Predictable data caches in multi-core architectures

Supervisors : Isabelle Puaut, Damien Hardy
INRIA Rennes - Bretagne Atlantique, ALF project

Benjamin Lesage
June 2009

Research Master's degree in Computer Science
University of Rennes 1

Contents

Introduction

Real-time computing differs from general purpose computing in the sense that non functional properties may be of primary importance. Such a property is timing. Indeed in real-time systems, the time when the result is available is as important as the result itself. From the timing point of view, there are two families of real-time tasks: *soft* real-time and *hard* real-time. Missing a deadline, in the former case, might just result in a quality of service loss. However, in the latter case, the consequences of a deadline miss might be catastrophic: human life loss, environmental disaster... Because of these risks, upper bounds on the execution time of tasks are required; we need to know when a task will be over for sure.

Therefore the estimation of the actual worst case execution time (WCET) of a task, i.e. its longest possible execution time, is crucial to prove that the task meets its deadline. This estimation of the concrete WCET should be safe, i.e. larger than or equal to any possible execution time. It also has to be tight, close to the actual WCET of the task. In addition to the schedulability analysis of a system, establishing the assurance that each task meet its deadline, the derived WCET might also help not to overestimate this system hardware requirements.

However, due to their mostly dynamic behaviour, many architectural features require great efforts to be safely analysed. Nonetheless, these mechanisms, such as caches or pipelines, have to be considered during WCET computations as they tend to be commonly used in embedded real-time systems. Much research has been undertaken to predict WCET in complex uni-core architectures with caches, especially instruction caches. In contrast, few contributions tackle issues raised by the use of data caches such as imprecise static data address prediction or writes to the memory.

The same goes for multi-core architectures, which is annoying as these architectures tend to spread more and more. The main pitfall of multi-core architectures is the sharing of hardware resources between the different cores. Caches are part of the shared resources, which might increase the amount of conflicts in a cache.

Data caches are scarcely included in architectures analysed in the context of WCET estimation. There are even, to the best of our knowledge, no contribution which considers shared data caches in multi-core architectures. This document is a first approach towards solutions to this problem. For the time being, other hardware features impacting the WCET will be put aside, but the curious reader might look for further information in [38].

After a brief overview of methods used to bound the WCET of tasks (Chapter 1), we detail assumptions we lay on the analysed architectures (Chapter 2). Once these elements have been taken for granted, we lay the foundation, in the form of a multi-level data caches analysis (Chapter 3), of methods to tackle the problems raised by multi-core architectures with shared data caches. Based on this foundation, we detail methods that may be used to reduce conflicts inherent to the use of data caches in multi-core systems (Chapter 4). Finally, these methods are discussed and evaluated (Chapter 5).

Chapter 1

Bounding Worst-case Execution Time

In this chapter, *dynamic analysis* (§ 1.1.1) and *static analysis* (§ 1.1.2), the two families of methods to bound the WCET of a task are first presented. These presentations help to locate low-level analyses, i.e. analyses accounting for hardware effects, in the context of WCET computation.

After that, an example of low-level analysis is introduced (§ 1.2) with a foregoing familiarisation with the workings of caches (§ 1.2.1). First, we introduce techniques considering a task in isolation (§ 1.2.2), then in a multi-tasking preemptive system (§ 1.2.3). The chapter continues with a brief overview of methods used to increase caches predictability (§ 1.2.4).

Finally, the few studies, to the best of our knowledge, related to cache analysis in the context of multi-core architectures (§ 1.3) precedes a presentation of this study contributions (§ 1.4).

1.1 Worst-case execution time estimation

Two approaches stand out concerning the estimation of a task WCET. Measurements on or simulation of the target hardware (§ 1.1.1) yields tight WCETs. The task can otherwise be statically analysed (§ 1.1.2). The extracted properties about its behaviour are then used to produce a WCET estimate.

1.1.1 Dynamic analysis methods

Given a hardware platform, a task and its worst-case input, its WCET can be computed by executing the task and measuring its execution time. The problem is that determining this worst-case input is a difficult task. Exploring the whole input domain of a task may be too expensive; the number of cases to explore might be exponential in the number of task inputs.

[36] addresses this difficulty by using genetic algorithms, the analysed task input set providing a suitable population to the algorithm. Unfortunately, local optimum can trap the algorithm, although a global optimum is required.

Similar explorative methods, as developed in [39], exhaustively investigate the set of paths of the task instead. Yet, the issues to tackle are the same, the number of explorable cases might be too important. Restrictions can be set on the exploration space, like considering only a set of paths such that *every instruction is executed at least once*. However, these reductions might jeopardize the safety of the yielded WCET estimate.

The main problem of using dynamic methods to bound the WCET is safety. The vastness of the set of inputs and initial states for an analysed task might prevent its exploration. If all paths are not exhaustively explored, unlike [39], there is no guarantee that the measured time is the worst-case execution time. Thus, dynamic methods are mainly used as a mean to validate static analysis methods. The bounds provided by measurements are pretty tight and, if not exact, lower than the actual WCET.

Another difficulty is to simply measure the execution time of the task. Facilities provided by the hardware to perform these measures should not disturb the program execution; the observation must not alter the experiments results (well-known *probe-effect* [6]).

Due to these limitations of dynamic methods, the remainder of the document assumes the use of static analysis methods only.

1.1.2 Static analysis methods

Unlike dynamic analysis, static analysis methods are a twofold process based on the scrutiny of the program source code or final executable. First, a low-level analysis (see Section 1.2 for an example considering caches) considers the hardware configuration to bound the execution time of determined units of the task. Then using this information and a logical representation of the program, i.e. the links between the different units, a high-level analysis estimates the worst-case execution path of the task.

Program representation. The unit often used is the *basic block* (BB). A BB is the maximum set of consecutive instructions without branches incoming/leaving it except at entry/exit points. The exit (respectively entry) point is located at the end (respectively beginning) of the BB. A *Control Flow Graph* (CFG) is used to keep track of the relationships between BBs. Vertices of the CFG graph are BBs of the application. Edges represent possible control flow between BBs. Such a structure is illustrated in Figure 1.1, with the addition of *Start* and *End* nodes to pinpoint the program entry and exit points respectively.

```
WHILE $BB_1$ DO
   IF $BB_2$ THEN
      $BB_3$
   ELSE
      $BB_4$
   FI
   $BB_5$
DONE
$BB_6$
```

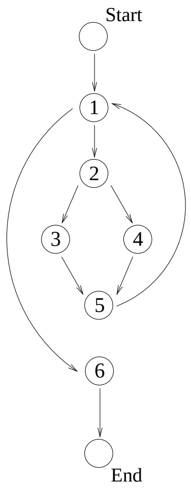


Figure 1.1: An example of simple CFG for a program including a loop with an embedded conditional. Six BBs have been extracted including : 1 the head of the loop, 2 the conditional evaluation, 3 (respectively 4) the then (respectively else) basic block.

Implicit Path Enumeration WCET estimation Techniques (IPET). Using a CFG, IPET methods [14] estimate the WCET of the corresponding task by solving an Integer Linear Programming (ILP) constrained problem. A constant t_b is associated to each BB b, holding its WCET as returned by the low-level analysis[1]. In addition, a frequency variable x_b represents the maximum execution frequency of BB b. The objective function, to compute the WCET of the task, is defined by:

$$max(\sum_{b \in BasicBlocks} x_b \times t_b) \quad (1.1)$$

where *BasicBlocks* is the set of BB of the task. Relations between the execution frequencies of BBs, so-called *flow constraints*, are represented using e_{b_1,b_2} variables, the execution frequency of the edge coming from b_1 to b_2 in the CFG.

The problem of knowing whether or not a task halts being undecidable [32], we need additional information, as an example supplied by annotations in the code, to bound loop iterations. Those bounds further restricts the solution of equation 1.1. As an example, the system generated for the CFG of Figure 1.1 :

[1]The WCET of a basic block is assumed to be known and context-independent

Flow constraints :

$$x_{Start} = 1$$
$$e_{Start,1} = x_{Start}$$
$$x_1 = e_{Start,1} + e_{5,1}$$
$$x_1 = e_{1,6} + e_{1,2}$$
$$x_2 = e_{1,2}$$
$$x_2 = e_{2,3} + e_{2,4}$$
$$x_3 = e_{2,3}$$
$$x_3 = e_{3,5}$$
$$x_4 = e_{2,4}$$
$$x_4 = e_{4,5}$$
$$x_5 = e_{3,5} + e_{4,5}$$
$$x_5 = e_{5,1}$$
$$x_6 = e_{1,6}$$
$$x_6 = e_{6,End}$$
$$e_{6,End} = x_{End}$$
$$x_{End} = 1$$

Loop bound constraint :

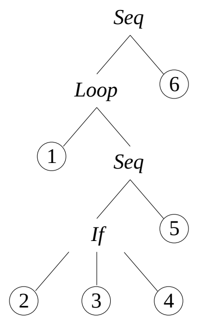
$$e_{5,1} \leq 100$$

WCET expression :

$$WCET = max(x_1 \times t_1 + ... + x_6 \times t_6)$$

Maximizing the objective function in the context of an ILP problem might be very time-consuming (NP-hard in theory [26]). However, with the set of constraints generated by such an analysis, the problem was proved in [14] to collapse to a LP one (polynomial resolution time [26]).

Tree-based analysis. To perform tree-based analysis [23], the tree attached to a task is constructed using a syntax-driven method. Nodes in the tree represent control structures of the programming language. Leaves are the BBs of the program and are valued by their worst-case execution time as computed by low-level analysis. The estimation of the WCET is then performed by a bottom-up traversal of the tree using specific combination rules for each kind of node. The tree corresponding to the CFG of Figure 1.1 is illustrated in 1.2 with the associated combination rules.



$$T(Seq(N_1, N_2)) = T(N_1) + T(N_2)$$
$$T(Loop(N_{Cond}, N_{Body})) = T(N_{Cond}) \times (MaxIter + 1) + T(N_{Body}) \times MaxIter$$
$$T(If(N_{Cond}, N_{Then}, N_{Else})) = T(N_{Cond}) + max(T(N_{Then}), T(N_{Else}))$$

Figure 1.2: An example of tree for the program presented in Figure 1.1. Three node types are illustrated: *Loop*, *Seq* and *If*. Considering *Loop* and *If* nodes, the leftmost son is the condition evaluation. The rightmost for the *Loop* (resp. *If*) node is the loop body (resp. else branch, thus leaving the intermediate node as the then branch). For a *Seq* node, sons are supposed to be executed from the leftmost to the rightmost one. Considering the combination rules for the tree, $T(N)$ is the WCET of node N. This value is computed by the low-level analysis if N is a leaf in the tree.

Tree-based methods allow a very fast estimation of WCET at the detriment of precision and flexibility. Indeed, tree-based analyses are close to the structure of the program. This allows plain localization of time-consuming structures in the application. But they are very sensitive to compiler optimizations that alter the application structure.

1.2 Low-level analysis, considering caches in uni-core architectures

Since this study focuses on the impact of caches on WCETs in multi-core architectures, the mechanism of caches is first introduced (§ 1.2.1). Then existing static cache analyses are presented (§ 1.2.2). Nonetheless, until Section 1.2.3, the analysed task is considered in isolation. After that, existing methods to consider multiple task are introduced, in the context of preemptive systems (§ 1.2.3). Finally, slightly different studies, working on cache modifications to improve predictability, conclude this Section (§ 1.2.4). The objective is to lay down basics and clues for cache analysis in the context of multi-core architectures.

1.2.1 Cache architecture

Caches are fast but small memories which aim at filling the increasing gap between fast processors and relatively slower main memories. A cache holds a subset of the main memory which is first inquired when the processor is looking for information from the memory. If the information is present in the cache, a so-called *cache hit*, it is directly delivered to the processor. Otherwise, if the requested information is absent from the cache, it is first conveyed from main memory to the cache, thus dealing with the *cache miss* occurrence.

However, upon a cache miss, the requested information is not solely inserted in the cache. Instead, a full memory block, whose size depends on the cache configuration, is transferred from the main memory.

The whole underlying idea of caches is to exploit the temporal and spatial locality principles of programs. These principles state that if a memory reference targeting location A occurs at time t, A or nearby memory locations have a strong probability of being referenced at time $t + \epsilon$.

Cache structure. Caches are divided in fixed-size blocks which represent the transfer unit with the main memory [28]. A given memory block, can only be stored in a fixed number of cache blocks according to its address in memory. The number of valid positions for this block is called the *associativity* degree or the number of *ways* of the cache. The group of valid cache blocks to store a memory block into is called a *set* of the cache. Such caches are called *set-associative* caches. Note that special cases of set-associative caches exist:

- direct mapped where the associativity degree of the cache is one (a memory block has a unique valid position in the cache)
- and fully associative caches when there is only one set in the cache, which then holds all the cache blocks (each memory block can go anywhere in the cache).

Figure 1.3 illustrates the different families of set-associative caches by emphasizing the valid positions of a given memory block.

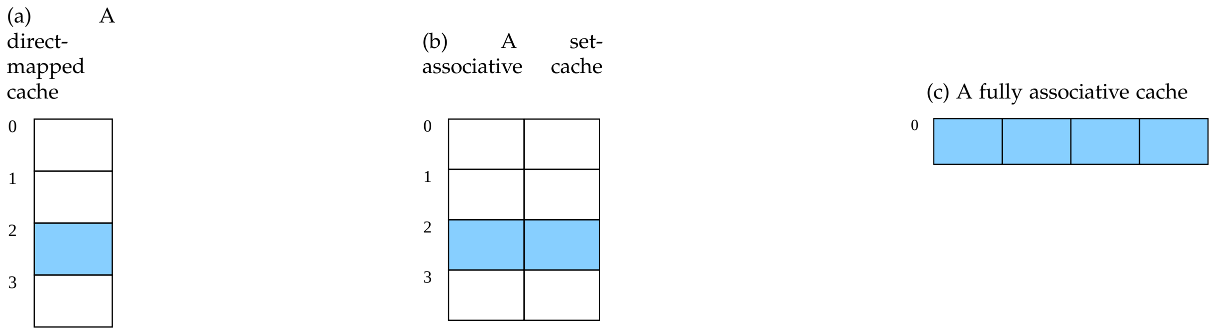

(a) A direct-mapped cache
0
1
2
3
(b) A set-associative cache
0
1
2
3
(c) A fully associative cache
0

Figure 1.3: The different families of caches are highlighted, by the depiction of the set of valid positions given a memory block, from the direct-mapped cache (a) to the 2-way set-associative cache (b), along with the fully associative cache (c).

