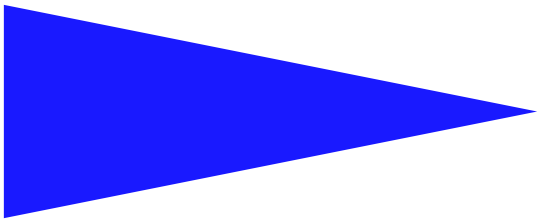
Scratchpad memories vs locked caches in hard real-time systems: a qualitative and quantitative comparison

Isabelle Puaut, Christophe Pais

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1818



SCRATCHPAD MEMORIES VS LOCKED CACHES IN HARD REAL-TIME SYSTEMS: A QUALITATIVE AND QUANTITATIVE COMPARISON

ISABELLE PUAUT AND CHRISTOPHE PAIS

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Scratchpad memories vs locked caches in hard real-time systems: a qualitative and quantitative comparison

Isabelle Puaut * and Christophe Pais **

Systèmes communicants
Projet CAPS

Publication interne n ° 1818 — Octobre 2006 — 22 pages

Abstract: Hard real-time tasks must meet their deadline in all situations, including in the worst-case one, otherwise the safety of the controlled system is jeopardized. In addition to this stringent demand for predictability, an increasing number of hard real-time applications need to be fast as well. As a consequence, architectures with caches and/or on-chip static RAM (scratchpad memories) are of interest for such applications. As compared to unlocked caches which may raise predictability issues for some cache replacement policies [HLTW03], locked caches and software-controlled on-chip static RAM are more easily amenable to timing analysis.

We propose in this paper an algorithm for off-line selection of the contents of on-chip memories. The algorithm supports two types of on-chip memories, namely locked caches and scratchpad memories. The contents of on-chip memory, although selected off-line, is changed at run-time, for the sake of scalability with respect to task size. The algorithm allows to make a quantitative comparison of worst-case performance of applications using these two kinds of on-chip memories. Experimental results show that the algorithm yields to good ratios of on-chip memory accesses on the worst-case execution path, with a tolerable reload overhead, for both types of on-chip memories. Furthermore, we highlight the circumstances under which one type of on-chip memory is more appropriate than the other depending of architectural parameters (cache block size) and application characteristics (basic block size).

Key-words: wcet, hard real-time system, compilation, software, hardware, memory, cache

(Résumé : tsvp)

Cette étude a été partiellement financée par les projets ANR Mascotte (ANR-05-PDIT-018-01) et Galogic

* isabelle.puaut@irisa.fr
** christophe.pais@irisa.fr

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Mémoires scratchpad vs caches gelés dans les systèmes temps-réel: une comparaison qualitative et quantitative

Résumé :

Un système temps-réel dur doit respecter toutes ses échéances dans toutes les situations, même dans celle de pire cas, si l'on ne veut pas laisser au hasard la sécurité du système. En plus de ce contraignant besoin de prédictabilité, les applications d'un tel système doivent s'exécuter rapidement. Pour cette raison, ces applications peuvent nécessiter des architectures contenant des caches et/ou des mémoires scratchpads. Comparé au fonctionnement dynamique habituel des caches conduisant à de nombreux problèmes en terme de prédictabilité, les caches gelés et mémoires scratchpad contrôlés par logiciel permettent une bien meilleure analyse temporelle.

Nous proposons dans cet article un algorithme permettant la sélection hors-ligne des contenus des mémoires sur-puces. Il supporte à la fois les caches et les mémoire scratchpad. Bien que choisi hors ligne, le contenu des mémoires on-chip est changé pendant l'exécution, pour permettre de prendre en compte les tailles importantes de certaines tâches. L'algorithme permet une comparaison quantitative des performances de pire-temps temps d'exécution en utilisant ces deux types de mémoire. Les résultats expérimentaux montrent qu'il conduit dans les deux cas à de bons ratios d'accès à la mémoire sur-puce sur le chemin du pire-temps d'exécution et que le surcôut lié au chargement dynamique dans la mémoire sur-puce est acceptable. De plus, nous soulignons les circonstances dans lequelles une mémoire sur-puce est plus ou moins approprié selon différents paramètres matériels (taille des bloc de cache) et différentes caractéristiques de l'application (taille des blocs de base).

