Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures.

Benjamin Lesage, Damien Hardy, Isabelle Puaut

Shared Data Cache Conflicts Reduction for WCET Computation in Multi-Core Architectures.

Benjamin Lesage Damien Hardy Isabelle Puaut
University of Rennes 1, UEB, IRISA
Rennes, France
{Benjamin.Lesage, Damien.Hardy, Isabelle.Puaut}@irisa.fr

Abstract

The use of multi-core architectures in real-time systems raises new issues regarding the estimation of safe and tight worst-case execution times. Indeed, the sharing of hardware resources occurring on such architectures is a new source of indeterminism. Caches, as one of these shared assets, become harder to analyse; concurrent tasks may any time alter their contents. This paper presents a safe method to estimate conflicts stemming from data cache sharing and their integration in data cache analyses. The other, and foremost, contribution of this paper is the introduction of bypass heuristics to reduce these conflicts, allowing for reuse to be more easily captured by shared caches analyses.

1 Introduction

Tasks in hard real-time systems, in addition to the correctness of computed values, have timing constraints; they are required to meet their deadlines in all situations, including the worst-case one. To attain this level of confidence, the *worst-case execution time* (WCET) of aforementioned tasks has to be estimated. These estimates must offer both tightness and safety properties. Tightness means that they must be as close as possible to the actual WCET of a task, not to overestimate the resources required by the system. Safety is the guarantee that the computed WCET is greater than or equal to any possible execution time.

Caches, whether they hold instructions, data or both, are a valuable mechanism when it comes to providing embedded real-time systems a sufficient throughput. Indeed, based on temporal and spatial localities of tasks, they help to fill the increasing gap between fast micro-processors and relatively slower main memories. However, caches come at the cost of a reduced predictability due to their dynamic behaviour which makes safe and precise WCET estimations on architectures with caches harder.

Much research has been undertaken during the last two decades with the objective of predicting WCET in architectures equipped with caches. These methods were first introduced for single-level instruction caches [15, 21] and, later, extended to support the analysis of hierarchies of non-inclusive instruction [9], data [11] or unified [3] caches. Other branches of study focus on the use of specific mechanisms, like locking [24] or partitioning [10], to increase cache predictability.

Multi-core architectures raise new issues in the context of WCET computation. Indeed, there might be hardware resources shared between multiple cores. Caches, as one of these resources, tend to be more difficult to analyse, their contents being possibly altered at any time by a concurrent task running on another core.

Contributions tackling with this behaviour are pretty scarce. On the one hand are contributions estimating and taking into account the additional indeterminism caused by concurrent tasks altering shared cache levels' contents [12, 7]. On the other hand are methods relying on the preclusion of sharing-related conflicts using partitioning [10], locking [24] or both [20].

This paper presents a WCET computation method for tasks in the context of multi-core architectures with non-inclusive data cache hierarchies. Similarly to [12, 7], we estimate shared data cache related conflicts and consider them during the analysis of the aforesaid shared data cache. Therefore, we deal with indeterminism stemming from both non-deterministic data accesses and data cache sharing. The other, yet foremost, contribution of this paper lies in the further introduction of bypass heuristics. Relying on the combination of a static decision taken through software means and specific hardware, bypass provides some control on analysed tasks' use of data caches. The proposed bypass heuristics are based either on reuse information computed by prior data cache analyses or on accessed data structures properties. These bypass heuristics aim at reducing inter-tasks conflicts and the pressure on shared cache levels, should it become too important.

The rest of this paper is organized as follows. Section 2 surveys related works. Assumptions on the tasks and hardware that can be analysed are presented in Section 3. Focusing on our proposals, Section 4, first, presents conflicts estimation and integration in the con-

text of shared data caches' analysis. Then Section 5 introduces a compiler-directed bypass mechanism and data-cache oriented heuristics to reduce these conflicts. Experimental results are given in Section 6. Finally, Section 7 summarizes this study and gives directions for future works.

2 Related works

Analysing an architecture with caches to produce a safe WCET estimate is a complex task due to the numerous predictability issues raised by their dynamic behaviour and replacement policies. With this objective in mind, many static cache analyses have been defined during the last lustra. Such analyses, to ensure the safety of subsequent timing analyses, must estimate cache contents at every point of the program while considering all execution paths altogether. These possible cache contents can be represented either using sets of *concrete cache states* [16] or using the more compact *abstract cache state* representation [21, 19, 6, 11].

One of the main approaches for the WCET analysis of architectures with set-associative caches [21] relies on *abstract interpretation* [5] and abstract cache states to perform three fixpoint analyses. These analyses aim at computing, respectively, if a memory block is *always* present in the cache, *may* be present in the cache or is *persistent* in the cache, i.e. once inserted in the cache it will not be evicted before its reuse. This information is then used to classify memory references' worst-case behaviour with respect to the cache.

Originally described for instruction caches, [21] was extended to support imprecise accesses in [19, 6]. These accesses, specific to data caches, arise as their target in memory may not be precisely computable off-line. [11] then further extended [21, 19, 6] for increased analysis precision and support for hierarchies of non-inclusive data caches.

Another solution to deal with this issue is the use of *Cache Miss Equations* [25, 23, 13]. To achieve data cache behaviour estimation, loops' iteration space is represented as a polyhedron. *Reuse vectors* [27] are set between iterations and used as a basis to set up and solve cache miss equations to accurately locate misses. However, according to the authors, it suffers from a lack of support for non properly nested loops or non affine array indexes.