Replacement policy. With the exception of direct-mapped caches, a memory block is restricted to a cache set composed of at least two cache blocks. In the event of a cache miss, if the host cache set of the addressed memory block is full, a block of this set has to be evicted from the cache. Which block is going to be evicted depends on the cache *replacement policy*. Among the most famous replacement policies are LRU (evicts the Least Recently Used block), PLRU (Pseudo-LRU, using approximate information about which block is the least recently used), random (evicts a random block) and Round-Robin (blocks are managed with in a FIFO style). The first one (LRU) is the most studied in WCET estimation research because of its higher predictability [24].

Cache contents nature. Two main kind of information are fetched by a task from the memory. First are the instructions it executes, loaded according to the control flow of the task. Second are the data it is working on, accessed under the impulse of specific instructions. In this context, caches can be used specifically to store instructions, to store data or for storing both in the case of unified caches.

Writing policy. Data and unified caches further differs from instruction caches as they have to deal with data modifications during execution. The writing mechanism has two main ways of accounting for the caches:

– *write-through* : both the cache and the memory are updated;
– *write-back* : only the cache is updated; the main memory update is delayed till the modified block is evicted from the cache.

In addition, two behaviours have been defined to deal with data absent from the cache upon a write:

– *write-no-allocate* : the modified block is not inserted in the cache if absent upon a writing;
– *write-allocate* : the block, if absent from the cache upon a writing, is inserted normally, as during a load, from the memory.

Cache hierarchies. To further reduce latencies of accesses to the memory, multiple caches can be chained together to build a hierarchy, like the one in Figure 1.4. If the main memory is the lowest level, the higher in the hierarchy, the faster the caches are but also the smaller. The successor of a cache in the hierarchy acts as the main memory in the context of a single cache.

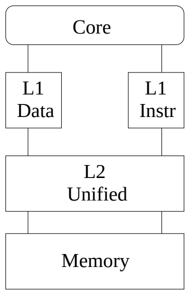


Figure 1.4: Illustration of the notion of memory hierarchy, with two levels of caches. Two L1 caches (one for Data and one for Instructions) and a unified L2 cache above the main memory.

Different properties can be defined between caches of subsequent levels. Inclusion means that the content of higher level caches is always included in lower level ones. On the other hand, exclusion prevents data from being present at different levels of the cache hierarchy. Finally, non-inclusive caches enforce neither inclusion nor exclusion.

1.2.2 Static cache analysis

Due to their dynamic behaviour caches raise predictability issues in the context of real-time systems. The idea of static cache analysis methods is to derive, for each memory reference, its worst-case behaviour (hit/miss) with regards to the instruction or data cache. This information is then used in the WCET computation of the BB. Times computed, for each BB, are then used by high-level analyses. As an example in IPET (§ 1.1.2), they are included in the execution time - constant t_b - of the BB.

To enforce safety, static cache analysis methods have to account for *every* possible cache contents, at every point in the execution, considering all paths altogether. Possible cache contents can be represented either as sets of *concrete cache states* [20] or by a more compact representation called *abstract cache states* (ACS) [31, 18].

Two main classes of approaches [31, 18] have been defined for the timing analysis of architectures with a single instruction cache. In [31], *abstract interpretation* [3] is applied. ACSs computed by three different analyses, using fixpoint computation, are attached to each instruction. Then it is possible to determine if a memory block will *always* be present in the cache (*Must* analysis), *may* be in the cache (*May* analysis), or if a memory block will not be evicted from the cache once it has been loaded (*Persistence* analysis). A *cache categorization* (e.g. *always hit*) can then be assigned to every instruction based on the ACSs contents. Originally designed for set-associative instruction caches implementing the LRU replacement policy, the method has later been extended in [10] to other replacement policies for instruction caches. Support for hierarchies of instruction caches was added later in [9] using *cache access classifications* to safely identify references that may, must or never occur at every level in the cache hierarchy.

In [18], a very similar method, *static cache simulation* is used to compute every possible content of the cache before each instruction. Abstract cache states are computed using a data-flow analysis. Similarly, a cache categorization is used to express the worst-case behaviour of instructions with regards to the cache. [19] extends the contribution of [18] to deal with set-associative instruction caches, in addition to direct-mapped caches.

A peculiarity of data caches, compared to instruction caches, arises as the precise target of some references may not be statically computable. This is due to the possible range of locations that might be accessed by the same instruction in different contexts. If for scalar or array accessed in a regular way a prediction might be possible (considering global variables or on stack accesses) [15], it is not a generality as an example if the access is dependent of the analysed task inputs.

A first solution, to deal with data memory references is to consider these imprecise accesses and their possible impact on ACSs. This was the direction followed by improvements to [19] and [31] in [37] and [4, 27] respectively.

An alternative solution to deal with data caches are *Cache Miss Equations* (CME) [35, 34]. To estimate the cache behaviour, the iteration space of loop nests is represented as a polyhedron. *Reuse vectors* [40] are then defined between points of the iteration space. CME are then set up and resolved to accurately locate misses. This method has been successfully applied to data caches. However, according to the authors, this approach suffers from a lack of support for data dependent conditionals.

Another scarcely addressed characteristic of data caches is the impact of memory modifying instructions. In [5], an analysis was proposed based on the *write-back* update policy. The objective is to estimate the *write-backs*, i.e. the moment when a modified data might be replaced in the cache.

1.2.3 Preemptive multi-task systems

Yet, the presented methods relies on the analysis of a task in isolation. In this context, cache unpredictability results from *intra-task* conflicts, i.e. conflicts due to different possible paths in the task control flow. Multi-tasking systems add another pitfall in the form of *inter-task* conflicts. The execution of tasks, sharing the cache with the analysed one, might alter cache contents upon each preemption and thus invalidate the results of the static cache analyses.

In the context of preemptive systems, the analysed task might relax the processor to the benefit of a higher priority task. When the analysed task regain control of the processor, the cache contents might have changed thus triggering unpredicted cache misses. The *Cache Related Preemption Delay* (CRPD), introduced in [12, 20, 29], is an estimate of the cost induced by the preemption of the analysed task by another task from the caches point of view.

By computing, using methods similar to 1.2.2, the intersection of *reaching cache blocks*, blocks which may be present in the cache given a program point, and *live cache blocks*, blocks that may be reused beyond the same program point, one can define the concept of *useful cache blocks* [12]. *Useful cache blocks* are the blocks that, if evicted from the cache, might trigger unpredicted misses after a preemption. A rough estimation of the CRPD is to compute the product of a cache miss latency and the size of the *useful cache blocks* set.

This method was later refined, as an example in [20], to analyse the preempting task jointly with the preempted one. The objective is to estimate the cache blocks that may be used by the preempting task, thus evicting useful blocks of the preempted one. [20] further relies on the computation of the combination of all the cache states that might reach a program point. The objective is to improve the precision of the analysis by computing the possible scenarios. [29] on the other hand mainly target the number of preemptions the analysed task might suffer and especially masked preemptions, i.e. the preempting task being itself preempted.

1.2.4 Improving cache predictability

Techniques to classify memory accesses according to their behaviour with regards to the cache have been presented. Most of them try to deal with the predictability issues raised by intra-task conflicts, inter-task conflicts, accesses indeterminism (in the case of data caches) or cache replacement policies. However, considering these elements in the analysis increases pessimism. Cache partitioning, locking and accesses bypassing have been introduced to tackle the sources of these predictability issues.

Locking. Locking requires hardware support to lock part of (or all) the cache. Blocks in the locked cache area remain in the cache and no additional information can be inserted in this same area; the replacement policy is deactivated for this area. Invariable contents can be loaded in the cache before it is locked. Thus we know precisely the cache contents and any access to out of cache information will result in a miss (conversely, an access to data locked in the cache results in a hit). Such a mechanism addresses most of the predictability issues described earlier. Nonetheless, the locking-incurred misses might aggravate the task average-case and worst-case execution time.

The first approach to locking, *static cache locking*, was applied as an example to instruction caches in [1, 22]. Using this method, the cache is locked for the whole task execution and preselected content is loaded during the system initialisation. The major issue is the selection of information to lock in the cache. Only these pieces of information indeed produce cache hits and an unfortunate choice might significantly degrade the average and worst case execution time of tasks, compared to an unlocked cache. The method proposed in [22], as an example, locks in the cache the blocks with the highest execution frequency. However, changes of the worst-case execution path of the task occurring as content is locked in the cache are not considered. The genetic algorithm described in [1] does not suffer from this issue but the additional precision comes at the expense of computation time.

To reduce the loss inherent to the use of static locking, an option is to use *dynamic locking*. Instead of locking the same content during the whole task execution, only chosen regions of the program are executed using a locked cache. In [33], focusing on data caches, regions whose behaviour might be hard to predict are locked. Otherwise, the cache is left unlocked. On the other hand, [21] proposes to divide the whole task in regions, each executed with a locked cache. A cache content is then associated to each region. Again, the non-trivial problem of determining a correct set of regions and/or cache contents to lock was addressed using genetic [21] or greedy [33, 21] algorithms.

Partitioning. *Cache partitioning* may be used to reduce inter-tasks conflicts. Restrictions on each task apply in such a way that a task can only use its allocated subset of the cache. Partitioning can be achieved by software methods like [17] which uses both the linker and the compiler to remap instructions or data of each task to specific cache partitions. Hardware methods as proposed in [11] alter the function mapping blocks to the cache sets according to their address, restricting the cache allocation of a given task. Compared to locking, the main difficulty lays in the correct partitioning of the cache. A task should have a partition whose size fits its requirements. The different metrics to drive this decision process include, among other things, the different tasks working sets size in [17] or the global system utilization [25]. The major drawback of partitioning is that tasks are executed using smaller cache sizes which might degrade their WCETs.

Bypassing. Finally, the *bypass* of data structures classified as unpredictable has been introduced in [15]. Data structures (such as arrays) whose behaviour when accessed may not be statically computable are moved to a non-cacheable segment of the main memory. Thus, accesses to these structures will always miss in the caches which means they are detected as non conflicting with predictable data. Furthermore, inter-task conflicts may be reduced [7] as the number of conflicting cache blocks is lowered. However, the method depends on the analysis of accesses predictability. If this analysis is not precise enough, too many data structures might be bypassed.

1.3 Low-level analysis, considering caches in multi-core architectures

Multi-core architectures tend to develop since increasing the speed of a single core would be too costly considering electric consumption and thermal dissipation. As these architectures tend to spread, even in the real-time domain, one should still be able to yield safe and tight tasks WCET estimates. However, the main difference between uni and multi core architectures is that considering the latter, resources (such as buses, caches...) may be shared between the tasks executing on each core.

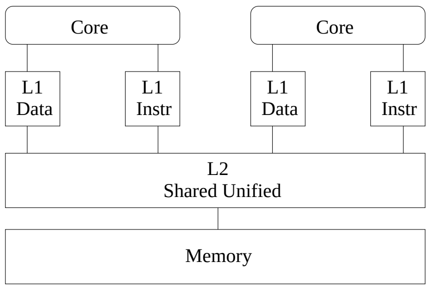


Figure 1.5: Multi-core architecture example with two cores sharing a unified L2 cache.

As illustrated in Figure 1.5, caches are such a shared resource and the problem may seem similar to multi-tasking systems introduced previously (§ 1.2.3). However, any of the tasks running on other cores might alter the cache *any* time while the analysed task is executed (in contrast to preemptions only in 1.2.3). Static cache analyses should take this phenomenon in consideration. Computing the interleaving of cache accesses for all the tasks sharing a cache is an unrealistic solution because of the tremendous number of possibilities. [41] addresses the problem, for a L2 shared instruction caches, by analysing a real-time task on a first core, while a non real-time task runs on an other core. They identify conflicting instructions between the two tasks, i.e. instructions mapping on the same cache set. Instructions are then classified as in static cache analyses (§ 1.2.2) taking into account the respective positions of conflicting cache blocks in the two tasks (mainly inside or outside loops). As shown in [7], the classifications derived using their methods may not be safe, assuming too few cache misses for the real-time tasks.

With the same idea of analysing a task and its rivals, [7] proposed a solution based on the calculation of *cache block conflict number* (CCN) for each set of the instruction cache, i.e. number of memory blocks mapping to the cache set for rival tasks. Cache categorizations are then updated according to memory blocks that remain in the cache once shifted from the CCN corresponding to their set. As the method is overly pessimistic, the authors also use bypass to reduce the amount of cache blocks which reach the shared cache levels.

On the other hand, [30] explores combinations of mechanisms to reduce instruction cache predictability issues which are strengthen by the multi-core context. Many solutions are proposed involving partitioning and both static and dynamic variations of locking. The study aims at outlining patterns of tasks whose behaviours are interesting under such restrictions. As an example, per-core partitions seem more interesting as they prevent tasks which are not running from keeping portions of the cache. The performance loss attributed to preemptions by tasks running on the same core is indeed less significant than the one induced by unused cache areas in per-task partitioning.