Mots clés : pire-temps d'execution, système temps-réel, compilation, logiciel, matériel, mémoire, cache

1 Introduction

In hard real-time systems all task deadlines have to be met in all situations for safety reasons. For that reason, many schedulability analysis methods rely on the knowledge of an upper bound for the execution times of tasks (WCETs, for Worst-Case Execution Times). WCET estimates have to be *safe* (i.e. greater than any possible execution time) and as *tight as possible* (as close as possible as the execution time of the longest path). Safe bounds for task execution times may be computed using *static WCET analysis methods* that obtain WCETs through a static analysis of task source and/or object code [PB00].

WCET of programs is obviously influenced by the hardware in use. The increasing performance gap between the processor and the off-chip memory has made it important to use some kind of on-chip memory in real-time embedded systems. Caches have been extensively used to bridge that gap. The advantage of caches is that the allocation and deallocation of memory blocks from the cache are managed by hardware, in a transparent manner to the programmer and compiler. Unfortunately, caches are source of predictability problems in hard real-time systems [HLTW03]. A lot of progress has been achieved in the last ten years to statically predict worst-case execution times (WCETs) of tasks on architectures with caches [Mue00, HLTW03, LMW96, LS99a]. However, cache-aware WCET analysis techniques are not always applicable due to the lack of documentation of hardware manuals concerning the cache replacement policies. Moreover, they tend to be pessimistic with some cache replacement policies (e.g. pseudo round-robin, pseudo-LRU, random replacement policies) [HLTW03, Ber06]. Lastly, caches are sources of *timing anomalies* in dynamically scheduled processors [LS99b] (a cache miss may in some cases result in a shorter execution time than a hit). In such situations, *caches locking techniques* are of interest.

Locking techniques exploit hardware support allowing the software (compiler or programmer) to control the cache contents: *load* information into the cache and disable the cache replacement policy (*lock* or *freeze* the cache). This ability to lock cache contents is available in several commercial processors (ColdFire MCF5249, PowerPC 440, MPC5554, ARM 940 and ARM 946E-S). The contents of the locked cache can be fixed for the whole execution of a task (static locking) or changed at run-time (dynamic locking). Dynamic cache locking techniques have been shown in [Pua06] to provide tight worst-case WCET estimates as far as applications exhibit temporal locality.

An alternative to caches for on-chip storage is scratchpad memory. Scratchpad memories are small on-chip static RAMs that are mapped onto the address space of the processor at a predefined address range. Their inherent predictability have made them popular in real-time systems. Contrary to caches, the task of allocating code/data memory to the scratchpad memory is under software control (it lies with the compiler or programmer). Significant effort has been invested in developing efficient allocation techniques for scratchpad memories [KRI+01, UB03, LGX05, VWM04]. Except [SMRC05], all these techniques aim to reduce the average execution time (ACET) of programs, through memory access profiles. Such ACET-oriented techniques are not necessarily suited to real-time systems, since the execution path followed is average may not be the worst-case execution path. Only [SMRC05] aims at optimizing tasks worst-case performance. However, in that study, scratchpad allo-

cation is static (scratchpad contents is not changed at run-time), raising performance issue when the amount of code/data is much larger than scratchpad size. To the best of our knowledge, no WCET-oriented dynamic scratchpad allocation method has been proposed since now.

The contributions of this paper are twofold:

- We propose an algorithm for allocating code portions in on-chip memory, supporting two very similar types of memories: scratchpad memories and locked caches. The algorithm operates off-line for the sake of predictability of memory accesses. It introduces multiple load points in the code of a single task and selects the values to be loaded at run-time into the on-chip memory. The algorithm is WCET oriented in the sense that it aims at minimizing the task WCET estimate. The algorithm is a generalization of the algorithm we have previously proposed in [Pua06] for off-line selection of the contents of locked instruction caches.
- We give a qualitative and quantitative comparison of the use of dynamic WCET-oriented cache locking and scratchpad allocation. Experimental results show that the worst-case performance of applications using the two types of memory are very close to each other in most cases. The sources of differences between the two approaches are highlighted. In particular it is shown how architectural parameters (cache block size) and task structure (size of basic blocks) impact the task worst-case performance.

This paper focusses on dynamic loading of *code* into scratchpad memories and locked caches only. Extending the work to data is discussed in section 6.

