core processors [8]

The cache analysis is applied successively on each level of the cache hierarchy, from the first cache level to the main memory. The analysis is contextual in the sense that it is applied for every call context of functions (functions are virtually inlined). The references considered by the analysis of cache level ℓ depend on the outcome of the analysis of cache level $\ell - 1$ to consider the filtering of memory accesses between cache levels, as depicted in Figure 2 and detailed below.

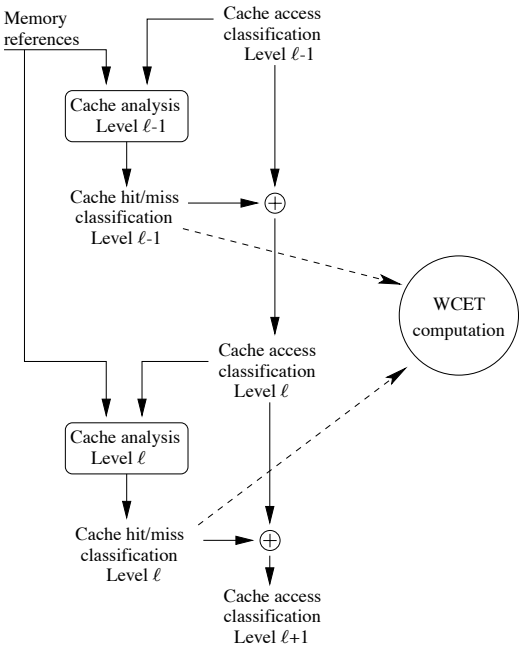


Figure 2. Multi-level cache analysis on a uni-core processor

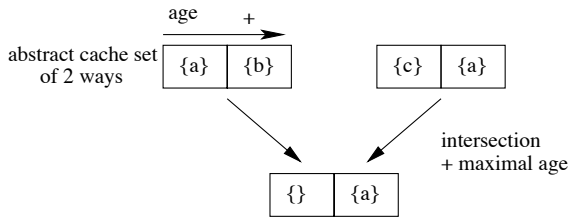
The outcome of the static cache analysis for every cache level ℓ is a *Cache Hit/Miss Classification (CHMC)* for each reference, determining the worst-case behavior of the reference with respect to cache level ℓ:

- *always-miss* (AM): the reference will always result in a cache miss,
- *always-hit* (AH): the reference will always result in a cache hit,
- *first-miss* (FM): the reference could neither be classified as hit nor as miss the first time it occurs but will result in cache hit afterwards,
- *not-classified* (NC): in all other cases.

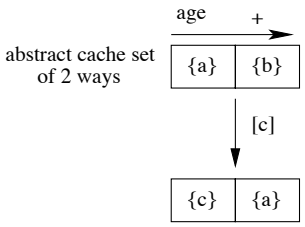
At every level ℓ, a *Cache Access Classification (CAC)* determines if an access may occur or not at level ℓ, and thus should be considered by the static cache analysis of that level. There is a CAC, noted $CAC_{r,\ell,c}$ for every reference r, cache level ℓ, and call context c[2]. The CAC defines four categories for each reference, cache level, and call context:

- A (Always): the access always occurs at the cache level.
- N (Never): the access never occurs at the cache level.
- $U - N$ (Uncertain-Never): the access could occur or not the first time but next accesses will never occur at the

[2]The call context c will be omitted from the formulas when the concept of call context is not relevant.



a. Join function of Must analysis



b. Update function of Must analysis

Figure 3. $Join$ and $Update$ functions for the Must analysis with LRU replacement

cache level. This category, subset of the U category described below, was added to our original cache analysis published in [8], because it allows to know that the access will occur at most once, which permits a more precise identification of single-usage blocks.
- U (Uncertain) when the access cannot be classified in the three above categories.

The cache analysis at every cache level is based on a state-of-the-art single-level cache analysis [24], based on abstract interpretation. The method is based on three separate fixpoint analyses applied on the program control flow graph, for every call context:

- a *Must* analysis determines if a memory block is always present in the cache at a given point: if so, the block is classified *always-hit (AH)*;
- a *Persistence* analysis determines if a memory block will not be evicted after it has been first loaded; the classification of such blocks is *first-miss (FM)*.
- a *May* analysis determines if a memory block may be in the cache at a given point: if not, the block is classified *always-miss (AM)*. Otherwise, if neither detected as always present by the *Must* analysis nor as persistent by the *Persistence* analysis, the block is classified *not classified (NC)*;

Abstract cache states (ACS) are computed for every basic block according to the semantics of the analysis (*Must*/*May*/*Persistence*) and the cache replacement policy by using functions ($Update$ and $Join$) in the abstract domain. $Update$ models the impact on the ACS of every reference inside a basic block; $Join$ merges two ACS at convergence points in the control flow graph (e.g. at the end of conditional constructs).