1.4 Contributions

The methods presented so far mostly target instruction caches and, with the exception of [7], still for instruction caches, do not address the problem of cache hierarchies. Similarly, the few studies focusing on multi-core have been introduced. But data caches, already scarcely addressed in uni-core based contributions, are plainly absent from multi-core related studies.

This study focuses on these two issues for data caches. First, we present to which extent the approach in [9] can be applied to data caches in chapter 3 along with a safe way of providing estimates of memory references contribution to the WCET, with regards to data caches. Based on this analysis, different variants of methods usable in the context of multi-core architectures with shared data caches are presented in chapter 4. On the one hand, partitioning precludes inter-tasks conflicts by providing isolation. On the other hand is the analytical method considering inter-core conflicts. We also introduce bypass heuristics to tighten our WCET estimates. Finally, these elements and their combination with bypass are evaluated (see chapter 5) and their respective interests discussed.

Chapter 2

Assumptions and notations

2.1 Hardware

Mono-core issues. As this study focuses on data caches, code is assumed not to interfere with data in the different considered caches. There is no assumption on the means used to achieve this separation, whether they are software or hardware based. An architecture without timing anomalies [16], caused by interactions between caches and pipelines, is however assumed; a miss is assumed to be the worst-case behaviour.

The considered cache hierarchy is composed of N levels of data caches, cache level 1 being the closest to the processor, as depicted in Figure 2.1. Each cache implements the LRU replacement policy. Using this policy, cache blocks in a cache set are logically ordered according to their age, the least (respectively most) recently referenced one being the oldest (respectively youngest). If a cache set is full, upon a load in this set, the evicted block is the oldest one whereas the most recently accessed block is the youngest one.

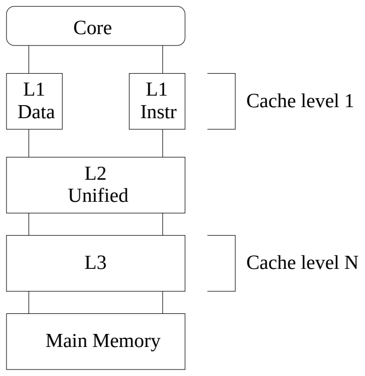


Figure 2.1: An example of 3-levelled cache hierarchy with unified L2 and L3 caches and distinct L1 data and instruction caches.

A datum accessed by an instruction is located in a single memory block; in other terms, data are aligned on cache block boundaries. Thus cache line size of level L is assumed to be a multiple of the cache line size of level $L-1$. However, no assumption is made concerning the cache sizes or associativities. Furthermore, the following properties are assumed to hold:

P1.[**load**] 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 $L-1$. Cache of level 1 is always accessed.

P2.[**load**] Every time a cache miss occurs at cache level L, the entire cache line containing the missing piece of information is loaded into the cache of level L.

P3.[**store**] The modification issued by a store instruction goes all the way through the memory hierarchy. Writes to the cache levels where the written memory block is already present are triggered, along with the update of the main memory. Otherwise, if the information is absent from a cache, this cache level is left unchanged.

P4.[**store**] Upon a write in a cache block, wherever is the cache in the hierarchy, no block age modification is induced.

P5. There are no action on the cache contents (i.e. lookup/modification) other than the ones mentioned above.

Property P1 rules out architectures where cache levels are accessed in parallel to speed up information lookup. P2 excludes architectures with exclusive caches, whereas P5 filters out cache hierarchies ensuring inclusion.

Properties P3 and P4 on the other hand address the store instructions behaviour. P3 ensures the use of the *write-through* update policy. In combination with P3, P4 corresponds among other things to the *write-no-allocate* update policy for members of the cache hierarchy, as illustrated in Figure 2.2.

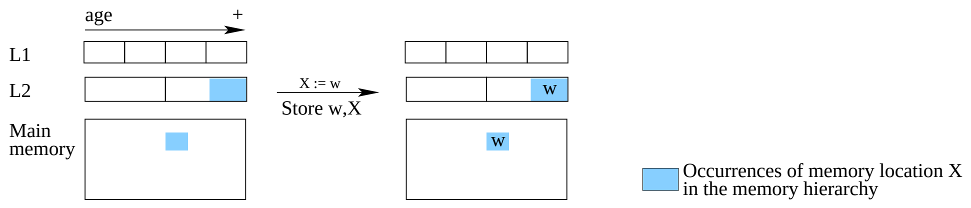


Figure 2.2: Example of the memory hierarchy behaviour upon a store instruction (write-through and write-no-allocate policies). The location of X, the updated location, in the memory hierarchy is highlighted.

Finally, the latencies to access the different levels of the memory hierarchy are assumed to be bounded and known.

Multi-core issues. Additional restrictions, focusing on the use of a multi-core system are presented below. Because we want to study multi-core related conflicts, instead of preemption related ones, a task instance is tied to a unique core. Additionally, while running on a core, a task instance is never preempted. Furthermore, there is no task migration at run-time. Tasks are further assumed not to synchronize with each other.

The architecture provides a per-instruction bypass mechanism; each load instruction can be statically set to bypass or not a cache level. If a load instruction i bypasses cache level L, all executions of i will left cache level L unchanged. As this choice is local to each instruction, it is not contextual.

Load instructions possibly accessing shared data are assumed to bypass all cache levels which are not shared by all the cores. This excludes private copies of shared data; there is no need for a coherency protocol.

2.2 Software

As abstract interpretation based methods have already been refined to deal with data caches [4, 27], we base our cache analysis on these studies. Static data cache analyses are further extended, and especially management of imprecise accesses in the may, must and persistence analysis with the support of different replacement policies.

Furthermore, the choice of caches partitions, explored in 4.1, is based on the algorithm presented in [25].

WCET estimation is performed using IPET based methods [14] and memory references contribution, with regards to data caches is estimated using our proposed mechanism (§ 3.3).

2.3 Notations and vocabulary

We define a *memory reference* as a reference to *data* in the memory triggered by a load or store instruction in a fixed call context; a memory reference is tied to a unique instruction and a unique call context. $memory\_references_t$ will denote the set of memory references of task t and $memory\_references_t(i)$

its restriction to the memory references issued by instruction i. Note that t may be omitted when analysing a well-identified task.

Speaking of the impacts or the classifications of an instruction, we refer in fact to the impacts or classifications of memory references linked to this instruction.

Furthermore, $M = N + 1$ refers to the main memory in the memory hierarchy and $memory\_blocks_{r,L}$ to the range of memory blocks that memory references r target on cache level L.

Chapter 3

Multi-level data cache analysis in the context of uni-core architectures

The main contribution of this document lays in the proposal and the comparison of methods to tackle the use of shared data caches in the timing analysis of multi-core architectures (§ 4). To achieve such an objective, a foundation to build data caches analysis is required. In light of these facts, we propose in this chapter a new multi-level data cache analysis, which will later be extended in Chapter 4 to support multi-core architectures. The proposed analysis, also detailed in [13], reuses most of the concepts defined in [9], which is however limited to instruction caches.

Most of the involved steps in our computation of WCET contribution for hierarchies of data caches are presented. The structure of the whole method is outlined in Figure 3.1.

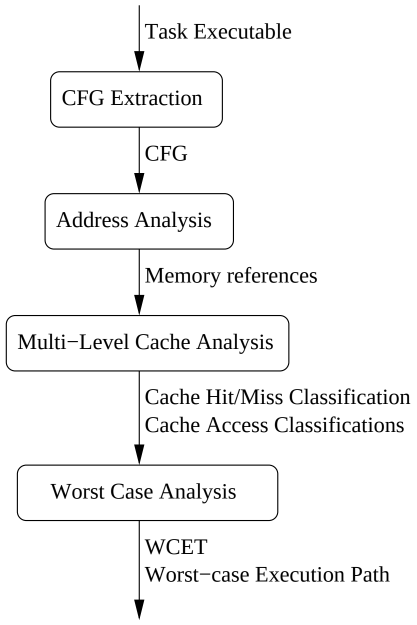

Task Executable
CFG Extraction
CFG
Address Analysis
Memory references
Multi−Level Cache Analysis
Cache Hit/Miss Classification
Cache Access Classifications
Worst Case Analysis
WCET
Worst−case Execution Path

Figure 3.1: Complete task analysis overview

After a first step that extracts a *Control Flow Graph* from the analysed executable, a *data* address analysis is performed (§ 3.1.1). The objective is to attach to every memory reference a safe estimate of the accessed data addresses.

Then, the caches of the hierarchy are analysed one after the other (§ 3.1) based on this address information. For each cache level and each instruction issuing memory operations (store and load instructions in the context of data caches), a *cache hit/miss classification* (CHMC) is computed. These classifications represent the worst-case behaviour of this cache level with regards to this memory reference.

To be safe, the analysis of a cache level further relies on a *cache access classification* (CAC, introduced in § 3.2) which represents, given a memory reference, a safe estimate of whether or not it occurs at a cache level.

In the end, a timing analysis of memory references with regards to data caches (§ 3.3) is performed with the help of both the CAC and CHMC for each cache level. Such information may then be used to compute the longest possible execution path and finally the WCET of the task.

3.1 Single level data cache analysis

Below, a data cache analysis abstracted from the multi-level aspect is introduced. All references are analysed and access filtering, according to other cache levels, is introduced later (§ 3.2). The analysis method which is presented along with the different mechanisms to handle the specificities of data caches, compared to instruction caches, can be seen as an extension of [9] to data caches.

Focusing on a data cache, the timing analysis of memory references is performed using the worst-case behaviour of this data cache for each memory reference. This knowledge is itself based on the cache contents at the studied program point, thus requiring a safe estimate of memory blocks present in the cache. Those estimates are computed using abstract interpretation and fixpoint computation.

3.1.1 Address analysis

The address analysis, in the context of data caches, aims at computing for every memory reference (as defined in chapter 2) the memory location it may access. Such information may however not be precisely computable, hence producing an over-approximation to yield safe values for subsequent analyses. These over-approximations take the form of ranges of possibly accessed memory blocks instead of a reference to a precise memory block.

The address analysis uses data-flow analyses which first computes stack frames for each valid call context and then analyse register contents for each basic block. Considering both global and on stack accesses, the precise address of scalar is yielded whereas the whole array address range is returned for accesses to array elements.

Note that the analysis used below is the one proposed in [8]. For further information, the interested reader might read [8] or [37] for an other example of address analysis.

3.1.2 Cache hit/miss classifications (CHMC)

A *cache hit/miss classification* is used to model the defined cache behaviours with regards to a given memory reference. To such a reference has already been attached a set of possibly accessed memory blocks by the address analysis. This set of memory blocks is used to compute the CHMC:

– *always-hit* (AH) : all the possibly accessed memory blocks are guaranteed to be in the cache;
– *first-miss* (FM) : for every possibly accessed memory block, once it has been first loaded in the cache, it is guaranteed to stay in the cache afterwards;
– *always-miss* (AM) : all possibly accessed memory block are known to be absent from the cache;
– *content-independent* (CI) : if the behaviour of the memory reference does not depend on the cache content;
– *not-classified* (NC) : if none of the above applies.

The CI classification is used for store instructions, which, according to our hypotheses do not depend on the cache contents.

3.1.3 Collecting cache information

Abstract cache states (ACSs) are used to collect information along the task CFG. This abstraction allows the modelling of a combination of concrete cache states, in terms of present memory blocks and their relative age. This is required as all paths have to be considered altogether to yield safe values.

Different analyses, similarly to [31], are required to collect information about cache contents at every program point. Compared to [27, 4], previous extensions of [31] for data caches, we are not confined to LRU replacement policy. Any modification of the analyses to handle different cache replacement policies for instruction caches can be adapted to data caches using our solution.

For each analysis, fixpoint computation, similar to algorithm 1, is applied on the program CFG, for every call context:

a *Must* analysis determines if a memory block is always present in the cache, at a given point, thus allowing a *always-hit* classification;

a *Persistence* analysis determines if a memory block will not be evicted from the cache once it has been first loaded, as in the definition of the *first-miss* classification;

a *May* analysis determines if a memory block may be in the cache at a given point, otherwise the block is guaranteed not to be in the cache thus possibly allowing a *always-miss* classification. If, at a given point, some possibly accessed memory blocks are present in the May analysis but neither in the Must nor the Persistence one, the *not-classified* classification is assumed for this memory reference.

Then, for each memory reference, its set of possibly accessed memory block is compared to the memory blocks inside the input ACS computed by each analysis.

Algorithm 1 Pseudo-algorithm for cache contents computation

```
//pick successively selects an element of a set, without infinitely omitting an element of this set
while pick memory reference $r$ with unstable $ACSs$ do
  if $r$ has only one predecessor, $r'$ then
    $ACS_{input}(r) \leftarrow ACS_{output}(r')$
  else
    $ACS_{input}(r) \leftarrow$ Join$_x$ $ACS_{output}$ of $r$ predecessors;
  end if
  //memory_blocks$_r$ is attached to $r$ by the address analysis
  $ACS_{output}(r) \leftarrow$ Update$_x$($ACS_{input}(r)$, $memory\_blocks_r$);
end while
```

Join functions : For instructions on branch reconvergence, the $Join_{Must}$, $Join_{Persistence}$ and $Join_{May}$ (respectively for the Must, Persistence and May analysis) are used as a mean to compute input ACS from output ACS of predecessors in the CFG:

$Join_{Must}$ computes the intersection of memory blocks present in the input ACSs, keeping for each one its maximal age as shown in Figure 3.2a;

$Join_{Persistence}$ keeps the union of memory blocks present in the input ACSs, as for $Join_{Must}$, the maximal age of memory blocks is kept;

$Join_{May}$ computes the union of memory blocks present in the input ACSs, keeping for each one its minimal age.

Considering data caches, these functions are the same than the ones defined for instruction caches [31].

Update functions : The effects of memory references on the cache are modelled using the $Update_{Must}$, $Update_{Persistence}$ and $Update_{May}$ for the Must, Persistence and May analysis respectively. In all the cases, only load instructions have an impact on the cache contents. According to our hypotheses, store instructions have no impact on ACSs (use of the write-through and write-no-allocate update policies).