The remainder of the paper is organized as follows. Section 2 presents the algorithm for the off-line selection of the contents of on-chip memory, and highlights the differences resulting from the type of on-chip memory into consideration. Section 3 discusses the reasons to use scratchpad memories instead of locked caches (or vice versa) in hard real-time systems. A quantitative comparison, in terms of WCETs and ratios of on-chip memory accesses along the worst-case execution path, is given in Section 4. We compare our work with related work in Section 5. Finally, we conclude in Section 6.

2 Selection of on-chip memory contents

This section presents an algorithm for off-line selection of the contents of two very similar classes of on-chip memories: locked caches and on-chip static RAM (scratchpad). The algorithm is applied off-line. It considers an isolated task, represented by its control flow graph (CFG). For every task, the algorithm selects (i) *reload points*, which are points where the on-chip memory will be loaded at run-time; (ii) *memory contents*, which are the pieces of code (basic blocks) to be loaded at run-time when control reaches the reload point. As a result, the code of the applications is divided into *regions* at the entry of which the contents of the on-chip memory is loaded[1]. Locations for reload points are loop pre-headers. On-chip memory contents is selected thanks to the knowledge of execution frequencies of basic blocks along the worst-case execution path (WCEP), obtained though an external WCET estimation tool. Furthermore, since the worst-case execution path may vary when a piece of code is elected to be loaded into on-chip memory, the WCEP is re-evaluated regularly in the course of the content selection procedure. The regions altogether cover all the code of the task. As a consequence, it can be known statically if a given instruction in the code will be an on-chip or an off-chip memory access.

2.1 Notations and assumptions

We consider a CPU a k-way set-associative instruction cache or a scratchpad memory:

- The cache is of size S_C and comprises a total of B blocks of S_B bytes each ($S_C = B * S_B$). Blocks are grouped into S *sets* of k cache blocks each; an instruction at address ad is mapped onto one of the k blocks of set $\lfloor \frac{ad}{S_B} \rfloor mod\ S$. We consider that there exists a mechanism to *load and lock* cache blocks into the instruction cache, inhibiting cache replacement on those blocks until they are unlocked. In the following, the term *program line* will denote a piece of code of cache-block size.
- The scratchpad is of size S_S; there is no a priori allocation unit in scratchpad. Allocation in scratchpad is only restricted by the some alignement constraints (for instance, alignment of instructions on 4 bytes boundaries).

In the following, t_{on} and t_{off} will denote respectively latencies to access on-chip memory (cache/scratchpad) and off-chip memory. We assume that the time for loading the piece of code sz into on-chip memory is a linear function of sz ($t_{reload} = t_i + t_l * sz$). Hardware parameters t_{on}, t_{off}, t_i, t_l are take from the hardware manuals (processor, system).

2.2 Content selection algorithm

The algorithm is made of two independent parts: selection of reload points (§ 2.2.1) and selection of on-chip memory contents (§ 2.2.2). The algorithm is illustrated on a small example in § 2.2.4.

[1] In multi-task applications, reloads of the on-chip memory occur at context switch times as well.

2.2.1 Selection of reload points

Reload points are placed at loop pre-headers (basic block before a loop header in the CFG) to exploit temporal locality. A cost function $CF(L)$, given in Equation (1), decides whether or not on-chip memory (cache/scratchpad) should be reloaded at the pre-headers of a loop L. $CF(L)$ is an estimation of the decrease of WCET estimate which would occur if loading the most frequently executed instructions of the loop. In the formulas: $f(s)$ denotes the total number of executions of a statement s along the WCEP; $mfi(L)$ denotes the most frequently executed instructions of loop L along the WCEP fitting in on-chip memory, $instr(L)$ denotes the whole set of instructions of loop L, and $pre-head(L)$ denotes the pre-headers of loop L.

$$WCET\_offchip(L) = \sum_{i \in pl(L)} f(i) * t_{off}$$

$$\begin{aligned} WCET\_onchip(L) = & \sum_{i \in mfi(L)} f(i) * t_{on} \\ & + \sum_{i \in instr(L) - mfi(L)} f(i) * t_{off} \\ & + \sum_{ph \in pre-head(L)} f(ph) * (t_i + t_l * |mfi(L)|) \end{aligned}$$

$$CF(L) = WCET\_offchip(L) - WCET\_onchip(L) \quad (1)$$

A positive value of $CF(L)$ means that enough WCET improvement is expected to compensate the reload cost. The pre-headers of the loops with positive values of $CF(L)$ are selected as reload points. It may be remarked that selection of reload points only depends on the code structure and on basic hardware parameters (on-chip and off-chip memory latencies); it does not depend on the considered on-chip memory (locked cache or scratchpad).