Figure 3 gives an example of an ACS of a 2-way set-associative cache with LRU replacement policy on a *Must* analysis (only one cache set is depicted). An *age* is associated to every cache block of a set. The smaller the block age the more recent the access to the block. For the *Must* analysis, each memory block is represented only once in the ACS, with its maximum age. It means that its actual age at run-time will always be lower than or equal to its age in the ACS.

At every cache level ℓ, the three analyses (*Must, May, Persistence*) consider all references r guaranteed to occur at level ℓ ($CAC_{r,\ell} = A$). References with $CAC_{r,\ell} = N$ are not analysed. Regarding uncertain references ($CAC_{r,\ell} = U$ or $CAC_{r,\ell} = U - N$), for the sake of safety, the ACS is obtained by exploring the two possibilities ($CAC_{r,\ell} = A$ and $CAC_{r,\ell} = N$) and merging the results using the $Join$ function. For all references r, $CAC_{r,1} = A$, meaning that the L1 cache is always accessed.

The CAC of a reference r for a cache level ℓ depends on CHMC of r at level $\ell - 1$ and the CAC of r at level $\ell - 1$ (see Figure 2). Table 1 shows all the possible cases of computation of $CAC_{r,\ell}$ from $CHMC_{r,\ell-1}$ and $CAC_{r,\ell-1}$.

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

Table 1. Cache access classification for level ℓ ($CAC_{r,\ell}$)

The CHMC of reference r is then used to compute the cache contribution to the WCET of that reference, which can be included in well-known WCET computation methods [12, 20].

3.2 Static multi-level cache analysis for multi-cores

Compared with its execution in isolation, the execution of a task on a multi-core architecture with shared cache(s) may introduce some extra misses in the shared cache resulting from the interfering tasks. In term of static analysis, it means that some accesses previously classified as *always-hit* (or *first-miss*) using a uni-core cache analysis (without considering the interfering tasks running on the other cores) may have to be changed into *first-miss* or *not classified*.

Compared to the uni-core analysis described in paragraph 3.1, considering inter-core interferences for shared cache level(s) requires to change the static cache analysis for the shared cache level(s), keeping intact the analysis of the private cache level(s). As illustrated in Figure 4, the analysis of a shared cache level ℓ estimates the worst-case number of conflicts per cache set due to tasks running on the other cores and then computes a cache classification to account for these con-

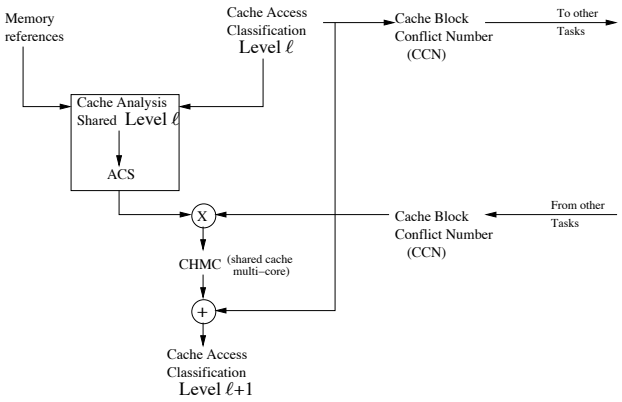


Figure 4. Cache analysis on a multi-core processor for a shared cache level ℓ

flicts. These two steps are detailed in the following paragraphs, before dealing with code sharing.

Note that the cache analysis method presented in this paragraph can be seen as a safe extension of the method presented in [28] to set-associative caches, multiple levels of shared caches and an arbitrary number of tasks/cores competing for the shared caches.

Estimation of interferences. For each shared cache level ℓ, the static cache analysis applied to each interfering task, provides information about which references may occur at level ℓ through the *CAC*. Any reference which may occur at level ℓ ($CAC_{r,\ell} \neq N$) has to be considered as interfering, regardless of the time when this access may occur to stay independent from task scheduling.

Thus, for each shared cache level ℓ, we compute a safe upper bound on the number of interfering blocks for each cache set. This number for a set s is named hereafter *cache block conflict number, CCN(s)* and is defined as follows:

$$\sum_{t}^{IT} \mid \{cb \in t, (\exists r \in cb, CAC_{r,\ell} \neq N) \wedge mapped(cb, s)\} \mid$$

where IT is the set of interfering tasks, cb is a cache block, r is a reference within cb and $mapped(cb, s)$ is a boolean function which returns true if cb is mapped to cache set s and false otherwise.