Considering data caches, the *Update* function of the different analyses is of particular interest. Indeed, it has to deal with accesses indeterminism, when a precise memory location cannot be statically defined for a load instruction. Two options have to be considered.

On the one hand, the precise accessed memory block may have been computed by the address analysis. The *Update* function is then pretty straightforward, as illustrated in Figure 3.2b for the Must analysis. Thus considering the $Update_{Must}$, the accessed block is put at the head of its cache set and the younger lines are shifted, i.e. made older and evicted if too old. Note that memory blocks outside an ACS are supposed to be older than the ones present in this same ACS. Furthermore, the different *Update* functions behaviour, for the must, may and persistence, in this case is the same than for instructions [31].

On the other hand, when the address analysis yields a set of possibly accessed memory blocks for a memory reference, only one member of this set is actually accessed. To model this behaviour, a copy of the ACS_{input} is created, using the appropriate *Update* function, for each possibly accessed memory block. Then, all the updated copies are unified, this time using the appropriate *Join* function to produce an ACS_{output}. This process is illustrated on Figure 3.2c, for the Must analysis.

(a) Join function of Must analysis

(b) Update function of Must analysis while accessing memory block c.

(c) Update function while accessing range of blocks [a,c]

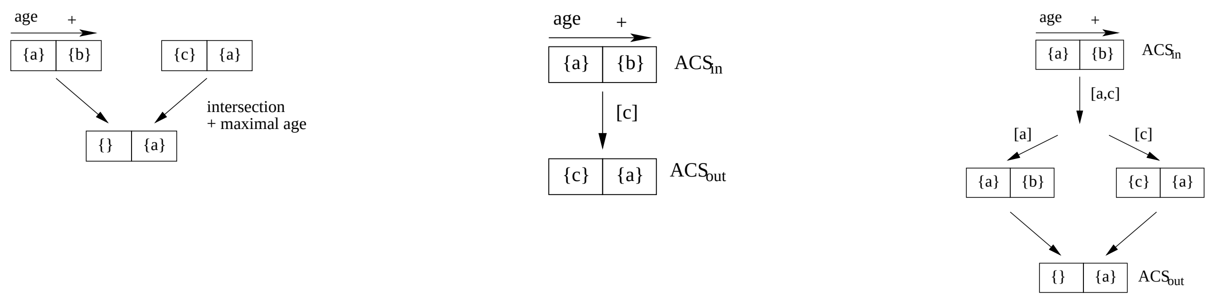


Figure 3.2: $Update_{Must}$ and $Join_{Must}$ functions effects on a 2-way abstract cache state, using the LRU replacement policy.

Termination of the analyses. In the context of abstract interpretation, to prove the termination of an analysis, it is sufficient to prove the use of a finite abstract domain and the monotony of the transfer functions.

ACS domain was shown to be finite in [31]. Moreover, [31] demonstrates the monotony of the $Join_x$ and $Update_x$ functions ($x \in \{Must, Persistence$ or $May\}$), for instructions and thus for accesses to precise memory blocks.

In our case, the modification to be proved is the one applied to this same $Update_x$ function to handle accesses to possibly referenced memory blocks.

Proof: When an ACS is updated using a set of possibly referenced memory blocks for an analysis x, a composition of the $Update_x$, for each block, and the $Join_x$ function is performed. As the composition of monotonic functions is monotonic, our modifications ensure monotony. □

3.2 Multi-level analysis

The cache analysis described in § 3.1 does not support hierarchies of caches, i.e. all references are considered for the caches by the analysis. But, a memory reference may not occur in all the cache levels of the hierarchy. This filtering by previous caches in the hierarchy, impacts the cache contents.

Hence, caches are analysed one after the other, from cache level 1 to N, and *cache access classifications* (CACs), as defined in [9], are used to represent occurrences of a memory reference r on cache level L.

3.2.1 Cache access classifications (CAC)

Different classifications have been defined to represent if memory reference r is performed on cache level L:

– *Always* (A) means that the access r is always performed at cache level L,
– *Never* (N) means that the access r is never performed at cache level L,
– *Uncertain-Never* (U-N) indicates that no guarantee can be given considering the first access to each possibly referenced memory block for r, but next accesses are never performed at level L,
– *Uncertain* (U) indicates there is no guarantee that the access to r will or will not be performed at level L.

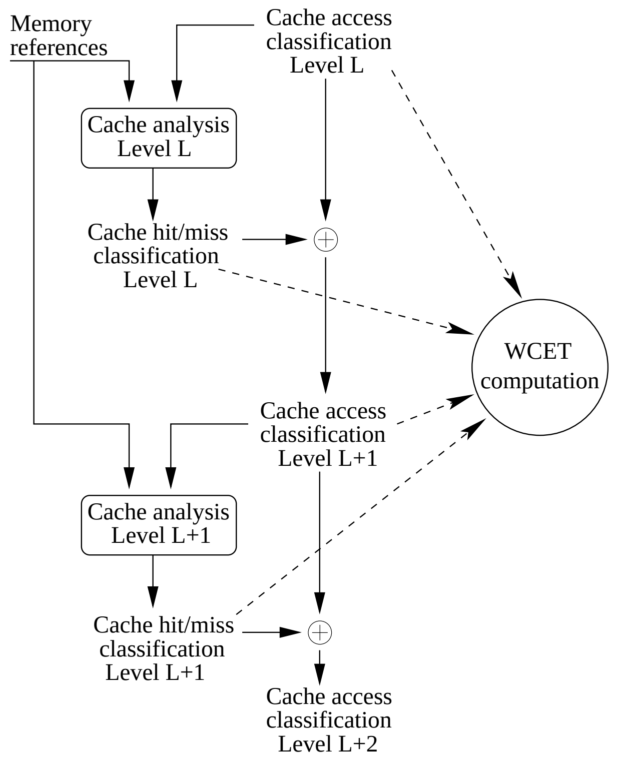

Memory references
Cache access classification Level L
Cache analysis Level L
Cache hit/miss classification Level L
WCET computation
Cache access classification Level L+1
Cache analysis Level L+1
Cache hit/miss classification Level L+1
Cache access classification Level L+2

Figure 3.3: Multi-level non-inclusive data cache analysis framework

The U and U-N classification is required as accessing to cache level L may not be the worst-case behaviour, as presented in [9].

For the L1 cache, CAC determination is simple as all memory references will be performed in this cache level (P1. § 2). This implies a A classification for every memory reference, considering the L1 cache.

Concerning greater cache levels, the CAC for memory reference r and cache level L is defined using both the CAC and the CHMC of the previous level cache level (see Figure 3.3) to safely model cache filtering in the cache hierarchy. This is illustrated in Table 3.1.

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

Table 3.1: Cache access classification for level L ($CAC_{r,L}$)

Cache level L in the hierarchy is only accessed if cache level $L-1$ is accessed and the access may be a miss in the $L-1$ cache level. If we do not know whether or not the access is a miss in the $L-1$ cache level, the access may or not be performed on cache level L.

If for reference r, the previous cache level, $L-1$, is never (N) accessed, neither will be the L cache level and its followers, whatever may be the classification of $L-1$ cache level. A AH classification at cache level $L-1$ filters accesses to subsequent cache levels the same way.

AM, on the other hand, does not filter any access; the CAC for cache level L will be the same than the CAC for cache level $L-1$ if the reference r always miss in cache level $L-1$.

Finally, FM and NC on cache level $L-1$ make information for the next cache level L imprecise, issuing respectively a U-N and U classification.

As CACs are meaningful only for load instructions, and the linked memory references, the CI categorization applied to store instructions is not considered.

3.2.2 Considering the CAC in the cache analysis

Once these classifications have been settled, they have to be considered in the multi-level cache analysis. The modifications to handle hierarchies of data caches are similar to the modifications to handle hierarchies of instruction caches [9]. A (respectively N) classification for a given instruction, means that the reference is (respectively not) performed on the considered cache level which should be modelled by the analysis. As for the U and U-N classifications, both the cases are possible: the access is performed (A) or not (N) on this cache level. Similarly to non deterministic accesses, the two alternatives are considered and their outcomes unified using the appropriate *Join* function of the analysis (see Figure 3.4).

The $Update_x$ function, $x \in \{Must, Persistence \text{ or } May\}$, is modified to consider these classifications:

$$Update_x^{ml}(ACS_{in}, r, L) = \begin{cases} Update_x(ACS_{in}, r, L) & \text{if } CAC_{r,L} = A \\ ACS_{in} & \text{if } CAC_{r,L} = N \\ Join_x(ACS_{in}, Update_x(ACS_{in}, r, L) & \text{if } CAC_{r,L} = \text{U-N} \vee CAC_{r,L} = \text{U} \end{cases}$$

where $Update_x^{ml}$ represents the multi-level version of the $Update_x$ functions presented earlier (§ 3.1). This definition can be inflected to $Update_{Must}^{ml}$, $Update_{Persistence}^{ml}$ or $Update_{May}^{ml}$ using the corresponding *Update* and *Join* functions of the Must, Persistence or May analysis respectively.

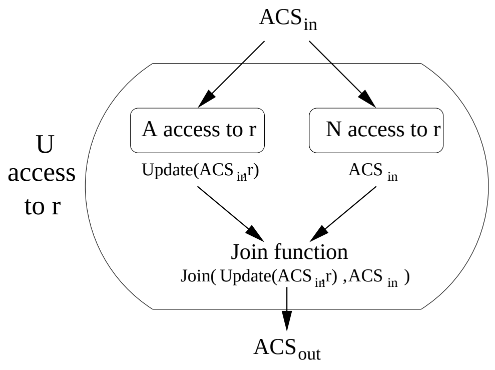


Figure 3.4: Handling U and U-N CAC in the cache analysis [9].

Termination of the analyses. The differences between the multi-level data cache analysis and the single-level data cache analysis are the $Update_x^{ml}$ functions, $x \in \{Must, Persistence \text{ or } May\}$, for the different analyses. However these functions were proved to be monotonic in Section 3.1. The proof in [9] thus holds for data caches as well.

We need to demonstrate that the $Update_x^{ml}$ function is monotonic for the four possible values of CAC.

Proof: For an A access, $Update_x$ and $Update_x^{ml}$ behave identically. $Update_x$ being monotonic, $Update_x^{ml}$ is also monotonic. Considering a N access, $Update_x^{ml}$ is the identity function and so is monotonic. Finally, considering an U or U-N access, $Update_x^{ml}$ composes $Update_x$ and $Join_x$. These two functions are monotonic, so is their composition. Thus $Update_x^{ml}$ is monotonic which guarantees the termination of our analysis. □

3.3 WCET estimation of tasks

CHMCs represent the worst-case behaviour of the cache given a memory reference. They are useful to compute the contribution of references to the WCET. This contribution can then be used in existing methods to compute the WCET. In our case, we focus on IPET based methods which estimates the WCET by solving an ILP problem [14].

The timing of the memory reference r, with regards to the data caches, is divided in two parts. $first$ and $next$ respectively distinguish the first and successive iterations of loops. We define $COST\_first(r)$ and $COST\_next(r)$ as the respective contribution to the WCET of memory reference r for the first and successive iterations of the loop in which the reference is enclosed, if any[1]. If no loop encloses r, $COST\_first(r)$ will be implicitly preferred, by the ILP solver, over $COST\_next(r)$ as the cost of r. Indeed $COST\_first(r)$ accounts for all first misses of memory reference r and $COST\_first(r) \geq COST\_next(r)$.

With $freq_r$ a variable computed in the context of IPET analysis and representing the execution frequency of r along the worst-case execution path of the task (like the x_i variables in section 1.1.2), the following constraints are defined:

$$freq_r = freq_{first,r} + freq_{next,r}$$

$$freq_{first,r} \leq 1$$

The WCET contribution of the reference r with regards to data caches is then defined as:

$$WCET\_data\_contribution(r) = COST\_first(r) \times freq_{first,r} + COST\_next(r) \times freq_{next,r}$$

As we focus on an architecture without timing anomalies, it is safe to consider that NC references behave like AM ones, which are the worst-case on our architecture. Therefore, U accesses to a cache level behave as A ones, from the timing analysis considering data caches point of view[2]. We define $always\_contribute(r)$ and $never\_contribute(r)$ as the sets of memory hierarchy levels which respectively always or never contribute to the execution latency of memory reference r :

$$never\_contribute(r) = \{L \mid 1 \leq L \leq M \wedge CAC_{r,L} = N\}$$

$$\begin{aligned} always\_contribute(r) = \quad & \{L \mid 1 \leq L \leq M \wedge CAC_{r,L} = A\} \bigcup \\ & \{L \mid 1 \leq L \leq M \wedge CAC_{r,L} = U\} \end{aligned}$$

with $M = N + 1$ the main memory in the memory hierarchy.

One option to deal with a FM classification for reference r on cache level L would be to consider, for every execution of r, that cache level $L+1$ is accessed: $CAC_{r,L} =$ U-N $\Rightarrow L \in always\_contribute(r)$. However, it might be overly pessimistic with regards to the semantic of the FM and the de facto inherited U-N classifications. We need additional notations to depict this behaviour and tighten r contribution to the WCET.

Given a memory reference r and a cache level L, let $memory\_blocks_{r,L}$ be the set of r target memory blocks on cache level L, as computed by the address analysis. This information is computed using the cache blocks size of cache level L. We only need to know a bound on the size of this set:

[1]Remember that r is contextual and attached to a unique instruction. A memory reference tied to the same instruction, but another context may be enclosed in different loops.

[2]From the ACS computing point of view, they keep different meanings.

$$|memory\_blocks_{r,L}| = \lceil \frac{Addr\_range_r}{cache\_block\_size_L} \rceil$$

with $Addr\_range_r$ the size of the address range computed by the address analysis for memory reference r and $cache\_block\_size_L$ the size of level L cache blocks.