Shared instruction cache analysis, in the context of multi-core architectures, has also already been explored in prior works such as [20, 28, 12, 7]. The former study, [20], compares locking and partitioning as means of precluding cache sharing-induced conflicts. [28] relies on the parallel analysis of two conflicting tasks to issue a classification of accesses' behaviour. Concerning [12, 7], they rely on conflicts estimation and integration in cache analyses. Then [12] uses synchronisation between tasks to reduce the sets of considered conflicts beyond each synchronisation point. On the other hand, [7] detects *Static Single Usage* cache blocks and set them as bypassing shared cache levels.

Bypass was also used in previous studies [22, 13] in the context of data caches. In [13], structures accessed by unpredictable accesses, whose precise target address is not known statically, are first identified. These structures are then marked as non-cacheable to improve the precision of data cache analyses. Instead, [22] decides for each access whether it should bypass a cache level or not. A dynamic and a static selection procedure are presented, each time relying on instructions' hit rate.

Like [12, 7], this paper follows the conflict estimation and integration in cache analyses approach, namely, the data cache analysis introduced in [11]. However, since we focus on hierarchies of data caches, we address additional issues such as data coherency and sharing. We also introduce static per-instruction bypass strategies which allows for more fine-grained decisions when compared to per-data structure based ones. Various bypass heuristics, based on accessed structures properties or statically available reuse information, are presented. The objective of these bypass heuristics is to help reducing inter-task conflicts stemming from data caches sharing.

3 Assumptions and notations

Data caches being the core of this study, code is assumed not to interfere with data in the analysed cache levels. Whether this separation is achieved through hardware or software means is irrelevant.

The studied cache hierarchies are made up of N data cache levels, the number 1 being the closest to the processor and conversely cache level number N being the farthest. Each cache of the hierarchy is expected to implement the LRU replacement policy; when eviction is required in a cache level, the least recently used cache block is selected. Any cache level in the hierarchy may be shared between two or more cores.

Analysed hierarchies are assumed to implement the non-inclusive policy. For load instructions, when in search for a piece of information, cache levels are investigated one after the other from the top of the hierarchy, closest to the processor, until the required information is found. The missing cache line will be inserted in all cache levels traversed during the search where it could not be found.

Concerning store instructions, the use of the *write-through* policy is assumed in combination with the *write no-allocate* policy; the modification issued by a store instruction goes all the way to the main memory, updating all cache levels where the information is to be found on its way. Compared to the *write-back* update policy, the *write-through* update policy suffers from greater store latencies, the price for a far higher predictability.

Tasks migration during execution and preemptions related delays are out of the scope of this study and are not of our concern for the time being. Task synchronisation might be used in the analysed system but we do

not attempt to benefit from this information. Likewise, we do not try to take advantage of any knowledge about shared data, yet they are safely considered during data cache analyses. Instead, we focus on the impact of cache sharing related delays.

A **memory reference** is defined as a reference to *data* triggered by a load or store instruction in a fixed call context. In addition, a memory reference does not issue more than one data-related memory operation. This assumption is reasonable with regard to existing architectures.

Latencies to access the different cache levels, shared as well as privates ones, are assumed to be bounded and known prior to the timing analysis, as an example using mechanisms such as *Time Division Multiple Access* [18] or the bus arbiter introduced in [17].

Table 1 summarizes additional definitions, notations and acronyms used throughout the present paper.

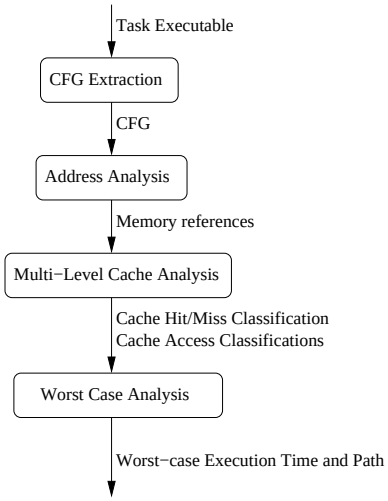
4 Data cache analysis

Robust foundations are required when it comes to estimating the contribution of both private and shared data caches in the timing analysis of a task. In the following, the steps to build an analysis, considering a task without interferences due to concurrent tasks on other cores, are first presented (§ 4.1). The estimation and integration of cache sharing-related conflicts are introduced by extending this basis (§ 4.2).

4.1 Multi-level uni-core data cache analysis [11]

This section outlines a multi-level data cache analysis, which various steps are outlined in Figure 1. For the time being, the analysed task has the monopoly over data caches in the hierarchy, it does not suffer from interferences caused by tasks running on the other cores. The consideration of data cache sharing is delayed to section 4.2.

Figure 1: Task analysis overview



The first step of the presented analysis is to extract, from the analysed executable, a *Control Flow Graph* (CFG). Using this graph, a data address analysis is performed. The objective is to estimate, for each memory reference, a safe set of the data addresses it might access. Based on the analysis used in [8], both for global and on-stack accesses, the precise address of the accessed memory block is computed for scalars accesses whereas the whole array address interval is returned in case of array accesses. Insights on the subject of address analysis, for the interested reader, can be found in [2].