Accounting for inter-core interferences in cache classification. The number of conflicts per set $CCN(s)$ is used, together with the Abstract Cache States (ACS) provided by the cache analyses (*Must* and *Persistence*), to determine a new CHMC accounting for inter-task interferences.

The ACS produced by the *Must* analysis keeps the oldest age of a cache block in the cache set. Accounting for interfering tasks implies that in the worst-case this age has to be increased by $CCN(s)$, with s the cache set. If this corrected age is still less or equal than the cache associativity then the block will be ensured to be in the cache, otherwise the cache block

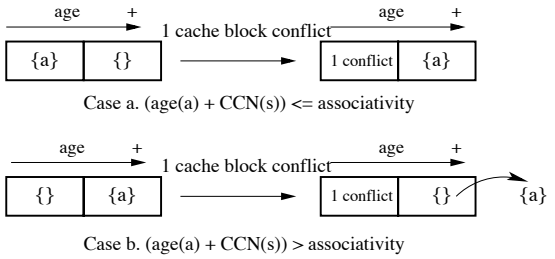


Figure 5. Accounting for inter-core interferences

is considered absent from the *Must* ACS. Figure 5 illustrates both cases. The same procedure is applied to the *Persistence* analysis. Conversely for the *May* analysis, which determines the cache blocks which may be in the cache, no modification is required.

Because of inter-core interferences, the CHMC of a reference on a multi-core platform may be more pessimistic than its equivalent in the uni-core case. Thus, indirectly, the CAC of the next cache level might also differ. For our approach to be safe for multiple levels of shared caches, we analyse for each task the shared cache levels in sequence and adjust the resulting CACs before analysing the next shared cache level.

Code sharing. To take into account the effect of code sharing between tasks due to shared libraries, we divide $CCN(s)$ into $CCN_{private}(s)$ and $CCN_{shared}(s, cb)$ (interferences caused by shared blocks might not always be destructive). The ranges of addresses of the shared code are assumed to be given as a parameter of the analysis. $CCN_{private}(s)$ represents the number of conflicts per set due to the private code of interfering tasks, and is computed as before without considering shared code addresses. Conversely, $CCN_{shared}(s, cb)$ represents the number of conflicts for a cache block cb mapped to cache set s caused by shared code. $CCN_{shared}(s, cb)$ is computed in two steps.

- The first step determines the set $Shared(s)$ of shared interfering cache blocks mapped to set s. Because of code sharing, each block belonging to a shared library and used by an interfering task has to be considered only once. Thus, $Shared(s)$ is defined as the union, for all interfering tasks, of the set of used shared cache blocks mapped to set s ($CAC_{r,\ell} \neq N$).
- The second step to compute $CCN_{shared}(s, cb)$ stems from the fact that the analysed task may also use some block in $Shared(s)$. Regarding *Must* analysis, at every program point, a shared block is considered as conflicting with block cb present in ACS_{Must} only if its age in ACS_{Must} is strictly higher than *age(cb)* or absent from the ACS_{Must}:

$$CCN_{shared}(s, cb) = \\ \mid Shared(s) \setminus \{b \in ACS_{Must}, age(b) \leq age(cb)\} \mid$$

Similarly, regarding *Persistence* analysis:

$$CCN_{shared}(s, cb) = \\ | \ Shared(s) \setminus \{b \in ACS_{Persistence}, age(b) \leq age(cb)\} \ |$$

Finally, for both the *Must* and *Persistence* analysis, the formula which determines if a cache block *cb* is evicted is:

$$age(cb)+CCN_{private}(s)+CCN_{shared}(s, cb) > associativity$$

Similarly to a system without code sharing, the *May* analysis needs not be modified.

4 Interference reduction using bypass of static single-usage blocks

Upon a cache miss, the conventional operation mode of a cache hierarchy is to retrieve the missing block from lower levels and to store them into all upper hierarchy levels. However, it is difficult to assert that storing the block into intermediate levels will be really useful. In some cases, a block stored in the cache after a miss may not be accessed again before its eviction. Such blocks, named single-usage blocks, contribute to the well known cache pollution phenomenon [18].

Static cache analysis methods have the ability to estimate single usage blocks at compile-time. The main contribution of this paper is to estimate such *static single usage* (SSU) blocks, and force the bypass of such blocks from the shared cache(s), in order to reduce pollution in shared caches and thus to tighten the WCET estimates.

Figure 6 illustrates the modifications in the WCET estimation procedure of Section 3.2 to identify SSU blocks and to account for the bypass of such blocks. These modifications, concerning shared caches levels only, are detailed below.

4.1 Identification of Static Single Usage (SSU) blocks

For a given shared cache level ℓ, a *static multiple usage block* is defined as a block statically known to be accessed multiple times and still present in the shared cache when reused, in at least one execution context. Any other block is termed *static single usage (SSU)* block.