Furthermore, we define max_freq_r as the maximum statically computable execution frequency of reference r. Such an information is computed as the product of the maximum number of iterations of all loops containing r :

$$max\_freq_r = \prod_{lo \in Loops(r)} max\_iter_{lo}$$

where $Loops(r)$ are the loop containing the memory reference r and max_iter_{lo}, the maximum iteration attribute for loop lo. Note that the whole task is itself considered to be a loop enclosing all memory references and whose $max\_iter = 1$. This ensures that the max_freq attribute of an otherwise not enclosed in a loop memory reference is 1.

Each element of $memory\_block_{r,L}$ produces at most one miss in the cache level L, the first time it is accessed, according to the definition of the FM classification. Thus, reference r will not produce more than $min(max\_freq_r, |memory\_blocks_{r,L}|)$ misses on cache level L.

Note that $min(freq_r, |memory\_blocks_{r,L}|)$, with $freq_r$ the execution frequency of r on the worst-case execution path of the task, would be a tighter bound. However, this bound is required to compute $freq_r$ and vice versa thus leading to the chicken-and-egg problem.

Once these elements have been defined, we can bound the number of occurrences of memory reference r on cache level L:

$$max\_occurrence(r,L) = \begin{cases} 0 & if\ L \in never\_contribute(r) \\ max\_freq_r & if\ L \in always\_contribute(r) \\ min(|memory\_blocks_{r,L-1}|, max\_occurrence(r,L-1)) & if\ CHMC_{r,L-1} = FM \\ max\_occurrence(r,L-1) & otherwise \end{cases}$$

Intuitively, if cache level L is never (respectively always) accessed by memory reference r, there cannot be any (respectively more than max_freq_r) occurrences of r on cache level L. Similarly, there cannot be more occurrences of r on cache level L than on cache level $L-1$. Finally, if $CHMC_{r,L-1} = FM$, only the first accesses to memory blocks belonging to $memory\_blocks_{r,L-1}$ will occur on cache level L.

The definition of $COST\_next(r)$ is pretty straightforward, as we only have to count the access latency of cache levels L which are guaranteed to always contribute to the timing of r. Other accesses are considered in $COST\_first(r)$:

$$COST\_next(r) = \begin{cases} \sum_{L \in\ always\_contribute(r)} ACCESS\_latency_L & \text{if r loads data from the memory} \\ STORE\_latency & \text{if r stores data in the memory} \\ 0 & \text{otherwise} \end{cases}$$

$COST\_first(r)$ is a bit more worrisome to define. In some cases, we have a bound on the number of occurrences of memory reference r on cache L. As previously stated, the solution of not considering these bounds might be unnecessary pessimistic. Therefore, we chose to use the $COST\_first(r)$ to hold these additional latencies. It could be understood as considering all the possible first misses in the first execution of r, in addition to all the always accessed cache levels latencies:

$$COST\_first(r) = \begin{cases} \sum\limits_{L \in \ always\_contribute(r)} ACCESS\_latency_L + \sum\limits_{CAC_{r,L} = \text{U-N}} ACCESS\_latency_L \times max\_occurrence(r,L) & \text{if r loads data from the memory} \\ STORE\_latency & \text{if r stores data in the memory} \\ 0 & \text{otherwise} \end{cases}$$

Chapter 4

WCET estimation in multi-core architectures

The multi-level data caches analysis, proposed in Chapter 3, was introduced to be used (as is or refined when required) in the analysis of multi-core architectures with shared data caches. To compute these WCET estimates, are now introduced different methods. These methods address the fact that using such a multi-core context the cache may be trashed at any time by a task sharing it with the analysed task; *any rival task* may *any time* alter the cache contents.

The first proposed approach is to preclude multi-core-induced inter-task conflicts using partitioning, computed by an algorithm presented in Section 4.1. On the other hand, Section 4.2 introduces refinements of the aforementioned multi-level data cache analysis to include estimated conflicts. Finally, bypass heuristics are defined (§ 4.3) to be later combined with the data cache analysis and tighten our estimates.

4.1 Partitioning the shared caches

To study the usability of partitioning in the context of multi-core architectures with shared data caches, the *dynamic programming algorithm* defined in [25] is used. Actually, this method benefits from a high genericity, allocating so-called *memory segments* (sets or ways of a cache, main memory...) with the objective of minimizing a metric (the system utilization in the original contribution). Furthermore, it is exact it the sense that it computes the best possible allocation of memory segments to tasks with regards to the given metric. Finally, considering the cache sizes we use, it remains tractable.

The problem of computing an optimal partitioning with regards to a given metric is similar to the *knapsack problem*. A cache can hold a limited number of partitions. Each couple partition/task has an impact on the metric. The objective, as depicted in Algorithm 2, is to minimize the chosen metric, without allocation more memory segments than available in the cache.

Thereafter, we assume a set-based partitioning with partitions allocated on a per-tasks basis. For the sake of partitioning implementation, a partition is further assumed to hold a number of sets which is a power of two. Finally, we will use the sum of the WCETs of all the tasks in the system as our metric. This is equivalent to a periodical load with each task having the same period. Given a memory segment allocation, the WCET of a task is computed using our multi-level data cache analysis or its refinements including bypass.

4.2 Data cache analysis with full interferences for shared data caches

Partitioning relies on tasks isolation to prevent inter-task conflicts and computes tasks WCETs using the multi-level data cache analysis previously defined. Methods like [7, 41], for instruction caches, on the other hand relies on the estimation of conflicts between rival tasks to update memory references classifications and estimate a safe but worsen WCET for a task.

Algorithm 2 Pseudo-algorithm for cache partitioning

```
Function computePartitions(Task t, Integer available_segments)
$best\_value \leftarrow \infty$
$best\_allocation \leftarrow \emptyset$
//Recall the computed value if it exists
if $\neg computed(t, available\_segments)$ then
  for $partition \in valid\_partitions(available\_segments)$ do
    $allocation \leftarrow computePartitions(t-1, available\_segments - partition)$
    if $value(allocation) < best\_value$ then
      $best\_allocation = allocation$
      $best\_value = value(allocation)$
    end if
  end for
  //Keeping the computed value for later use
  $computed(t, available\_segments) \leftarrow best\_allocation$
else
  $best\_allocation \leftarrow computed(t, available\_segments)$
end if
return $best\_allocation$
```

As [41] does not yet support multiple rival tasks, we use [7] as a basis for extensions of the aforementioned multi-level data cache analysis. First, conflicts stemming from data cache sharing are identified (§ 4.2.1). Then, conflicts are integrated in the multi-level data cache analysis (§ 4.2.2) to compute the WCET of the task.

4.2.1 Estimation of cache conflicts

A first step to perform data cache analysis while taking all interferences into account, in a multi-core context, is to compute inter-task induced conflicts. They are the kind of conflicts that stem from the sharing of a cache with other tasks. The idea is to estimate how and when is the shared cache used by these rival tasks. Once this information is known, or a safe estimate has been computed, we are able to consider its impact on cache contents during the shared cache analysis.

A first solution to estimate cache conflicts would be to study the interleaving of all the memory references of rival tasks with memory references of the analysed task. This would give the most precise information, but the required computing power would be prohibitive in terms of time and space. [41], in the context of instruction caches, abstracts the rival task memory references sequencing, keeping only their position (inside or outside loops). Cache classifications are then updated according to the rival memory references which map in the same cache set and the respective positions of both the rivals and the analysed memory references. However, the proposed solution considers only one rival task.

Instead, based on [7], we abstract both the accesses sequencing and positions in the rival tasks. Only a set of the memory blocks used by rival tasks and possibly reaching cache level L is kept. With the help of these abstractions, we can consider, in the data cache analysis, that any of these rival memory blocks may be accessed any time while the analysed task is executed.

Furthermore, a rival memory block may only conflict with memory blocks of the task mapping to the same cache set s. The set of $conflict\_blocks_{i,L}(s)$ for set s of cache level L while considering task i is defined as:

$$\begin{aligned} b \in conflict\_blocks_{i,L}(s) \Leftrightarrow \quad & SET(b) = s \wedge \\ & (\exists t \neq i, \exists r, 1 \leq t \leq T \wedge r \in memory\_references_t \wedge \\ & b \in memory\_blocks_{r,L} \wedge CAC_{r,L} \neq N) \end{aligned}$$

where T is the number of tasks of the considered system (including i the analysed task) and $memory\_blocks_{r,L}$ the range of memory blocks accessed by r on cache level L.

This means that memory block b has to be considered as a conflicting block for task i on cache level L, if there is another task t whose accesses to memory block b might access cache level L. In addition, we do not need to know precisely the memory blocks that may cause conflicts in the cache, only the *cache block conflict number* (CCN) is kept for each set s of cache level L:

$$CCN_{i,L}(s) = |conflict\_blocks_{i,L}(s)|$$

Note that due to indeterministic accesses in the rival tasks, without any mechanism to reduce the number of memory references performed on each cache level, all memory blocks have to be accounted for in the CCN. This may be pessimistic as in fact only one is accessed. Pessimism is further increased as the number of occurrences of references to each memory block is forgotten.

4.2.2 Accounting for interferences in the data cache analysis

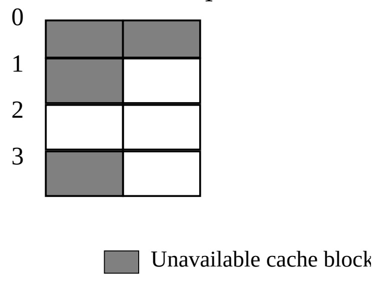
Once the $CCN_{i,L}$ for task i and cache level L is known, data cache analysis of level L is performed as usual using the previously defined multi-level data cache analysis. However, the manipulated ACSs are modified to take conflicts in consideration. For each set s of the ACS, $CCN_{i,L}(s)$ cache blocks are supposed to be unusable during the task analysis; only $associativity_L - CCN_{i,L}(s)$ cache blocks are available, with $associativity_L$ the associativity degree of cache level L.

Example To clarify this definition, we will show how CCNs are considered using a brief example. Consider a unique L1 shared data cache, the $conflict\_blocks$ for the *analysed* task have been computed:

$$conflict\_blocks_{analysed,1} = \begin{cases} 0 \rightarrow \{a, e, c\} \\ 1 \rightarrow \{d\} \\ 2 \rightarrow \emptyset \\ 3 \rightarrow \{b\} \end{cases}$$

The available cache space, during the data cache analysis for the *analysed* task, under these conditions is presented in 4.1.

Figure 4.1: Available cache space considering conflicts



4.3 Bypassing memory references

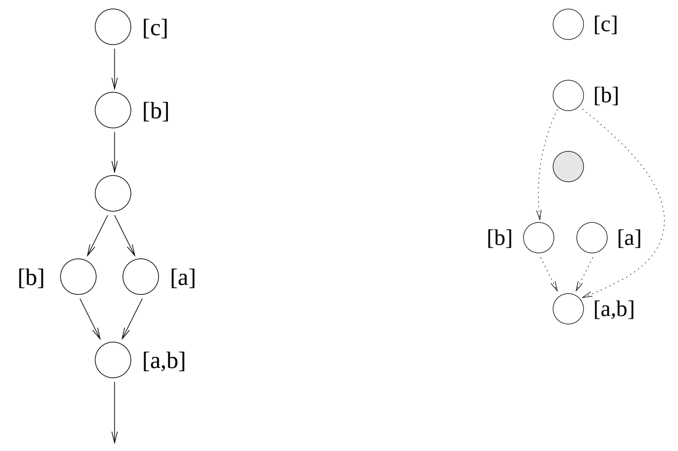
Bypassing in the context of caches is the absence of cache contents alteration upon a memory reference. A memory reference bypassing a cache level will still look for information in this cache level but neither the cache contents (in case of a miss in the cache) nor the cache blocks ages (in case of a hit in the cache) will be modified. By preventing the occurrences of memory references on given cache levels, bypass can be used as a mean to reduce cache conflicts whether they are intra-task or inter-tasks.

The former use is interesting in the context of partitioning as small partition sizes and an increased number of memory blocks mapping to the same sets might increase the amount of conflicts. The latter use benefits to data cache analysis accounting for conflicts by reducing the number of conflicting blocks. Still, to achieve such conflicts reductions, one has to carefully choose how, when and to which extent should bypass be used.

In our case, for our bypass mechanism to remain realistic, we assumed, instead of a per-memory reference decision, a per-load instruction decision on which cache levels should be bypassed. Thus the bypassing is based on a non contextual decision. For the time being, our heuristics (§ 4.3.2) reach local decisions based on the impacts of memory references linked to an instruction on its neighbours, i.e. the following instructions in the CFG accessing the same data. To gather information about a load instruction neighbourhood, we compute its dependencies (§ 4.3.1). Only load instructions are considered as they are the only one impacting the cache contents. Finally, once a decision was reached for each load, a data cache analysis is performed taking into account these decisions (§ 4.3.3).

4.3.1 Loads dependency computation

An example of the requested links, for the CFG of Figure 4.2a, is illustrated in 4.2b. As there is only one memory reference to memory block c, it has no possible next reference. Similarly, the last memory reference to blocks a or b, has no successor. On the other hand, the first reference to b impacts the results of the other one or may impact the behaviour, with regards to the cache, of the last memory reference.



(a) CFG of a task with highlighted memory references

(b) Loads dependency computed for tasks in 4.2a

Figure 4.2: Example of loads dependency computing on a task.

This first reference to memory block b loads this block in the cache. If the second memory reference to b hits in the cache, it may be thanks to this first load. The first memory reference to b impacts the second one, they are linked. Similarly, the first reference to a, if executed, possibly impact the last reference of the task if a is accessed. Conversely, the memory reference to c inserts in the cache a block that will not be reused, this reference is not linked to any other reference.