Then, caches of the hierarchy are analysed one by one, from the cache closest to the processor to the furthest one. Upon each level and for each instruction issuing memory operations (loads and stores in the context of data caches), its worst-case behaviour with regard to the data cache is computed in terms of a *Cache Hit/Miss Classification* (CHMC). Three abstract interpretation-based cache contents analyses are performed to achieve this classification [21]. Given a program point, they respectively determine cache blocks that will always be present (ensuring an *Always-Hit* -AH- classification), persistent cache blocks in loops, allowing for accesses to be classified as *First-Miss* (FM) and cache blocks that may be present. References to blocks that will not be present can be classified as *Always-Miss* (AM). If none of these classifications can be ensured, the reference's classification is set to *Not-Classified (NC)*.

The safety of the multi-level data cache analysis relies on a so-called *Cache Access Classification* (CAC). The CAC depicts, given a memory reference, a safe estimate of whether it will *Always* or *Never* occurs on a given cache level. If no such guarantee can be statically given about a memory reference occurrence on a cache level, it is classified as *Uncertain*. Since the first cache level is always accessed, an *Always* CAC for all references to the L1 cache is assumed. Concerning subsequent cache levels' CAC, it depends on their direct predecessor's CAC and CHMC classifications. As it is not the subject of this contribution, we will not detail this computation. Interested readers should refer to [9] for additional details.

The final stage, WCET computation with respect to data caches uses both the CAC and CHMC for each cache level. Suffice to say that, given a memory reference, latencies for all possibly accessed cache level should be considered.

4.2 Multi-core shared data cache analysis

To take cache sharing into account, the uni-core multi-level data cache analysis (§4.1) needs to be extended. Indeed, rival tasks running on other cores might access a shared cache level, hence altering its contents. Not taking such alterations in consideration would result in unsafe classifications of memory references' behaviour with regard to this shared cache level and subsequent ones, as a consequence.

To tackle with this issue, we use a twofold process.

Category	Name	Description
Cache parameters	$CacheAssociativity_L$	Cache level L associativity degree.
	$CacheSet_L(b)$	Memory block b cache set on cache level L.
Memory reference	*memory reference*	Reference to *data* triggered by a load or store instruction in a fixed call context.
	$Instruction_r$	The instruction tied to memory reference r.
	$memory\_references_t$	Task t memory references.
	$memory\_blocks_{r,L}$	Memory blocks possibly accessed by reference r on cache level L.
Cache Classifications	$CHMC_{r,L}$	Memory reference r Cache Hit/Miss Classification on cache level L.
	AH	*Always-Hit* CHMC classification.
	FM	*First-Miss* CHMC classification.
	AM	*Always-Miss* CHMC classification.
	NC	*Not-Classified* CHMC classification.
	$CAC_{r,L}$	Memory reference r Cache Access Classification on cache level L.
Conflict estimation	$conflict\_blocks_{t,L}(s)$	Conflicting cache blocks mapping to cache set s for task t on shared cache level L.
	$task\_blocks_{t,L}(s)$	All cache blocks, task t may store in cache set s, cache level L.
	$CCN_{t,L}(s)$	Number of conflicting cache blocks mapping to cache set s of cache level L for task t.
Bypass	$Bypass(i, L)$	True if instruction i bypasses cache level L.
	RB	Reuse Bypass strategy, selects instructions which access memory blocks not detected as reused.
	$next\_reference_L(r, b)$	Closest memory references accessing memory block b after r, such that there is a path from r to this reference without any other possible reference to b.
	$hitCache_L(r)$	True if memory reference r might hit on cache level L.
	$successors(n, G)$	Node n successors in graph G.
	CFG_t	Task t Control Flow Graph.
	$CFG_t \triangleright b$	Reduction of CFG_t to memory references to block b.
	$mayTriggerHit(r, L)$	True if memory reference r brings into cache level L blocks that may later be reused before they are evicted.
	AIB	All Indeterministic Bypass strategy, selects indeterministic memory references.
	X-RCUB	Reduced Cache Usage Bypass strategy, brings a task's cache usage below the specified X cache ways bound.
Evaluation metrics	$DMCU_{L2}$	Data Cache Maximum Usage, the number of different memory blocks a task might store in cache level $L2$.
	$PHRL_{L2}$	Predicted Hit Ratio for $L2$ cache level.

Table 1: Notations and acronyms

First, conflicts stemming from data cache sharing with the tasks running on other cores are estimated. Then, this conflict estimation is used during the shared cache level analysis to update memory references' CHMC.

4.2.1 Conflict estimation

Inter-task conflicts in the context of multi-core architectures stems as *any* task, at *any* time, might alter shared cache contents. In the context of shared data cache analysis, the most intuitive approach to precisely model this behaviour would be to study the interleaving of all tasks' memory references. However, the computation of this set is in practice far too much space and time consuming [26].

Instead, we abstract both accesses' ordering and occurrence count from rival tasks. A task, concurrent to the analysed one, is supposed to access all of its memory blocks, any time, boundlessly, while the analysed task is executed. The set $conflict\_blocks_{t,L}(s)$ contains for task t and cache set s of cache level L those conflicting cache blocks, used by other tasks, that may enter in conflict for cache space with the analysed task's cache blocks:

$$conflict\_blocks_{t,L}(s) = \bigcup_{u \in (T - \{t\})} task\_blocks_{u,L}(s)$$

$$task\_blocks_{u,L}(s) = \{b \mid \exists r \in memory\_references_u \wedge b \in memory\_blocks_{r,L} \wedge CAC_{r,L} \neq Never \wedge CacheSet_L(b) = s\}$$

With $memory\_references_u$ the set of memory references in task u, $memory\_blocks_{r,L}$, the set of memory blocks possibly accessed by memory reference r on cache level L, $CAC_{r,L}$, the CAC of memory reference r with respect to cache level L and T the set of tasks in the considered system.