SSU blocks are estimated for every task taken in isolation, with no specific treatment for shared code, using the CHMC and CAC (upper part of Figure 6). More formally, we define the SSU identification function, which returns *true* if the analysed cache block is a SSU block, as follows:

$$\bigwedge_{c \in contexts} \ \bigwedge_{r \in cache\ block} f_{SSU}(CAC_{r,\ell,c}, CHMC_{r,\ell,c})$$

with f_{SSU} defined in Table 2.

Function f_{SSU} returns true if a reference in a given context is not guaranteed to be reused, and false otherwise. Blocks with $CAC_{r,\ell,c} = U - N$ and $CHMC_{r,\ell,c} = FM$ are classified as SSU blocks because there are known to be accessed only once, by definition of the U-N CAC category.

$CAC_{r,\ell,c}$ \ $CHMC_{r,\ell,c}$	AM	AH	FM	NC
A	true	false	false	true
N	true	true	true	true
U-N	true	false	true	true
U	true	false	false	true

Table 2. $f_{SSU}(CAC_{r,\ell,c}, CHMC_{r,\ell,c})$ for level ℓ

4.2 Static cache analysis on multi-cores with bypass of SSU blocks

The lower part of Figure 6 shows the integration of SSU blocks in the shared cache analysis. SSU blocks r are marked as bypassed ($CAC_{r,\ell} = BP$). During the cache analysis of every shared cache level ℓ, accesses to BP blocks are semantically equivalent to never accessed blocks (whose $CAC_{r,\ell} = N$). Conflict number $CCN(s)$ is computed like in Section 3.2 except that blocks whose $CAC = BP$ are not considered as interfering. The $CHMC$ of BP accesses is set to *AM*. No modification of the cache analysis internals is required.

To take into account the bypass information when analysing multiple levels of shared caches, when analysing a shared cache level ℓ a safe CAC has to be propagated to the next cache level. Since blocks marked as BP only impact the current cache level ℓ, the original CAC (before its replacement by BP) is propagated to cache level $\ell + 1$.

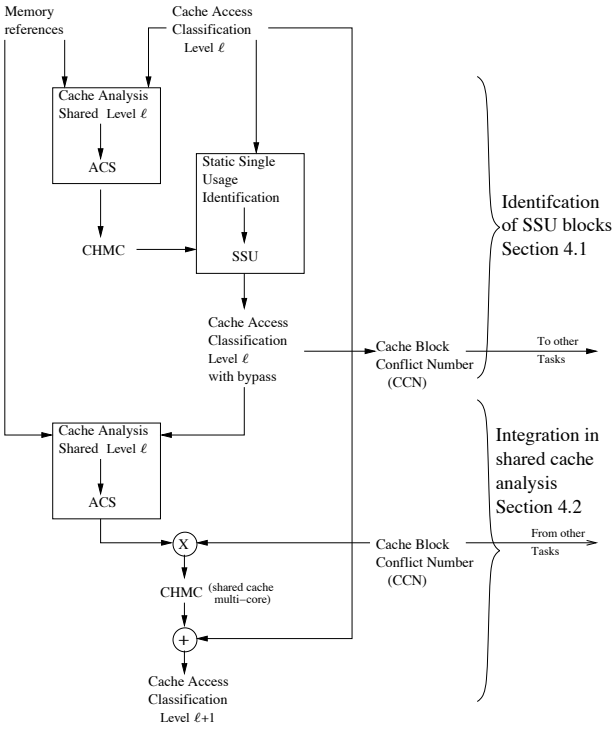


Figure 6. Cache analysis on a multi-core processor using bypass, for a shared cache level ℓ

4.3 Implementation

A straightforward way to implement our bypass approach is to use the scheme described in [19] where instructions have a dedicated bit to control their cacheability (1 bit per block). After the estimation of SSU blocks, this bit can be set at compile-time without any modification of the code memory layout. For multiple levels of shared caches, this solution requires n bits where n represents the number of shared cache levels (two bits for standard architectures). Other more complex implementations suggested in [13] propose distinguishing cached instructions from uncached instructions by addresses, or to dynamically turn on/off instruction caching. Nevertheless, these implementations need more compiler support.

5 Experimental results

In this section, the benefits of using the bypass mechanism proposed in Section 4 to reduce inter-task and intra-task interferences are evaluated. We first describe the experimental conditions (§ 5.1) and then we give and analyse experimental results for a 2-level cache hierarchy with a shared L2 instruction cache (§ 5.2).