The objective of the loads dependency computation step is to build, this kind of information for each instruction impacting on the cache contents. We want to know the set of memory references whose behaviour might depend on this instruction. Provided this information, one can locally decide for each instruction which cache levels it should bypass.

For each instruction, we compute $next\_loads_L(inst)$ the set of directly following load references:

$$next\_loads_L(inst) = \{r' \,|\exists r, b, r \in memory\_references(inst) \wedge b \in memory\_blocks_{r,L} \wedge r' \in next\_reference_L(r, b)\}$$

with $memory\_references(inst)$ the set of memory references tied to $inst$ instruction and $memory\_blocks_{r,L}$ the range of memory blocks accessed by memory reference r on cache level L. $next\_reference_L(r, b)$ can be seen as a contextual (per-reference information) extension of $next\_loads_L(inst)$.

For each memory reference r to memory block b issued by a *load* instruction, $next\_reference_L(r, b)$ is computed using a backward data flow analysis. It represents the set of directly following loads references to b. Memory references issued from store instructions are not considered as they have no impact on the cache contents.

Therefore, memory reference r' belongs to $next\_reference_L(r, b)$ if it possibly loads memory block b and, if there is a path in the CFG from r to r' without any possible reference to b. More formally:

$$\begin{aligned} next\_reference_L(r,b) = \quad & \{r' | b \in memory\_blocks_{r',L} \wedge r' \text{ is a load} \\ & \wedge(\exists P, P = r, r_1, ..., r_n, r', (\forall i, 1 \leq i \leq n, b \notin memory\_blocks_{r_i,L}))\} \end{aligned}$$

The data-flow equations defined for this computation depends on $NR_{in}(r, L)$ and $NR_{out}(r, L)$ which are functions of : $memory\_blocks \times \mathcal{P}(memory\_references)$; given a memory block, $NR_{out}(r, L)$ and $NR_{in}(r, L)$ yield a set of memory references. $NR_{in}(r, L)$ and $NR_{out}(r, L)$ are the sets of next memory references respectively before and after memory reference r. Actually, $NR_{out}(r, L)(b)$ is equivalent to $next\_reference_L(r, b)$:

$$NR_{in}(r, L) = (NR_{out}(r, L) \backslash kill(r)) \cup gen(r)$$

$$NR_{out}(r,L) = \begin{cases} \{b \rightarrow \emptyset | r' \in memory\_references \wedge b \in memory\_block_{r',L}\} & \text{if r is the last memory reference of the task} \\ \bigcup\limits_{succ \in Successors(r)} NR_{in}(succ, L) & otherwise \end{cases}$$

$Successors(r)$ is defined as the set of memory references following r according to the CFG and $memory\_references$ as the set of all memory references of the analysed task. Then $gen(r)$ and $kill(r)$ are defined for all memory references r as the sets of elements created or erased respectively by r :

$$gen(r, L) = \{b \rightarrow \{r\} \mid b \in memory\_block_{r,L} \wedge CAC_{r,L} \neq N\}$$

$$kill(r, L) = \{b \rightarrow R \mid b \in memory\_block_{r,L} \wedge CAC_{r,L} \neq N \wedge R \in \mathcal{P}(memory\_references)\}$$

4.3.2 Bypassing heuristics

The choice of bypassing or not a cache level is not trivial as it might have repercussions on the following memory references. Exploring all the possible solutions, bypassing or not for each load instruction (as they are the only ones impacting cache contents), would be too expensive. Instead we define heuristics to reach local decisions for each load instruction. The first and second heuristics consider the close environment of instructions, in terms of memory references, to reach such a decision. The third heuristic on the other hand only relies on the results of the address analysis. Taking into account bypass decisions in the data cache analysis is introduced afterwards.

$Bypass_x(inst, L)$ will be set to $true$ if instruction $inst$ bypasses cache level L, according to heuristic x. Conversely, $Bypass_x(inst, L) = false$ if instruction $inst$ do not bypass cache level L.

Conservative bypass. First, using the *conservative bypass* (CB) method instruction *inst* bypasses cache level L if there is no memory reference belonging to $next\_loads_L(inst)$ which hits in cache level L:

$$Bypass_{CB}(inst, L) = (\forall r \in next\_loads_L(inst), CHMC_{r,L} \neq AH \wedge CHMC_{r,L} \neq FM)$$

This heuristic is conservative in the sense that it choose not to bypass a cache level if related memory references may hit in the cache (and not only miss). In its current state, CB could also be defined using the notion of *live cache blocks* [12]. This might not be true if CB definition is refined to make a better use of the information about instructions neighbourhood. Yet, for the time being, CB bypasses cache level L for instruction *inst* if no memory reference issued from *inst* accesses a live cache block:

$$Bypass_{CB}(inst, L) = (\forall r \in memory\_references(inst), \forall b \in memory\_blocks_{r,L}, b \notin LCB_L(r))$$

where $LCB_L(r)$ is the set of lived cache blocks after memory reference r on cache level L. A lived cache block is a memory block, that may be in the cache after reference r, and may be reused before it is evicted from the cache. Therefore, CB ensures that a $Bypass_{CB}(inst, L) = true$ has no aggravating effect on the computed WCET.

Aggressive bypass. *Aggressive bypass* (AB) is kind of the opposite of CB since instruction *inst* bypasses cache level L if there is at least one following memory reference which do not hit in the cache:

$$Bypass_{AB}(inst, L) = (\exists r \in next\_loads_L(inst), CHMC_{r,L} = AM \vee CHMC_{r,L} = NC)$$

Indeterministic bypass. Finally, not relying on the computation of the *next_loads* relations is the *Indeterministic bypass* (IB). Instruction *inst* bypasses cache level L if the address analysis yields a range of memory blocks for one of the reference linked to *inst*, opposed to a single target memory block:

$$Bypass_{IB}(inst, L) = (\exists r \in memory\_references(inst), |memory\_blocks_{r,L}| > 1)$$

4.3.3 Data cache analysis with bypassed cache levels

Once the instructions have been set to bypass some cache levels, a new multi-level data cache analysis has to be performed, while considering bypass, to update memory references classifications. The effect of an instruction bypassing cache level stem during the ACSs computation step. Given memory reference r, cache level L and an input ACS ACS_{input}, if the instruction tied to r bypasses cache level L, ACS_{input} is left unchanged. Otherwise, ACS_{input} is updated by memory reference r as defined previously in our multi-level data cache analysis (§ 3.2). From the ACSs computing point of view, memory references whose linked instruction bypasses the analysed cache level behaves like memory references whose $CAC = N$.

Chapter 5

Experimental results

Two main methods to allow the inclusion of shared data caches in the timing analysis of multi-core architectures have been presented. Partitioning on the one hand, precludes multi-core related inter-tasks conflicts. On the other hand, data cache analysis has been refined to include these conflicts. These methods to tackle inter-task conflicts in the context of multi-core architectures with shared data caches both rely on our previously introduced multi-level data cache analysis. Then, bypass was introduced as a possible mean to reduce inter-task conflicts.

We want to be able to discuss the relevancy and relative interests of each method. First, focusing on uni-core architecture, merits of the aforementioned multi-level data cache analysis are analysed as well as the impacts of bypass on intra-task conflicts (§ 5.2). Then, in a multi-core context, the respective interests partitioning and multi-level data cache analysis accounting for conflicts are analysed (§ 5.3). In a first stage, the experiments are conducted without the help of bypass to reduce conflicts.

5.1 Experimental setup

Cache analysis and WCET estimation. The experiments were conducted on MIPS R2000/R3000 binary code compiled with gcc 4.1 with no optimization. The default linker memory layout is modified to ensure that the different sections of the task are aligned on 512B boundaries, to fulfil requirements of our measurement environment.

The WCETs of tasks are computed by the Heptane timing analyser [2], more precisely its Implicit Path Enumeration Technique (IPET). The timing of memory references with regards to the L1 and L2 data cache is evaluated using the method introduced in Section 3.3. Modules have also been developed for Heptane to handle cache partitions computation, bypassed instructions computation and multi-level data cache analysis. 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 caches to the WCET. The effects of other architectural features are not considered. In particular, timing anomalies caused by interactions between caches and pipelines, as defined in [16] are disregarded. The cache classification *not-classified* is thus assumed to have the same worst-case behaviour as *always-miss* during the WCET computation in our experiments. The cache analysis starts with an empty cache state.

Benchmarks. The experiments were conducted on a subset of the benchmarks maintained by Mälardalen WCET research group[1] commonly used to evaluate WCET estimation techniques. Table 5.1 summarizes the characteristics characteristics (size in bytes of sections *text*, *data*, *bss* (uninitialized data) and *stack*).

Cache hierarchy. The results are obtained on a 2-level cache hierarchy composed of a private 4-way L1 data cache of 1KB with a cache block size of 32B and a shared 8-way L2 data cache of 4KB with a cache block size of 32B. A perfect private instruction cache, with an access latency of one cycle, is

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

Name	Description	Code size (bytes)	Data size (bytes)	Bss size (bytes)	Stack size (bytes)
crc	Cyclic redundancy check computation	1432	272	770	72
fft	Fast Fourier Transform	3536	112	128	288
insertsort	Insertsort of an array of 11 integers	472	0	44	16
jfdctint	Fast Discrete Cosine Transform	3040	0	512	104
ludcmp	Simultaneous Linear Equations by LU Decomposition	2868	16	20800	920
matmult	Product of two 20x20 integer matrixes	1048	0	4804	96
minver	Inversion of floating point 3x3 matrix	4408	152	520	128
ns	Search in a multi-dimensional array	600	5000	0	48
qurt	Root computation of quadratic equations	1928	72	60	152
sqrt	Square root function implemented by Taylor series	544	24	0	88
statemate	Automatically generated code by STARC (STAtechart Real-time-Code generator)	8900	32	290	88

Table 5.1: Benchmark characteristics

assumed. 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). Upon store instructions a latency of 150 cycles is assumed to update the whole memory hierarchy.

Measurement environment. Simulation is used to evaluate the tightness of WCET estimates when possible, i.e. in uni-core architectures. The measure of the cache activity on a worst-case execution scenario uses the Nachos educational operating system[2], running on top of a simulated MIPS processor. We have extended Nachos with a two-level data cache hierarchy with LRU replacement policy at each level.

5.2 Uni-core environment

Considering the cache hierarchy. To evaluate the interest of considering the whole cache hierarchy, two configurations are compared in Table 5.2. In the first configuration, only the L1 cache is considered and all accesses to the L2 cache are defined as misses ($WCET_{L1}$, column 1). In the other configuration ($WCET_{L1\&L2}$, column 2), both the L1 and the L2 caches are analysed using our multi-level static cache analysis (§ 3.2).

Table 5.2 presents these results for the two cases using the tasks WCET as computed by our tool and based on our multi-level static analysis. The presented WCET are expressed in terms of cycles.

The third column of Table 5.2 represents the improvement, for each analysed task, attributable to the consideration of the second cache level of the hierarchy:

$$\frac{WCET_{L1} - WCET_{L1\&L2}}{WCET_{L1}}$$

Considering the L2 cache using our multi-level static analysis, may lead up to 45.20% improvement for the *fft* task. In average, the results are still interesting 6.06% with the exceptions of a subset of tasks not taking any benefit in the consideration of the L2 cache (*crc*, *insertsort*, *ludcmp*, *matmult*, *ns* and *sqrt*).

The issues raised by these tasks are twofold. On the one hand, tasks like *sqrt* and *insertsort* use little data. This small working set fits in the L1 cache. Thus, the L2 cache is only accessed upon the very first misses.

On the other hand are tasks like *ludcmp*, *matmult* and *ns* accessing large amount of data that fit neither in the L1 nor the L2 cache. *ludcmp* accesses a small portion of a large array of floats, which

[2]Nachos web site, http://www.cs.washington.edu/homes/tom/nachos/

Benchmark	**WCET considering only a L1 cache** (cycles)	**WCET considering both L1 and L2 cache** (cycles)	**Improvement considering the L2 cache**
crc	2627460	2627460	0.00 %
fft	4216150	2310240	45.20 %
insertsort	72547	72547	0.00 %
jfdctint	176953	170653	3.56 %
ludcmp	543949	543949	0.00 %
matmult	7680840	7680840	0.00 %
minver	94933	91933	3.16 %
ns	273209	273209	0.00 %
qurt	171526	148126	13.64 %
sqrt	27080	27080	0.00 %
statemate	374999	370949	1.08 %

Table 5.2: Static multi-level n-way analysis (4-way L1 cache, 8-way L2 cache, cache sizes of 1KB (resp. 4KB) for L1 (resp. L2)).

probably shows temporal locality. However, this information is lost as accesses to the different memory blocks composing the array are not ordered in time in our analyses. Concerning *matmult* and *ns*, they consist of loop nests running through big arrays. Again, there is temporal locality between the different iterations of the most nested loop. However, this locality is not captured as the FM classification applies to the outermost loop in such cases, and thus consider the whole array as being persistent or not.

Other tasks exhibit such small data set but, due to the analysis pessimism, some memory blocks might be considered as shifted outside the L1. However, they are kept in the L2 (*fft*, *jfdctint*, *minver*, *qurt*, *statemate* and to a lesser extent *crc*).

These elements tend to show that hierarchies of data caches can be safely analysed and that in average, analysis yields results which are tighter than considering only the L1 cache alone. Even better, the results are never worse while considering the L2 cache than while considering only the L1 cache.

Analysis precision. In order to estimate the tightness of the data cache analysis and the multi-level analysis, static analysis results are compared with those obtained by executing the programs in their worst-case scenario. However, due to the difficulty of identifying the worst-case input data, we only use the simplest benchmarks (*jfdctint*, *matmult* and *ns*). All these benchmarks are single-path programs, i.e. without potential flow dependency on input variables of the task.