We do not need to keep the precise set of conflicting memory blocks in the analysed shared cache and only keep track of the number of blocks which might cause conflicts in the cache. This number is hereafter named *cache block conflict number* (CCN):

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

4.2.2 Conflict integration

Once the *cache block conflict number* (CCN) has been computed for task t and cache level L, a data cache analysis of level L is performed using the aforementioned multi-level data cache analysis with a slight modification. To faithfully take into account cache conflicts stemming from sharing, the manipulated abstract caches states are modified. For each cache set s of shared cache level L, at most $CCN_{t,L}(s)$ cache blocks may be allocated to the tasks running on other cores during task t analysis: only

$CacheAssociativity_L - CCN_{t,L}(s)$ are, for sure, available for each cache set s. This potentially reduced available cache space is reflected by an alteration of memory references' CHMC, compared to the case where no conflicts were considered.

Focusing on data sharing between tasks, shared cache blocks may have been brought into the cache by rival tasks and may be present during the analysed task execution. To model this phenomenon, memory references to such shared blocks cannot be classified as *Always-Miss* anymore for shared cache levels; the *Not-Classified* classification is assumed instead.

We also assume that accesses to shared data bypass all private cache levels. Knowing that no task has access to private copies of shared data and using the *write-through* update policy, all copies of a datum are updated upon stores; no coherency protocol needs to be enforced.

5 Reducing conflicts using bypass

The conservatism inherent to conflicts consideration, as introduced in the previous section, might lead to little cache space detected as available by shared data cache analyses; too many conflicts may have to be accounted for.

In order to reduce these inter-task conflicts, we introduce a *bypass* mechanism (§ 5.1). The decision for an instruction to bypass a shared cache level is taken through software and implemented, at run-time, through hardware. With the objective of reducing the pressure on shared cache levels, we define heuristics to statically select accesses to be bypassed at runtime (§ 5.2).

5.1 Bypass mechanism

Each load instruction can statically be set as bypassing any shared cache level. Because this is a *per-instruction* decision, different load instructions, with the same target in memory, may have different bypass behaviour. It also implies that a given load instruction will have, at run-time, the same bypass behaviour in all execution contexts.

For a load instruction to bypass a cache level means that, upon the execution of this instruction, if there is a miss on this cache level, the searched memory block will *not* be inserted in that level. Whereas if the instruction hits on a bypassed cache level, no cache block, in this cache level, will see its age altered in any way. Therefore, this instruction does not contribute to the eviction of memory blocks from bypassed cache levels, it does not generate conflicts on these levels.

Such a mechanism requires specific hardware to be implemented. Architectures implementing IA-64 locality hints [1], which share similarities with bypass, might provide insights towards such an implementation. The bypass decision itself is statically taken through software means, e.g. the compiler.

In the following, $Bypass(i, L)$ will define instruction i behaviour with regard to cache level L such that $Bypass(i, L)$ is $true$ if instruction i bypasses data cache level L and $false$ otherwise.

Accounting for bypass information

Once $Bypass(i, L)$ has been computed for a given shared cache level, a data cache analysis of the bypassed cache level L and the subsequent ones is required to take this information into account. Upon an instruction bypassing cache level L, bypass is modelled by leaving cache contents unchanged. Note that even if $Bypass(i, L) = true$, instruction i might still be classified as *Always-Hit*, *First-Miss*, etc., as cache level L is still investigated at run-time.

$Bypass(i, L)$ also has an impact on data cache sharing-induced conflicts. The computation of the set of conflicting cache block generated by a task u has to be altered, not to include the contribution of memory references bypassing cache level L:

$$task\_blocks_{u,L}(s) = \{b \mid \exists r \in memory\_references_u \wedge b \in memory\_blocks_{r,L} \wedge CAC_{r,L} \neq Never \wedge \neg Bypass(Instruction_r, L) \wedge CacheSet_L(b) = s\}$$

With $memory\_references_u$ the set of memory references in task u, $memory\_blocks_{r,L}$, the set of memory blocks possibly accessed by memory reference r on cache level L, $CAC_{r,L}$, the CAC of memory reference r with respect to cache level L and $Instruction_r$ is the instruction tied to memory reference r.

5.2 Bypass heuristics

For each load instruction and each shared cache level, we have to decide whether or not this instruction should bypass this cache level. Exploring the whole solution space would allow us to choose the most fitted solution, but this would be far too time consuming. Instead, we introduce heuristics based either on statically computable reuse information (§5.2.1) or accessed data structures properties (§5.2.2).

5.2.1 Bypass strategy based on static knowledge of reuse

The **Reuse bypass** (RB) strategy relies on principles similar to the ones introduced in [7] for instruction caches. In [7], instruction cache blocks bypass a cache level if they are classified as *Static Single Usage*, i.e. they are not detected as reused before their eviction from the cache. The *Static Single Usage* status of cache blocks is determined using information on the CHMC and CAC obtained by prior cache analyses.