Two performance metrics are used to evaluate our proposal. The first one is the hit ratio along the worst-case execution path in the L1 and L2 cache (noted HR_{L1} and HR_{L2}). The worst-case execution path (WCEP) used to compute HR_{L1} and HR_{L2} is evaluated from the frequency of basic blocks returned by the Implicit Path Enumeration Technique (IPET) WCET computation. The second metric is the number of cycles per instruction due to L2 miss (NCPI[3]) along the WCEP. This second metric is complementary to the first one and quantifies the impact in cycles of the proposed bypass scheme. No comparisons with measured values are given, because generating the worst-case interference for the shared L2 cache is extremely difficult to achieve due to the impact of the tasks timing and scheduling on the actual interferences for the shared L2 cache [27].

5.1 Experimental setup

Cache hierarchy. The results are obtained on a 2-level cache hierarchy composed of a private 4-way L1 cache of 1KB with a cache block size of 32B and a shared 8-way L2 cache of 4KB with a cache block size of 32B. Cache sizes are small compared to usual cache sizes in multi-core architectures. However no public real-time benchmarks with a large cache footprint are available to experiment our proposal. As a consequence, we have selected quite small commonly used real-time benchmarks and adjusted cache sizes such that the benchmarks do not fit entirely in the caches. All caches are implementing a LRU replacement policy. Latencies of 1 cycle (respectively 10 and 100 cycles) are assumed for the L1 cache (respectively the L2 cache and the main memory).

[3] $NCPI = \frac{\#L2\ miss * Memory\ latency}{\#instruction\ along\ the\ WCEP}$

Cache analysis and WCET estimation. The experiments were conducted on MIPS R2000/R3000 binary code compiled with gcc 4.1 with no optimization and with the default linker memory layout. The WCETs of tasks are computed by the Heptane timing analyzer [2], more precisely its Implicit Path Enumeration Technique (IPET). The analysis is context sensitive (functions are analysed in each different calling context). To separate the effect of the caches from those of the other parts of the processor micro-architecture, WCET estimation only takes into account the contribution of instruction caches to the WCET. The effects of other architectural features are not considered. The cache classification *not-classified* is assumed to have the same worst-case behavior as *always-miss* during the WCET computation in our experiments. The cache analysis starts with an empty cache state. For space consideration, WCET computation is not detailed here, interested readers are referred to [8].

Benchmarks. The experiments were conducted on eight benchmarks (see Table 3 for the applications characteristics). All benchmarks are maintained by Mälardalen WCET research group[4].

Name	Description	Code size (bytes)
crc	Cyclic redundancy check computation	1432
qurt	Root computation of quadratic equations	1928
lms	LMS adaptive signal enhancement	2828
fdct	Fast Discrete Cosine Transform	3468
fft	Fast Fourier Transform	3536
minver	Inversion of floating point 3x3 matrix	4408
adpcm	Adaptive pulse code modulation algorithm	7740
statemate	Automatically generated code by STARC (STAtechart Real-time-Code generator)	8900

Table 3. Benchmark characteristics

5.2 Results for a multi-core architecture with a shared L2 instruction cache

Impact of L2 bypass in a system without inter-core interference. An interesting observed side effect of the proposed bypass approach is that it allows decreasing the WCET estimate of a task without any concurrent task competing for the L2 cache, compared with the WCET estimate of the same task without any bypass. Indeed, all non-reused program blocks in a shared cache level are bypassed. As a consequence, this can avoid *intra-task* conflicts for shared cache blocks.

This phenomenon is quantified in Table 4, by considering a task executing on one core without any interfering task running on the other cores competing for the shared L2 cache.

For every task (column 1), we give the estimated hit ratio in the L1 cache HR_{L1} (column 2). Columns 3 (resp. 4) consider system without L2 bypass (resp. with L2 bypass as described

[4] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

in Section 4). Each of those column gives HR_{L2} and the NCPI. Finally, column 5 gives the percentage of bypassed blocks in the L2 cache ($\frac{\#bypassed\ cache\ blocks}{\#cache\ blocks}$). The higher HR_{L1}, the lower the number of references to the shared L2 cache. The lower the NCPI, the lower the impact of L2 cache misses on the WCET. The percentage of bypassed blocks given in column 5 indicates how much pollution is avoided in the L2 cache. The higher the bypass ratio, the higher the expected reduction of inter-core interference.

Bench.	HR_{L1}	L2 no bypass HR_{L2} (NCPI)	L2 bypass HR_{L2} (NCPI)	bypass ratio
minver	93.99%	39.76% (3.62)	39.76% (3.62)	94.92%
adpcm	89.74%	33.60% (6.81)	33.96% (6.77)	88.02%
fdct	87.25%	84.03% (2.04)	84.03% (2.04)	5.5%
statemate	83.40%	0.72% (16.49)	1.21% (16.40)	98,92%
fft	88.76%	1.97% (11.02)	12.50% (9.83)	92.79%
crc	93.10%	98.97% (0.07)	98.97% (0.07)	88.89%
lms	87.24%	0.61% (12.68)	0.61% (12.68)	94.38%
qurt	93.57%	12.56% (5.62)	12.56% (5.62)	98.36%

Table 4. Impact of L2 bypass on WCET estimation with no interference for the L2 cache.