The measured and estimated worst-case execution time would not be informative enough to discuss the precision of our analysis. For each benchmark (column 1), for the measurement (column 3) and the static analysis (column 4), different metrics (column 2) have been extracted and presented in Table 5.3:

– The contribution (in cycles) of data caches to the WCET of the benchmark (line 1)
– The number of executed memory references on each level of the memory hierarchy, meaning the L1 (line 2), the L2 (line 3) cache levels and the memory (line 4) represented by the number of references to the L1 cache and the number of considered misses in both the L1 and L2 cache.

The ratio in column 4 represents the increase of the estimated WCET, $WCET_{est}$, over the measured one, $WCET_{mea}$. It is computed as such :

$$\frac{WCET_{est} - WCET_{mea}}{WCET_{est}}$$

Note that, according to the analysis results presented in Table 5.2 the three benchmarks were supposed to exhibit nearly no use of the L2 cache. In fact, according to our measurements, these benchmarks exhibits a very good temporal locality. The poor precision analysis we attain has at least two sources :

– the address analysis is not precise enough concerning arrays. When a memory reference accesses an array, its full address range is attached to the reference. However, as in *jfdctint*, no distinction is made between a reference that might hit many elements in the array and a reference that

Benchmark	Metrics	Measurement	Static analysis	WCET increase
jfdctint	WCET contribution of data caches	129753	170653	23.96 %
	nb of L1 accesses	1444	1444	
	nb of L1 misses	12	307	
	nb of L2 misses	12	265	
matmult	WCET contribution of data caches	3878815	7680840	49.50 %
	nb of L1 accesses	114191	114191	
	nb of L1 misses	1568	24006	
	nb of L2 misses	155	24006	
ns	WCET contribution of data caches	185689	273209	32.03%
	nb of L1 accesses	5468	5468	
	nb of L1 misses	81	628	
	nb of L2 misses	81	628	

Table 5.3: Precision of the multi-level n-way analysis (4-way L1 cache, 8-way L2 cache, cache sizes of 1KB (resp. 4KB) for L1 (resp. L2).

hits only a precise element of the array. This first source of imprecision is further illustrated in Table 5.4 using the task *sqrt* which has no indeterministic memory references. In this context, the number of estimated misses in the L1 and L2 cache is very close the measured number of misses. Note that the difference in the number of L1 accesses comes from the fact that the worst-case input for sqrt is not well-defined.

– temporal locality is poorly captured as illustrated by the examples of *ns* and *matmult*. Accesses to the arrays used by these tasks can not be classified as FM or AH, as it would require the whole array to fit in the cache for such a classification. This behaviour does not suit to big arrays. Actually, considering a regular array run as the one in *ns* there may be more hits in the cache than misses. Two elements acceded in successive iterations are either in neighbour memory blocks or in the same memory block. And indeed, considering a big enough L1 cache, we have very tight estimates for *ns* has presented in Table 5.5.

Note that *ns* accesses each array element only once, in a regular way. Which means that memory blocks are loaded only once from the main memory and are not reused once they have been evicted from the cache. This might give a trail for an extension of the FM classification definition.

Benchmark	Metrics	Measurement	Static analysis
sqrt	nb of L1 accesses	179	371
	nb of L1 misses	5	7
	nb of L2 misses	5	7

Table 5.4: Static multi-level n-way analysis (4-way L1 cache, 8-way L2 cache, cache sizes of 1KB (resp. 4KB) for L1 (resp. L2)) compared with measurements for a task without indeterministic memory references.

Benchmark	Metrics	Measurement	Static analysis
ns	nb of L1 accesses	5468	5468
	nb of L1 misses	81	82
	nb of L2 misses	81	82

Table 5.5: Static multi-level n-way analysis (4-way L1 cache, 8-way L2 cache, cache sizes of 32KB (resp. 4KB) for L1 (resp. L2)) compared with measurements.

The main source of differences between measured and estimated WCET imputes to access indeterminism. These differences tend to underline the limits of pure analysis for this kind of accesses. These elements lead us to the experimentations of bypass in the context of uni-core architectures.

5.2.1 Bypassing effects on uni-core architectures

Originally introduced with the objective of reducing inter-task conflicts in the context of multi-core architectures, we want to see if bypass could not be applied on uni-core architectures to obtain tighter WCET estimates, by reducing intra-task conflicts. Four different configurations are presented in Table 5.6. Similarly to results about the analysis precision, the WCET of the tasks (in cycles, line 1) is not the only concern and is presented along with other metrics :

- the number of instructions (line 2) and its restriction to load instructions only (line 3),
- the number of instructions bypassing the L1 (line 4) and/or the L2 (line 5) cache.

The configurations exposed in Table 5.6 range from the use of no bypass heuristic (column 2), which means that no instruction is bypassed to the use of the IB bypass heuristic (column 6). Heuristics CB and AB using the aforementioned load dependency graph are also presented (column 4 and 5 respectively).

Concerning the strict number of instructions bypassing caches L1 or L2, the results are quite intuitive in the sense that AB, the most aggressive heuristic, bypasses more instructions than CB and IB. Since few benchmarks have data structures which fit in the cache, most of the indeterministic accesses to these structures are classified as NC or AM. A reference in a loop which runs through an array will probably depends on itself ($r \in next\_reference(r)$, with $next\_reference$ defined in § 4.3) and if classified as NC or AM, will be bypassed.

However, even using this aggressive strategy, the WCET do not increase for any of the benchmarks. Actually, considering *fft*, *minver*, *qurt* and *statemate*, both CB and AB improves the WCET. IB has a similar behaviour on these tasks with the exception of *minver*. This comes from the point stated earlier: imprecise references to big data structures have a strong probability of being bypassed. Bypassing such references means that more room is left in the cache for other memory blocks.

ludcmp and *ns* reveal a problematic aspect of the bypass heuristics defined so far. Because of references to massive arrays, no reference hits in the L2 cache. Intuitively, accesses to these array are bypassed from the cache. However, once these accesses have been bypassed, the CHMCs of memory references are not revalued and all loads instructions bypasses the L2 even if they would fit in the cache with the decision taken so far.

Finally, the number of bypassed references for each cache level may not be the same using the IB. This behaviour is normal seeing that if an instruction has no reference possibly accessing a cache level, this cache level is not bypassed.

Thus bypass seems interesting to reduce intra-task conflicts, thus allowing tighter WCET estimates, even in the context of uni-core architectures with hierarchies of data caches. It may be a track to follow to balance data caches analysis pessimism induced by imprecise accesses.

5.3 Multi-core environment

We will now consider three different systems composed of set of tasks, as depicted in Figure 5.1, with various behaviour with regards to the L2 cache. Each task is assumed to run on a different core. The underlying idea is to evaluate the respective interests of partitioning and multi-level data cache analysis while accounting for conflicts.

(a) *Light task set*, the tasks fits in the L2 cache.	(b) *Heavy tasks set*, tasks with large working set size.	(c) *Mixed task set*, tasks from *Heavy* and *Light*.
jfdctint	ludcmp	fft
minver	ns	ns
qurt	matmult	qurt
statemate		matmult

Figure 5.1: Considered systems in our multi-core environments.

In a first stage (§ 5.3.1), bypass will not be used to reduce conflicts. To ensure fairness in the

Benchmark	Metric	Without Bypass	Using CB	Using AB	Using IB
crc	WCET contribution of data caches	2627460	2627460	2627460	2627460
	nb instructions	358	358	358	358
	nb loads	60	60	60	60
	nb L1 bypassing loads	0	5	6	2
	nb L2 bypassing loads	0	34	42	8
fft	WCET contribution of data caches	2310240	2293170	2227970	2227970
	nb instructions	884	884	884	884
	nb loads	238	238	238	238
	nb L1 bypassing loads	0	78	85	74
	nb L2 bypassing loads	0	32	53	33
insertsort	WCET contribution of data caches	72547	72547	72547	72547
	nb instructions	118	118	118	118
	nb loads	15	15	15	15
	nb L1 bypassing loads	0	0	0	1
	nb L2 bypassing loads	0	0	0	1
jfdctint	WCET contribution of data caches	170653	170653	170653	170653
	nb instructions	760	760	760	760
	nb loads	146	146	146	146
	nb L1 bypassing loads	0	2	2	0
	nb L2 bypassing loads	0	1	1	0
ludcmp	WCET contribution of data caches	543949	543949	543949	519049
	nb instructions	717	717	717	717
	nb loads	169	169	169	169
	nb L1 bypassing loads	0	16	16	8
	nb L2 bypassing loads	0	169	169	40
matmult	WCET contribution of data caches	7680840	7680840	7680840	7680840
	nb instructions	262	262	262	262
	nb loads	45	45	45	45
	nb L1 bypassing loads	0	2	2	0
	nb L2 bypassing loads	0	33	33	3
minver	WCET contribution of data caches	91933	84733	84733	91933
	nb instructions	1102	1102	1102	1102
	nb loads	263	263	263	263
	nb L1 bypassing loads	0	3	3	2
	nb L2 bypassing loads	0	16	16	3
ns	WCET contribution of data caches	273209	273209	273209	273209
	nb instructions	150	150	150	150
	nb loads	26	26	26	26
	nb L1 bypassing loads	0	3	3	1
	nb L2 bypassing loads	0	26	26	3
qurt	WCET contribution of data caches	148126	145826	145446	145286
	nb instructions	482	482	482	482
	nb loads	117	117	117	117
	nb L1 bypassing loads	0	26	32	25
	nb L2 bypassing loads	0	79	82	27
sqrt	WCET contribution of data caches	27080	27080	27080	27080
	nb instructions	136	136	136	136
	nb loads	29	29	29	29
	nb L1 bypassing loads	0	1	1	0
	nb L2 bypassing loads	0	4	4	0
statemate	WCET contribution of data caches	370949	359749	359749	359749
	nb instructions	2225	2225	2225	2225
	nb loads	397	397	397	397
	nb L1 bypassing loads	0	65	65	26
	nb L2 bypassing loads	0	149	149	26

Table 5.6: Effects of different bypass heuristics on static multi-level n-way cache analysis (4-way L1 cache, 8-way L2 cache, cache sizes of 1KB (resp. 4KB) for L1 (resp. L2)) considering an uni-core architecture.

comparison between partitioning and data cache analysis while considering conflicts, we will consider that all tasks are periodic with identical periods. Then, bypass is combined with the methods in the objective of tightening WCET estimates of the tasks by reducing conflicts(§ 5.3.2).

Note that no comparison with measured WCET is performed in the remainder of this chapter, dealing with multi-core architectures, as the worst-case behaviour of such an architecture is a lot harder to define (and thus measure) than in a uni-core context as all interleaving between the tasks of a system would be tested.

5.3.1 Multi-core without bypass

This section aims at estimating the respective benefits of partitioning and data cache analysis with full interferences while using a shared data cache in a multi-core system. In this first step, no additional mechanism will be used to reduce inter or intra tasks conflicts.

Table 5.7 introduces for each system, for each task of this system, its WCET (line 1 for each task) when using partitioning and when analysing the system while considering full conflicts. In addition to these estimated WCETs, the number of L1 accesses and L1 and L2 misses are presented on line 2, 3 and 4 respectively for each task.

First of all, the *Heavy* task set behaviour is not altered by either partitioning or static cache analysis considering full conflicts. The tasks behaves as if they had no L2 data cache. Indeed, none of these tasks benefits from the L2 cache in uni-core environments. The things are the same in multi-core one, without any mechanism to reduce conflicts. With a high pressure on the L2 cache, there is no difference.

Then considering *Light*, the integration of conflicts in the multi-level data cache analysis seem to be the most interesting. Tasks have enough space in the L2 cache not to be bothered by the other. Concerning partitioning on the other hand, the results are not as good. The smaller available cache sizes worsen the WCET estimates of *jfdctint* and *minver*, compared to a uni-core architecture without shared caches. With such a low pressure on the L2 cache, the analysis performs better than the partitioning.

Conversely, the WCET estimates using our multi-level data cache analysis on a partitioned shared cache with the *Mixed* task set are better, at least for *fft* than the ones achieved by analysing the tasks and taking all conflicts in consideration. Note that fft, in the context of a partitioned cache gets all the memory segments because of its massive WCET improvement with big partition sizes.

In summary, considering small systems or big enough caches, it seems more interesting to analyse data caches while taking conflicts into account. Partitioning, because of the small partition sizes allocated to each task, may indeed worsen the estimated WCET. Conversely, partitioning suits better to systems with tasks generating many conflict, like *mixed*. Because these mechanisms mainly target inter-task conflicts, the use of the L2 cache is not improved in any way for any of the considered task.

5.3.2 Multi-core using bypass to tighten WCET estimates

Bypass was shown to possibly have positive impacts on tasks by reducing intra-task conflicts. However, this mechanism was basically introduced to reduce inter-tasks conflicts for analyses taking them into account. We focus in this section on the combination of the different bypass heuristics introduced previously with partitioning, (1) to reduce intra-task conflicts inside each partition and (2), with data cache analyses considering conflicts, to lower the amount of conflicts that should considered.

The experiment results for the three considered systems are presented in Table 5.8. For each task, partitioning is opposed to data cache analyses considering inter-task conflicts. Each time, the results of a bypass heuristic have been taken into consideration whether it is CB (column 4 and 7, respectively with partitioning of data cache analysis considering conflicts), AB (column 5 and 8, similarly) or IB (column 6 for a combination with partitioning and 9 if integrated in a data cache analysis considering conflicts). The included metrics are the WCET of each task (line 1), the number of references to the L1 cache (line 2) and the number of considered misses on this same cache level (line 3). Finally, the number of considered misses on the L2 shared cache is to be found on line 4. Remember that the results without bypass were introduced in Table 5.7.