In the context of [7], instruction caches, a single instruction will always access the same memory block and instructions accessing the same memory block are not scattered all along the task's CFG. On the other hand, in our context, data caches, a same memory block can be the target of memory references issued by different load instructions and conversely, a memory reference may target many memory blocks.

Given an instruction i, the RB strategy tries to determine if it may cause a hit in cache L. For instruction i to cause a hit means that it brings into the cache blocks that are statically detected as reused, i.e. subsequent memory references to these cache blocks are classified as hits (AH or FM) by a prior data cache analysis. If instruction i does not meet these conditions, its impact on cache level L is expendable:

$$Bypass_{RB}(i, L) = \forall r \in memory\_references_i, \neg mayTriggerHit(r, L)$$

With $memory\_references_i$ the set of memory references tied to instruction i. $mayTriggerHit(r, L)$ is true if memory reference r may load, in cache level L, information that is statically detected as reused:

$$mayTriggerHit(r, L) = \forall b \in memory\_blocks_{r,L}, \exists nr \in next\_references_L(r, b) \wedge hitsCache_L(nr)$$

$$hitsCache_L(nr) = (CHMC_{nr,L} = AH \vee CHMC_{nr,L} = FM)$$

Where $CHMC_{nr,L}$ is the worst-case behaviour of memory reference nr with regards to data cache level L as computed by a prior L cache level analysis (§4).

Given a memory reference r, $next\_references_L(r, b)$ is the set of closest memory references following r and accessing memory block b: a memory reference nr belongs to $next\_reference_L(r, b)$ if both r and nr access the memory block b such that there is a path from r to nr without any other possible reference to b. $next\_reference_L(r, b)$ contains memory references that may profit from memory block b loaded in the cache by reference r:

$$next\_reference_L(r, b) = \{r' | b \in memory\_blocks_{r,L} \wedge r' \in successors(r, CFG_t \triangleright b)\}$$

Where $memory\_blocks_{r,L}$ is the set of memory blocks possibly accessed by memory reference r on cache level L and $successors(n, G)$ is the set of successors of node n in graph G. $CFG_t \triangleright b$ defines the CFG of task t reduced to memory references to memory block b; $CFG_t \triangleright b$ is constructed by removing from the CFG of task t basic blocks, then instructions, that are not memory references to memory block b while keeping edges through transitivity.

5.2.2 Bypass strategies based on accessed data structures properties

Indeterministic references, which precise target in memory cannot be statically computed, have two drawbacks. First, upon such a reference, we statically do not know its precise target in memory. As a consequence, during data cache analyses, we only know a set of memory blocks that *may* be inserted in the cache. Secondly, having the assurance that an indeterministic reference hits in the cache requires the whole set of memory blocks it may access to be present in the cache, which may not be possible if this set is too big.

Such indeterministic references are the prime target of the **All Indeterministic Bypass** (AIB) strategy. All these references are bypassed by the AIB strategy to prevent accesses' indeterminism and its impact on data cache analyses. Note that this is the most aggressive of all proposed solutions:

$$Bypass_{AIB}(i, L) = \exists r \in memory\_references_i \wedge |memory\_blocks_{r,L}| > 1$$

Where $memory\_references_i$ is the set of memory references tied to instruction i and $memory\_blocks_{r,L}$ the set of memory blocks possibly accessed by memory reference r on cache level L.

The **Reduced Cache Usage Bypass** (X-RCUB) strategy is a less aggressive parametrised strategy. Accesses of a task are bypassed until it does no longer use more than a fixed number, X, of cache ways. X-RCUB allows for control on the maximum shared cache space occupied by a task and therefore, allows for control on the conflicts it generates on shared cache levels.

We choose to bypass in priority accesses of a task with the largest predicted accessed memory ranges. Because of their indeterminism and heavier impact on the cache, they are the less likely to trigger reuse captured by data cache analyses:

while $\exists s \in CacheSet_L, |task\_blocks_{t,L}(s)| > X$ **do**
  pick reference r with the largest $memory\_blocks_{r,L}$
  $Bypass_{X-WSIB}(Instruction_r, L) \leftarrow true$
end while

Where $task\_blocks_{t,L}(s)$ is the whole set of memory blocks, mapping to cache set s of cache level L, that might be brought in cache by task t (§5.1) and $Instruction_r$ is the instruction tied to memory reference r.

6 Experimental results

6.1 Experimental setup

Cache analysis and WCET estimation: Experiments presented hereafter have been conducted on MIPS R2000/R3000 binary code compiled with gcc 4.1 without any code optimization and using the default linker memory layout. WCET estimates were computed using the Heptane timing analyser [4], more precisely its *Implicit Path Enumeration Technique* (IPET). Memory references were classified using the techniques presented in previous sections (§4). Based on this classification, data caches WCET contribution is evaluated using the methods presented in [11]. Focusing on caches, WCET estimates only take their contribution to the WCET. Contributions of other architectural features are not integrated in the presented results. This embodies pipelines and the timing anomalies emerging from their interaction with caches [14]. Assuming the cache classification *Not-Classified* has the same worst-case behaviour than the *Always-Miss* one is thus safe. The cache analysis starts with an empty cache.

Benchmarks: A subset of the benchmarks maintained by Mälardalen WCET research group[1] is used to conduct experiments. The characteristics of each benchmark are exposed in Table 2 (from left to right are the sizes in bytes of *code*, *data*, *bss* -uninitialized data- and maximum *stack* sections).