As expected, our bypass scheme provides values of HR_{L2} always larger than or equal to when no bypass mechanism is used. For three tasks (*statemate, adpcm, fft*), HR_{L2} is strictly larger than when not using L2 bypass. In these cases, not storing SSU blocks in the L2 instruction cache allows detecting them as reused blocks which where previously classified as misses (because they were in conflict with a SSU block). In the best case (*fft*), HR_{L2} is multiplied by a factor of 6, which significantly reduces the task WCET estimate. In terms of number of cycles per instruction due to L2 misses, for *fft* the NCPI is improved by more than 1 cycle per instruction with our bypass scheme, while for the two other tasks the improvement is less significant.

In terms of percentage of bypassed cache blocks, the ratio is high for all tasks except *fdct*, meaning that the degree of pollution in the shared L2 cache is important. For these applications, reducing L2 cache pollution is expected to drastically reduce inter-core interference, which will be shown in the next paragraph. For *fdct*, the percentage of bypassed blocks is low, explained by the code structure, consisting of two large loops whose code does not fit into the L1 cache but fits entirely in the L2 cache. Note that the percentage of SSU blocks is much higher than when single-usage blocks are detected dynamically, such as in [18], reporting an average number of single-usage blocks of 33% for the SPEC CPU 2000 benchmarks. This difference comes from the fact that static cache analysis, for the sake of safety, underestimates the set of reused blocks compared to real executions.

Multi-core cache analysis with and without bypass. In this paragraph an architecture with two cores is considered. We estimate the WCET of a task running on one core, competing for the shared L2 cache with one of the eight tasks of Table 3 running on the other core, in a context without code sharing between tasks.

The results are presented in Table 5. For each task, the results without bypass of the L2 cache (first line) and with bypass (second line) are given. For each configuration, we give HR_{L2} obtained: *(i)* without interfering task running on the other core (see also Table 4) *(ii)* with an average of HR_{L2} with each of the eight tasks successively competing for the shared L2 cache; *(iii)* with the interfering task having the highest amount of interference.

		no-interf	multi-core	
Bench.	**bypass**	**alone** HR_{L2} (NCPI)	**average of adversaries** HR_{L2} (NCPI)	**highest adversary** HR_{L2} (NCPI)
minver	no	39.76% (3.62)	24.85% (4.51)	0% (6.01)
	yes	39.76% (3.62)	39.76% (3.62)	39.76% (3.62)
adpcm	no	33.60% (6.81))	20.60% (8.14)	0% (10.26)
	yes	33.96% (6.77)	33.76% (6.80)	32.90% (6.88)
fdct	no	84.03% (2.04)	24.88% (9.58)	0% (12.75)
	yes	84.03% (2.04)	73.32% (3.40)	7.34% (11.81)
statemate	no	0.72% (16.49)	0.19% (16.57)	0% (16.60)
	yes	1.21% (16.40)	1.21% (16.40)	1.21% (16.40)
fft	no	1.97% (11.02)	1.23% (11.10)	0% (11.24)
	yes	12.50% (9.83)	12.50% (9.83)	12.50% (9.83)
crc	no	98.97% (0.07)	61.65% (2.63)	0% (6.90)
	yes	98.97% (0.07)	98.97% (0.07)	98.97% (0.07)
lms	no	0.61% (12.68)	0.38% (12.70)	0% (12.75)
	yes	0.61% (12.68)	0.61% (12.68)	0.61% (12.68)
qurt	no	12.56% (5.62)	7.85% (5.92)	0% (6.43)
	yes	12.56% (5.62)	12.56% (5.62)	12.56% (5.62)

Table 5. Estimated worst-case L2 hit ratio and NCPI of the analysed task (bypass vs no bypass).

In column *average of adversaries*, the increase of worst-case hit ratio compared with a system without L2 bypass is significant. In terms of NCPI, the difference is substantial for a majority of tasks.