Compared to the use of the same methods without bypass, the *Light* task set tends to have a similar behaviour: accounting for conflicts in the data cache analysis yields tighter WCET estimates because the L2 cache is big enough. However, using bypass with partitioning, the estimated WCETs are smaller than using partitioning alone. Bypass allows a reduction of the intra-task conflicts induced by the small cache partitions (see *minver* as an example in Table 5.7 and 5.8).

Similarly, for *Heavy*, the results are close to the one using the same methods without bypass. But, as previously illustrated (see Table 5.6 for *ludcmp*) *ludcmp*, with the help of IB, tends to benefit from

System	Benchmark	Metric	Partitioning	Full conflicts
Light	jfdctint	WCET contribution of data caches (cycles)	176953	170653
		nb of L1 accesses	1444	1444
		nb of L1 misses	307	307
		nb of L2 misses	307	265
	minver	WCET contribution of data caches (cycles)	94933	91933
		nb of L1 accesses	1067	1067
		nb of L1 misses	248	248
		nb of L2 misses	248	228
	qurt	WCET contribution of data caches (cycles)	148126	148126
		nb of L1 accesses	1948	1948
		nb of L1 misses	287	287
		nb of L2 misses	131	131
	statemate	WCET contribution of data caches (cycles)	370949	370949
		nb of L1 accesses	2346	2346
		nb of L1 misses	534	534
		nb of L2 misses	507	507
Heavy	ludcmp	WCET contribution of data caches (cycles)	543949	543949
		nb of L1 accesses	5765	5765
		nb of L1 misses	1721	1721
		nb of L2 misses	1721	1721
	matmult	WCET contribution of data caches (cycles)	7680840	7680840
		nb of L1 accesses	114191	114191
		nb of L1 misses	24006	24006
		nb of L2 misses	24006	24006
	ns	WCET contribution of data caches (cycles)	273209	273209
		nb of L1 accesses	5468	5468
		nb of L1 misses	628	628
		nb of L2 misses	628	628
Mixed	fft	WCET contribution of data caches (cycles)	2310240	4216150
		nb of L1 accesses	23932	23932
		nb of L1 misses	16676	16676
		nb of L2 misses	3970	16676
	matmult	WCET contribution of data caches (cycles)	7680840	7680840
		nb of L1 accesses	114191	114191
		nb of L1 misses	24006	24006
		nb of L2 misses	24006	24006
	ns	WCET contribution of data caches (cycles)	273209	273209
		nb of L1 accesses	5468	5468
		nb of L1 misses	628	628
		nb of L2 misses	628	628
	qurt	WCET contribution of data caches (cycles)	171526	171526
		nb of L1 accesses	1948	1948
		nb of L1 misses	287	287
		nb of L2 misses	287	287

Table 5.7: Effects of different methods for static multi-level n-way cache analysis (4-way L1 cache, 8-way L2 shared cache, cache sizes of 1KB (resp. 4KB) for L1 (resp. L2)) in multi-core environments.

the L2 cache. In combination with partitioning, it means that the memory segments can be given to *ludcmp* as the other tasks would not benefit from any segment. In the context of a data cache analysis considering conflicts, *ludcmp* can still benefit from the cache as BP reduces the amount of conflicts; not using bypass for the other tasks might have left too many conflicts precluding *ludcmp* from using the

System	Benchmark	Metric	Partitioning			Analysis considering conflicts		
			CB	**AB**	**IB**	**CB**	**AB**	**IB**
Light								
	jfdctint	WCET contribution of data caches (cycles)	176953	176953	170653	170653	170653	170653
		nb of L1 accesses	1444	1444	1444	1444	1444	1444
		nb of L1 misses	307	307	307	307	307	307
		nb of L2 misses	307	307	265	265	265	265
	minver	WCET contribution of data caches (cycles)	87733	87733	92083	84733	84733	91933
		nb of L1 accesses	1067	1067	1067	1067	1067	1067
		nb of L1 misses	248	248	248	248	248	248
		nb of L2 misses	200	200	229	180	180	228
	qurt	WCET contribution of data caches (cycles)	145826	145446	145286	145826	145446	145286
		nb of L1 accesses	1948	1948	1948	1948	1948	1948
		nb of L1 misses	183	130	129	183	130	129
		nb of L2 misses	129	130	129	129	130	129
	statemate	WCET contribution of data caches (cycles)	363799	359749	359749	359749	359749	359749
		nb of L1 accesses	2346	2346	2346	2346	2346	2346
		nb of L1 misses	464	464	464	464	464	464
		nb of L2 misses	464	437	437	437	437	437
Heavy								
	ludcmp	WCET contribution of data caches (cycles)	543949	543949	519049	543949	543949	519049
		nb of L1 accesses	5765	5765	5765	5765	5765	5765
		nb of L1 misses	1721	1721	1721	1721	1721	1721
		nb of L2 misses	1721	1721	1555	1721	1721	1555
	matmult	WCET contribution of data caches (cycles)	7680840	7680840	7680840	7680840	7680840	7680840
		nb of L1 accesses	114191	114191	114191	114191	114191	114191
		nb of L1 misses	24006	24006	24006	24006	24006	24006
		nb of L2 misses	24006	24006	24006	24006	24006	24006
	ns	WCET contribution of data caches (cycles)	273209	273209	273209	273209	273209	273209
		nb of L1 accesses	5468	5468	5468	5468	5468	5468
		nb of L1 misses	628	628	628	628	628	628
		nb of L2 misses	628	628	628	628	628	628
Mixed								
	fft	WCET contribution of data caches (cycles)	2293170	2293170	2227970	2293170	2227970	2227970
		nb of L1 accesses	23932	23932	23932	23932	23932	23932
		nb of L1 misses	15843	15843	16107	15843	16107	16107
		nb of L2 misses	3964	3964	3468	3964	3468	3468
	matmult	WCET contribution of data caches (cycles)	7680840	7680840	7680840	7680840	7680840	7680840
		nb of L1 accesses	114191	114191	114191	114191	114191	114191
		nb of L1 misses	24006	24006	24006	24006	24006	24006
		nb of L2 misses	24006	24006	24006	24006	24006	24006
	ns	WCET contribution of data caches (cycles)	273209	273209	273209	273209	273209	273209
		nb of L1 accesses	5468	5468	5468	5468	5468	5468
		nb of L1 misses	628	628	628	628	628	628
		nb of L2 misses	628	628	628	628	628	628
	qurt	WCET contribution of data caches (cycles)	153926	145446	145286	145826	145446	145286
		nb of L1 accesses	1948	1948	1948	1948	1948	1948
		nb of L1 misses	183	130	129	183	130	129
		nb of L2 misses	183	130	129	129	130	129

Table 5.8: Effects of different bypass methods on partitioning and data cache analysis considering conflicts for static multi-level n-way cache analysis (4-way L1 cache, 8-way L2 shared cache, cache sizes of 1KB (resp. 4KB) for L1 (resp. L2)) in multi-core environments.

L2 cache.

Finally, *Mixed* exhibits promising results for both methods in combination with bypass. Indeed, both the data cache analysis considering conflicts and the partitioning tend to have better results. Though this time, data cache analysis considering conflicts has improved benefits. Indeed, using bypass *matmult* reduces cache pollution and thus let *fft* benefit from the L2 shared data cache.

Concerning the different bypass heuristics, it is hard to draw a neat line between their respective benefits. We can only observe that AB tends to allow tighter WCET estimates that CB.

Bypass, which already demonstrated some interesting results on uni-core architectures, tends to benefit to methods focusing on multi-core architectures with shared data caches too. In this context, data cache analysis considering conflicts show the best estimated WCET tightening. Still, partitioning benefits too from improved segment allocations with bypass.

Conclusion

This document began with a brief overview of existing methods to perform timing analyses of tasks in uni-core and, to a lesser extent, multi-core architectures. The methods that were presented specifically target the analysis of instruction or data caches.

Based on these elements, extension to existing abstract interpretation based cache analyses have been proposed to handle access indeterminism induced by the use of data caches. Then, this analysis was used to define a multi-level data cache analysis which considers LRU set-associative non-inclusive caches. The considered caches of the hierarchy implement the *write-through*, *write-no-allocate* update policies.

In a second stage, two main tracks were followed to consider hierarchies of data caches in the context of multi-core architectures: per-task partitioning on the one hand which reduces the portion of cache affected to each task to preclude inter-task conflicts and, data cache analysis considering conflicts, on the other hand, adapted to data caches. To reduce the impact of these two methods on the worst-case performances of tasks, three different, per-instruction, bypass heuristics were proposed.

Results first shows that considering the whole cache hierarchy is interesting with a computed WCET being in average 6% smaller than the WCET considering only the first cache level. Even if not mentioned here, the computation time is fairly reasonable, results being computed for all the benchmarks in a couple of minutes.

Bypass was also shown to be promising as it tighten the computed tasks WCET, even on uni-core architectures. Indeed, it partially compensates the many concerns that should be addressed in data cache analyses: address analysis precision and accesses sequencing. Still no heuristic was proved to be better than another in all the cases. A clue to improve heuristics might be the use of an incremental process to choose bypassed instructions. The modifications induced by the bypassing of an instruction could be spread along its neighbours.

Between partitioning and conflict estimation, none was shown to outperform the other. The former can benefit to task sets mixing heavy cache polluting tasks and cache fitting ones. The latter seems to suit to well-proportioned systems (tasks fit in the cache). Both benefit from the use of bypassing.

Keeping the same study context, hybrid partitioning/conflict estimation methods could be investigated, tasks sharing different partitions along their execution with the objective of reducing inter-task conflicts within each partition. This would require data mapping alterations, which in itself might be another interesting track to follow.

Note that we address only a subset of the problems encountered by the use of, first, data caches and then multi-core architectures. Most of the relevant issues being put aside using assumptions on the hardware. This allows us to make a first step towards more generic solutions. Still, the architecture defined here might have good performance with regards to both average and worst case execution time, but the proof is left as a future work. Similarly the integration of write-backs, coherency protocols and accesses interleaving is an open issue.

Bibliography

[1] Marti Campoy, A. Perles Ivars, and J. V. Busquets Mataix. Static use of locking caches in multitask preemptive real-time systems. In *In Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*, 2001.

[2] Antoine Colin and Isabelle Puaut. A modular & retargetable framework for tree-based wcet analysis. In *ECRTS '01: Proceedings of the 13th Euromicro Conference on Real-Time Systems*, page 37. IEEE Computer Society, 2001.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[4] Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real-time systems. In *LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 16–30. Springer-Verlag, 1998.

[5] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-timesystems. *Real-Time Syst.*, 17(2-3):131–181, 1999.

[6] Jan Gustafsson. Usability aspects of wcet analysis. In *ISORC '08: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 346–352. IEEE Computer Society, 2008.

[7] Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. Research Report RR-6907, INRIA, 2009.

[8] Damien Hardy and Isabelle Puaut. Predictable code and data paging for real time systems. In *ECRTS '08: Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, pages 266–275. IEEE Computer Society, 2008.

[9] Damien Hardy and Isabelle Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 456–466. IEEE Computer Society, 2008.

[10] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, July 2003.

[11] D. B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *IEEE Real-Time Systems Symposium*, pages 229–239, 1989.

[12] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47(6):700–713, 1998.

[13] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. WCET analysis of multi-level set-associative data caches. In *9th Int' Workshop on Worst-Case Execution Time Analysis*. To appear, 2009.

[14] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 456–461. ACM, 1995.

[15] Thomas Lundqvist and Per Stenström. A method to improve the estimated worst-case performance of data caching. In *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, page 255. IEEE Computer Society, 1999.

[16] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 12. IEEE Computer Society, 1999.

[17] Frank Mueller. Compiler support for software-based cache partitioning. *SIGPLAN Not.*, 30(11):125–133, 1995.

[18] Frank Mueller. *Static cache simulation and its applications*. PhD thesis, 1995.

[19] Frank Mueller. Timing analysis for instruction caches. *Real-Time Syst.*, 18(2/3):217–247, 2000.

[20] Hemendra Singh Negi, Tulika Mitra, and Abhik 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. ACM, 2003.

[21] Isabelle Puaut. Wcet-centric software-controlled instruction caches for hard real-time systems. In *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 217–226. IEEE Computer Society, 2006.

[22] Isabelle Puaut and David Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium*, page 114. IEEE Computer Society, 2002.

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

[24] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Syst.*, 37(2):99–122, 2007.

[25] J. E. Sasinowski and J. K. Strosnider. A dynamic programming algorithm for cache memory partitioning for real-time systems. *IEEE Trans. Comput.*, 42(8):997–1001, 1993.

[26] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., 1986.

[27] Rathijit Sen and Y. N. Srikant. WCET estimation for executables in the presence of data caches. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 203–212. ACM, 2007.

[28] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[29] Jan Staschulat, Simon Schliecker, and Rolf Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 41–48. IEEE Computer Society, 2005.

[30] Vivy Suhendra and Tulika 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. ACM, 2008.

[31] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache andpath analyses. *Real-Time Syst.*, 18(2/3):157–179, 2000.

[32] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[33] Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for higher program predictability. In *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 272–282. ACM, 2003.

[34] Xavier Vera, Björn Lisper, and Jingling Xue. Data caches in multitasking hard real-time systems. In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*, page 154. IEEE Computer Society, 2003.

[35] Xavier Vera and Jingling Xue. Let's study whole-program cache behaviour analytically. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 175. IEEE Computer Society, 2002.

[36] Joachim Wegener and Frank Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Syst.*, 21(3):241–268, 2001.

[37] Randall T. White, Frank Mueller, Chris Healy, David Whalley, and Marion Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Syst.*, 17(2-3):209–233, 1999.

[38] 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. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.

[39] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. *PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis*, volume 3463/2005 of *Lecture Notes in Computer Science*, pages 281–292. Springer Berlin, March 2005.

[40] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44. ACM, 1991.

[41] Jun Yan and Wei 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*. IEEE Computer Society, 2008.