Cache hierarchy: Results presented in this section have been computed by assuming a two-level cache hierarchy. A 4-way private data cache with a 32B block size and a 1KB total size composes the first level of the hierarchy (L1). The second level of the hierarchy is made of an 8-way 4KB L2 shared data cache with the same 32B block size. Small cache sizes are selected to allow for sharing-related conflicts to arise. A perfect instruction cache, with a one cycle access latency, is assumed. Both the L1 and the L2 data caches implement the LRU replacement policy.

6.2 Experimental results

The objective of this section is to study the impact on tasks of conflicts related to data cache sharing and the contributions of bypass to reducing tasks' pressure on shared cache levels, and its resulting ability to lessen the number of conflicts.

First, we study the effect of bypass without taking any cache sharing into account in section 6.2.1. Thereby, we exhibit the impact of bypass on intra-task conflicts. The ability of bypass to reduce inter-task conflicts and the effects of said conflicts on tasks' predicted performances are studied in section 6.2.2.

6.2.1 Bypass in the context of uni-core architectures

In this paragraph, each task is assumed to be the sole task using both the L1 and the L2 cache levels. The objective is to study the impact of our bypass heuristics on intra-task conflicts and upon the pressure laid on the shared L2 data cache by each task. Four different configurations are studied, each time our bypass heuristics are applied on the L2 cache only: without bypass (No BP), using the reuse bypass (RB), all indeterministic bypass (AIB) and the 2-Reduced Cache Usage bypass (2-RCUB) strategies.

Note that for the sake of brevity we only study one configuration for the RCUB strategy: 2-RCUB. This configuration corresponds to an even distribution of the L2 data cache ways between members of four-task sets like the ones considered in later experiments.

The predicted hit ratio, PHR_{L2}, for the L2 cache level along the worst-case path computed by our analyser, is illustrated in Figure 2 for each task and each studied bypass configuration, from left to right: No BP, RB, AIB and 2-RCUB.

The analysed tasks show a heterogeneous predicted use of the L2 cache. Despite their small memory footprint, *fft*, *qurt* and *statemate* greatly benefit from the second level cache. *ns* and *matmult* exhibit the opposite behaviour, no

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

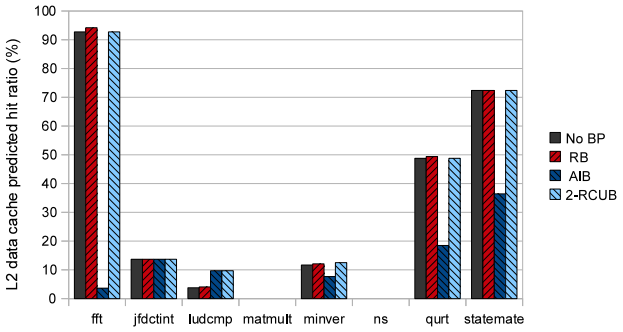


Figure 2: L2 cache predicted hit ratios for each benchmark in a uni-core architecture, within each bypass configuration.

reuse is captured on the L2 cache for their heavy footprint. *jfdctint*, *ludcmp* and *minver* show in-between captured reuse of the L2 cache.

With the exception of the AIB strategy, bypassing the cache has little to no effect on tasks' PHR_{L2}; if bypass decreases L2 inter-task conflicts, this is not beneficial for a task alone. RB and 2-RCUB do not worsen PHR_{L2} and RB may even give marginal benefits (*fft*, *ludcmp*, *minver* and *qurt*). However, AIB tends to reduce PHR_{L2} in exchange for a reduced pressure on the L2 cache level as will later be illustrated.

ludcmp is an interesting exception as it benefits from both the use of 2-RCUB and AIB strategies. Indeed, these heuristics bypass accesses to one of its data structure which would otherwise occult other accesses.

The expected strength of bypass lies in its capacity for reducing the pressure laid by a task on the shared data cache: the more accesses bypass the L2 data cache level, the less blocks are stored in this cache level, as a consequence, the less conflicting cache blocks are generated by a task. To estimate this impact, we introduce tasks' maximum L2 cache usage, $DMCU_{L2} = |\bigcup_{s \in CacheSet_{L2}} task\_blocks_{t,L2}(s)|$, in Figure 3. $DMCU_{L2}$ is the number of data memory blocks a task might store in cache level L2.

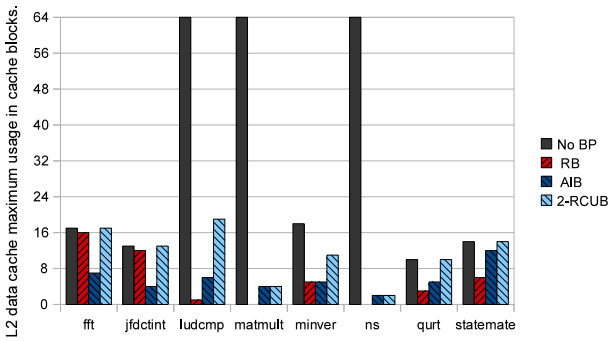


Figure 3: Tasks L2 data cache maximum usage for each task, using each bypass strategy and bounded to 64, in terms of cache blocks count.