In the worst-case (column *highest adversary*), without bypass, the multi-core cache analysis always results in a value of HR_{L2} of 0% (in that case, the NCPI value is maximal). This value of 0% does not always occur with the same competing task. Moreover, for all tasks, the highest adversary is not unique; several adversary tasks result in a HR_{L2} of 0%. Said differently, the worst-case hit ratio in a system without L2 bypass is extremely poor. This demonstrates the pessimism of methods such as the base method presented in Section 3 and the approach described in [28], that consider *all* interferences between cores without any mechanism to decrease inter-core interference. If we now compare HR_{L2} when bypassing *SSU* blocks with HR_{L2} without interference, the value is the same for six out of the eight considered tasks, and very close for *adpcm* (1.04%). In contrast, for *fdct* the difference is significant (76,69%). This case occurs when *fdct* competes with itself for the shared L2 cache (recall that no code sharing is assumed in this paragraph), and results in a high volume of L2 cache consumed by *fdct*. When *fdct* is not competing with itself anymore, HR_{L2} becomes 75.06% and the difference of HR_{L2} compared to an interference free situation is then less

than 9%. In terms of NCPI, the average improvement is 2.1 cycles per instruction in the worst case scenario compared to a system without a bypass mechanism and up to 6.83 for *crc*.

In summary, using our bypass scheme, the amount of inter-core interferences is drastically reduced and the multi-core WCET is generally very close to the WCET without interferences for the shared L2 cache.

Scalability issues. The scalability of our bypass scheme is analysed by considering all tasks, each running on a distinct core, as interfering tasks of the analysed task. In this configuration the cumulated size of all interfering tasks is around 8 times bigger than the size of the L2 shared instruction cache. The results are presented in Table 6. The *fdct* task which needs the highest volume of L2 capacity even in the case of our bypass approach, is ommited in column 3 and considered in column 4. Recall that no hit in the L2 cache can be guaranteed even with one single task running on the other core when no bypass is used, as previously shown in Table 5.

Bench.	no-interf	all interfering tasks	
		except fdct	including fdct
	HR_{L2} (NCPI)	HR_{L2} (NCPI)	HR_{L2} (NCPI)
minver	39.76% (3.62)	38.55% (3.69)	0% (6.01)
adpcm	33.96% (6.77)	21.97% (6.83)	15.59% (8.66)
fdct	84.03% (2.04)	66.08% (4.32)	1.63% (12.54)
statemate	1.21% (16.40)	1.16% (16.41)	0% (16.60)
fft	12.50% (9.83)	6.82% (10.47)	0.88% (11.14)
crc	98.97% (0.07)	98.97% (0.07)	0% (6.90)
lms	0.61% (12.68)	0.61% (12.68)	0% (12.75)
qurt	12.56% (5.62)	12.56% (5.62)	0% (6.43)

Table 6. Estimated worst-case L2 hit ratio and NCPI of the analysed task (bypass, 7-8 interfering tasks).

When task *fdct* is left out, for a majority of tasks (*minver*, *statemate*, *crc*, *lms*, *qurt*) the decrease of HR_{L2} is low, compared with HR_{L2} when no adversary task runs on the other cores. For the other tasks, the decrease of HR_{L2} is larger. However, for all tasks, there are still hits in the L2 cache when using bypass. Due to the inter-cores interferences, the NCPI is increased but just by 0.38 cycles per instruction in average compared to the NCPI without interferences and by 2.28 in the worst-case (*fdct*). Moreover, the NCPI has an average improvement of 2.1 and up to 8.43 (*fdct*) compared to a system without bypass (Table 5). This shows the small impact of the inter-cores interferences on the WCET when using our bypass scheme in this configuration.

In contrast, when *fdct* is kept, the decrease of HR_{L2} is significant because *fdct* consumes a large percentage of the L2 cache capacity. With this kind of task, the proposed bypass approach is not sufficient to decrease the amount of interferences for the L2 cache. Additional methods such as cache partitioning have to be used to isolate such cache consuming tasks from the other tasks.

Code sharing. The impact of considering code sharing (paragraph 3.2) is evaluated in Table 7. Due to the difficulties to find different degrees of shared code in real-time tasks, the evaluation was achieved with a single task, running on 2 (resp. 3 cores), and considering in the analysis that a certain percentage of its code is shared by the task instances. Table 7 gives NCPI for the *fdct* task, running on one core, when one (resp. two) instance(s) of *fdct* are running on the other core (resp. two other cores). The amount of shared code between the instances is varied between 0% (no code sharing) to 100% (all the code is shared between the two instances). A percentage of x indicates that the first $\frac{x*code\_size}{100}$ bytes of code are shared between the competing task instances.

% of shared code	1 interfering instance		2 interfering instances	
	no bypass NCPI	bypass NCPI	no bypass NCPI	bypass NCPI
0 %	12.75	11.81	12.75	12.75
10 %	12.54	11.50	12.75	12.75
20 %	10.46	9.01	12.75	12.75
30 %	8.28	7.76	12.75	12.54
40 %	8.07	7.76	12.75	12.54
50 %	8.07	7.76	12.75	12.54
60 %	7.45	5.78	11.92	10.77
70 %	7.13	5.36	11.92	9.94
80 %	3.60	3.15	9.53	6.30
90 %	2.04	2.04	5.68	5.36
100 %	2.04	2.04	2.04	2.04

Table 7. Estimated NCPI of fdct with code sharing (no bypass/bypass, 1-2 fdct interfering instances).