Name	Description	Code size (bytes)	Data size (bytes)	Bss size (bytes)	Maximum Stack size (bytes)
fft	Fast Fourier Transform	3536	112	128	288
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	1200	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
statemate	Automatically generated code by STARC (STAtechart Real-time-Code generator)	8900	32	290	88

Table 2: Benchmark characteristics

Our proposed heuristics managed to reduce tasks' pressure on the L2 data cache. In general, AIB, which has a negative impact on tasks' PHR_{L2}, exhibit a high conflict reduction potential. It is followed by the less aggressive RB and finally the 2-RCUB strategies.

2-RCUB does its job and brings tasks' $DMCU_{L2}$ below the specified 2-ways bound. In our context, only *ludcmp*, *matmult*, *minver* and *ns* are concerned. Furthermore, as expected, no reuse was detected for the largest accesses of these tasks.

Speaking of RB, except for *fft* and *jfdctint*, it has an interesting strong impact on the cache usage of all benchmarks, without any negative impact on their PHR_{L2}.

As for *fft* and *jfdctint*, their high L2 data cache reuse rate leaves little room for RB to work with. The case of *jfdctint* is not as straightforward as the case of *fft* which has a high PHR_{L2}. Many of the structures accessed by *jfdctint* are classified as persistent by the L2 data cache analysis, and cannot be eliminated by the RB strategy. However, accesses to these structures are located in loops with small iteration counts and the first occurrences of these accesses are accounted for as misses, to load the structure in the L2 cache, hence a small PHR_{L2}.

The case of *matmult* and *ns* is also interesting. Neither deterministic accesses nor indeterministic ones are detected as reused by L2 data cache analyses. RB then outperforms AIB because it eliminates both families of accesses while AIB focuses on indeterministic ones only.

6.2.2 Impact of bypass in the context of multi-core architectures

From now on, we assume that the L2 data cache is shared between multiple tasks. Compared to the previous section, cache analyses now take into account the inter-task conflicts that may arise from data cache sharing. We estimate both the impact of inter-task conflicts and of bypass on tasks' L2 cache usage using the predicted hit ratio on the L2 cache, PHR_{L2}.

To evaluate the behaviour of a task regarding a shared cache level, we need to know the concurrent tasks conflicting for the said cache. Three different task sets have been built to exhibit some form of contrast. Each task in a set runs on its own core and is in conflict for the L2 cache with the other tasks of the set:

- **light** task set (*jfdctint*, *minver*, *qurt*, *statemate*) is composed of tasks with the smallest data memory footprint. They do not need a lot of cache space; their cumulated $DMCU_{L2}$, estimated without bypass, is of 55 cache blocks while the L2 cache level can hold 128 cache blocks.
- **heavy** task set (*matmult*, *ns*, 2 instances of *ludcmp*) uses tasks with a more important use of the cache (1595 cache blocks in the cumulated tasks' $DMCU_{L2}$). If *ludcmp* benefits from the L2 cache level, *matmult* and *ns* do not.
- **mixed** task set (*matmult*, *ns*, *qurt*, *fft*) comprises tasks the *heavy* set and the *light* one with the addition of *fft*. The cumulation of its composing tasks' $DMCU_{L2}$ reaches 262 cache blocks out of which 27 belongs to *fft* and *qurt* together.

PHR_{L2} is given for tasks of the *light*, *heavy* and *mixed* sets in Figures 4, 5 and 6 respectively. Again, our bypass heuristics are used only on the shared L2 data cache level and the same heuristic is used for all tasks at the same time. The four scenarios are: without bypass (No BP), reuse bypass (RB), all indeterministic bypass (AIB) and 2-reduced cache usage bypass (2-RCUB).

As expected for the *light* task set, accounting for conflicts seems to be a reasonable solution. Except for *qurt*, tasks of the *light* set are unaffected by conflicts stemming from L2 cache sharing with the other tasks. Without bypass or using the 2-RCUB strategy, *qurt* does not have enough room in the L2 cache level. With the help of RB, *qurt* manages to use some cache space. In summary, conflict estimation is reasonable and our heuristics behave well when there is little pressure on shared cache levels.

For the *heavy* task set on the other hand (Figure 5), considering cache sharing-related conflicts without bypass, no reuse is detected in the shared L2 cache level. Using bypass, only the instances of *ludcmp* benefit from this cache level. Either way, neither *ns* nor *matmult* exhibit captured reuse: bypass cannot improve their behaviour which are not due to inter-task conflicts but intra-task ones. Nonetheless, bypass reduces these tasks' impact on the shared cache level, so much so that *ludcmp* can benefit from the room freed by bypass on this cache level. Note that the better performances of AIB and 2-RCUB, for *ludcmp*, are related to their impact on intra-task conflicts and

not inter-task ones for this task (§ 6.2.1). On such cache-heavy tasks, using a majority of large data structures, all three heuristics perform well.

Concerning the *mixed* task set, again, without bypass, cache sharing-related conflicts are too important for reuse to be detected in the shared L2 cache level. With or without bypass, considering *matmult* and *ns*, no reuse of the shared cache level is captured. Still, their impact on this cache level is alleviated by our bypass heuristics; with bypass, things get better for *fft* and *qurt*. The negative impact of AIB on *fft*'s PHR_{L2} is the only responsible of its low PHR_{L2} in this context. Concerning *qurt*, the differences between RB and 2-RCUB originate from the better performances of RB at reducing $DMCU_{L2}$, and thus conflicts, for all tasks of the set.