The table shows that code sharing tightens WCET estimation as compared to a system in which code sharing is not considered.

Analysis time. In terms of computation time, the most time consuming situation of our experiments was described in paragraph 5.2. In this situation, up to nine tasks were analysed to determine the blocks to be bypassed for each of them, followed by a WCET estimation of the analysed task to account for inter-core interferences and bypassed cache blocks. The whole process always took less than 3 minutes on a Intel Core 2 Duo E6700 (2.66 GHz) with 2 GB of RAM.

6 Conclusions and future work

Estimating WCETs for multi-core platforms is very challenging because of the possible interferences between cores due to shared hardware resources such as shared caches. We have proposed in this paper a technique to reduce the amount of inter-task interferences, achieved by caching in the shared instruction caches(s) only blocks statically known as reused. Experimental results have shown that our approach allows to drastically reduce the WCET of tasks compared to methods which consider all inter-core conflicts and do not attempt to reduce their amount.

Our ongoing work is to extend our approach to data, unified caches and other cache replacement policies. Another direction for future research is to use bypass for non-shared caches

as well, in order to further reduce intra-task conflicts. Another direction would be to explore joint WCET estimation and scheduling to avoid some inter-task conflicts. Finally, a last direction would be to compare our proposal with cache locking and partitioning schemes, which avoid inter-task interference at the cost of a reduced cache volume per task/core.

Acknowledgments. The authors are grateful to Benjamin Lesage, Nathanaël Prémillieu, Pierre Michaud, André Seznec and to the anonymous reviewers for feedback on earlier versions of the paper.

References

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

[2] A. Colin and I. Puaut. A modular and retargetable framework for tree-based WCET analysis. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 37–44, Delft, The Netherlands, June 2001.

[3] H. Dybdahl and P. Stenström. Enhancing last-level cache performance by block bypassing and early miss determination. In *Asia-Pacific Computer Systems Architecture Conference*, pages 52–66, 2006.

[4] J. Engblom and A. Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, Dec. 1999.

[5] M. Farrens, G. Tyson, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *In Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 93–103, 1995.

[6] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

[7] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for real-life processor. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485, Tahoe City, CA, USA, Oct. 2001.

[8] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the 29th Real-Time Systems Symposium*, pages 456–466, Barcelona, Spain, Dec. 2008.

[9] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE, vol.9, n7*, 2003.

[10] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for WCET estimation. *Real Time Systems*, 34(3), Nov. 2006.

[11] T. Lundqvist and P. Stenström. 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*, pages 255–262, 1999.

[12] S. Malik and Y. T. S. Li. Performance analysis of embedded software using implicit path enumeration. *Design Automation Conference*, 0:456–461, 1995.

[13] S. McFarling. Program optimization for instruction caches. *SIGARCH Comput. Archit. News*, 17(2):183–191, 1989.

[14] F. Mueller. *Static cache simulation and its applications*. PhD thesis, Florida State University, 1994.

[15] F. Mueller. Timing analysis for instruction caches. *Real Time Systems*, 18(2-3):217–247, 2000.

[16] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 201–206, 2003.

[17] M. Paolieri, E. Qui nones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for wcet analysis of hard real-time multicore systems. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 57–68, New York, NY, USA, 2009. ACM.

[18] T. Piquet, O. Rochecouste, and A. Seznec. Exploiting single-usage for effective memory management. In *ACSAC '07: Proceedings of the 12th Asia-Pacific conference on Advances in Computer Systems Architecture*, pages 90–101. Springer-Verlag, 2007.

[19] G. N. Prasanna. Cache structure and method for improving worst case execution time. Patent Pending, 2001. US 6,272,599 B1.

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

[21] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 49–60, Washington, DC, USA, 2007. IEEE Computer Society.

[22] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 278–286, 2004.

[23] V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 300–303, 2008.

[24] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real Time Systems*, 18(2-3):157–179, 2000.

[25] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing analysis for data and wrap-around fill caches. *Real Time Systems*, 17(2-3):209–233, 1999.

[26] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem—overview of the methods and survey of tools. 7(3), 2008. ACM Transactions on Embedded Computing Systems (TECS).

[27] R. Wilhelm, D. Grund, J. Reineke, M.Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2009, To be published.

[28] J. Yan and W. Zhang. WCET analysis for multi-core processors with shared l2 instruction caches. In *RTAS '08: Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 80–89, 2008.