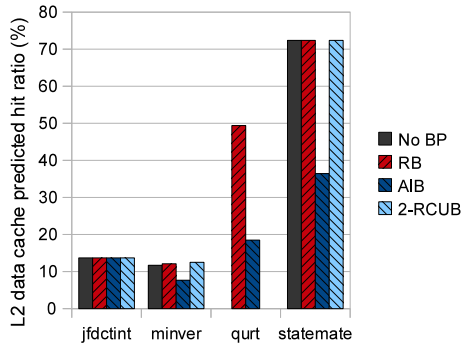


Figure 4: *Light* task set tasks predicted L2 hit ratio for each bypass strategy while competing for the shared L2 data cache.

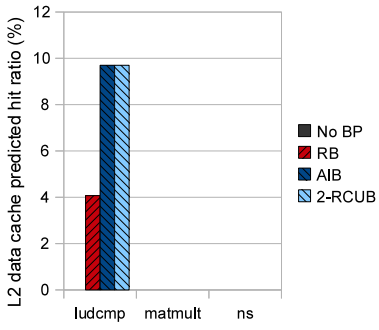


Figure 5: *Heavy* task set tasks predicted L2 hit ratio for each bypass strategy while competing for the shared L2 data cache.

Finally, considering computation time, the full analysis of a system (all tasks, both cache levels), in all bypass scenarios previously presented, always took less than 3 minutes on an Intel Core 2 Duo (2.53Ghz) with 4 GB of RAM.

These experimentations raise many issues regarding conflicts estimation and integration in the analysis of shared data caches on multi-core architectures. Considering altogether tasks with massive cache maximum requirements might result in low or null predicted hit ratios in shared cache levels. In addition, increasing the precision of conflicts estimation might be costly.

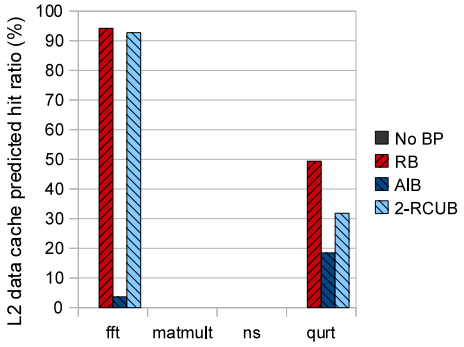


Figure 6: *Mixed* task set tasks predicted L2 hit ratio for each bypass strategy while competing for the shared L2 data cache.

A mechanism to reduce this amount of conflicts has been explored in this paper with the objective of bringing back the system shared cache usage under a reasonable limit, but without under exploiting shared cache levels (as this might results in poor average case performance). The introduced bypass strategies, aware of tasks behaviour, are a solution to this problem to a certain extent, i.e. when the pressure on shared cache level due to useful accesses is not overwhelming. Using RB, once not-reused memory references have been removed, a lot of pressure may still lie on shared caches. More aggressive heuristics, like AIB or the controlled 2-RCUB, are an interesting trade-off between individual task performances and global pressure on shared cache levels.

7 Conclusion

In this paper, we have presented a multi-level shared data cache analysis. This approach first estimates inter-tasks conflicts stemming from data cache sharing, conflicts which are accounted for in the shared data cache analysis. We also presented data-cache aware heuristics, based on the bypass mechanism, which aim at reducing these conflicts. Results show that plainly considering data cache sharing related conflicts, without any mechanism to reduce their number, is not a scalable approach as it tends to result in little to no reuse captured in shared cache levels. The bypass heuristics proposed in this document were shown to be an interesting solution to this issue.

Focusing on data cache sharing, it might be interesting to compare conflicts estimating methods to conflicts precluding ones such as locking or partitioning. Or even combine both methods to reduce hot spots in the shared cache or reduce the number of tasks that have to be considered as conflicting with the analysed one. Including tasks preemptions or migrations between cores during execution might be other interesting extensions.

References

[1] IA-64 application developer's architecture guide. 1999.

[2] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *IN CC*, pages 5–23. Springer-Verlag, 2004.

[3] S. Chattopadhyay and A. Roychoudhury. Unified cache modeling for WCET analysis and layout optimizations. In *IEEE Real-time Systems Symposium (RTSS)*., 2009.

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

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[6] C. Ferdinand and R. 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.

[7] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proceedings of the 30th Real-Time Systems Symposium*, pages 68–77, Washington D.C., USA, Dec. 2009.

[8] D. Hardy and I. 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] D. Hardy and I. 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] D. Kirk. Smart (strategic memory allocation for real-time) cache design. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 229–237, Dec 1989.

[11] B. Lesage, D. Hardy, and I. Puaut. WCET analysis of multi-level set-associative data caches. In N. Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[12] Y. Li, V. Suhendra, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *IEEE Real-time System Symposium (RTSS)*, 2009.

[13] T. Lundqvist and P. 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.

[14] T. Lundqvist and P. 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.

[15] F. Mueller. Static cache simulation and its applications. PhD thesis, 1994.

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

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

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

[19] R. 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.

[20] V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 300–303, New York, NY, USA, 2008. ACM.

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

[22] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 93–103, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[23] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *Real-Time Systems Symposium*, Cancun, Mexico, 2003.

[24] X. Vera, B. Lisper, and J. Xue. Data cache locking for tight timing calculations. *Trans. on Embedded Computing Sys.*, 7(1):1–38, 2007.

[25] X. Vera and J. 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.

[26] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(7):966–978, 2009.

[27] M. E. Wolf and M. 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.

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