2.2.2 Selection of on-chip memory contents

Selection of cache contents is based on frequency information along the WCEP. Since loading and locking a value into the instruction cache may change the WCEP, the WCEP is re-evaluated regularly. The algorithm for selection of cache contents is sketched below.

```
1  ToBePlaced = ListBasicBlocs;
2  (WCET,WCEP) = evaluate_WCET();
3  ListBB = SelectMostBeneficialBB(ToBePlaced,N);
4  while | ListBB | ≠ 0 do
5      for each bb in ListBB do
6          ListReloadPoints = getPoints(BB);
7          for each rp in ListReloadPoints do
```

```
8            Load(bb,rp);
9         end for
10     end for
11     (WCET,WCEP) = evaluate_WCET();
12     if WCET > WCET_previous_iteration return;
13     ListBB = SelectMostBeneficialBB(ToBePlaced,N);
14 end while
```

The algorithm fills progressively the on-chip memory at the reload points identified in § 2.2.1. This is done by considering successively all the program basic blocks, starting from the one with the maximum expected decrease of WCET estimate. Initially (line 1), the set of basic blocks to be considered (list *ToBePlaced*) includes all basic blocks of the program. All reload points have an associated empty content.

The algorithm then proceeds iteratively. At a given iteration, the group formed by the N most *beneficial* basic blocks are considered for locking (N is an algorithm parameter, defining how often the WCEP is re-evaluated). The notion of benefit of a basic block (function *SelectMostBeneficialBB*) is simply the execution frequency of the basic block along the worst-case execution path. The higher is the frequency, the higher is the chance that the basic block is loaded into on-chip memory.

The inner loop of the algorithm (lines 6 to 9) is dedicated to the loading of basic block *bb*. First we get the list of reload points at which *bb* may be loaded (line 6). Function *getPoints*, not detailed here for space considerations, returns the list of reload points directly dominating *bb* (reload points are arranged into an inter-procedural domination tree). The actual loading of the basic block is achieved by function *Load*, which differs depending on the type of memory under consideration (locked cache vs scratchpad, see below). The algorithm iterates until locking new basic blocks does not result in improvements of WCETs anymore, or until there are no more basic blocks to be considered (line 11 and 12). WCETs and WCEPs are estimated thanks to an external WCET estimation tool.

The WCEP and the cost function are re-evaluated regularly, after having considered the placement of N% of basic blocks (line 13). The lower is the value of N the better is the estimation of the WCEP along the whole algorithm and the better is the quality of the cache contents (but the longer is the execution time of content selection).

The differences between the loading of basic block into a locked cache and into a scratchpad memory are hidden in function *Load*:

- **Locked cache.** In this case, function *Load* allocates information in the locked cache on a per cache block basis. *Load* scans all program lines of the basic block. It inserts a program line *pl* if there is a free (not yet filled-in) cache block in the k ways *pl* is mapped onto. There is no modification of the memory layout of the application (see [Pua06] for more details on implementation considerations).

- **Scratchpad memory.** In the case of a scratchpad memory, function *Load* allocates information on a per basic block basis. *Load* uses a first-fit allocation strategy to

find a free block of the basic block size into the scratchpad and jointly determine the address where the basic block will be copied at run-time. It may be remarked that the first-fit allocation strategy is used at compile-time, contrary to more current use of such allocation strategies at run-time.

2.2.3 Implementation issues

Loading the contents of on-chip memory at reload points needs to activate some function at run-time to load the contents of on-chip memory. The activation of the load function must not change the code memory layout after the contents of on-chip memory is selected. Otherwise there would be a mismatch between the addresses of the memory areas selected by the content selection procedure and the addresses of the same areas after adding activations of the load function.

To avoid such a mismatch, we avoid any modification of the code memory layout through the use of one of the two techniques sketched below. One technique is to use the processor debug registers to raise an exception when a reload point is reached. Since possible reload points are not so numerous, another possibility is to initially (before reload points on on-chip memory contents are selected) reserve some place for a function call for *every possible* reload point (ie inserting a *nop* instruction at every loop pre-header). The place will be filled-in by a function call to the load procedure for *actual* reload points only.

2.2.4 Example

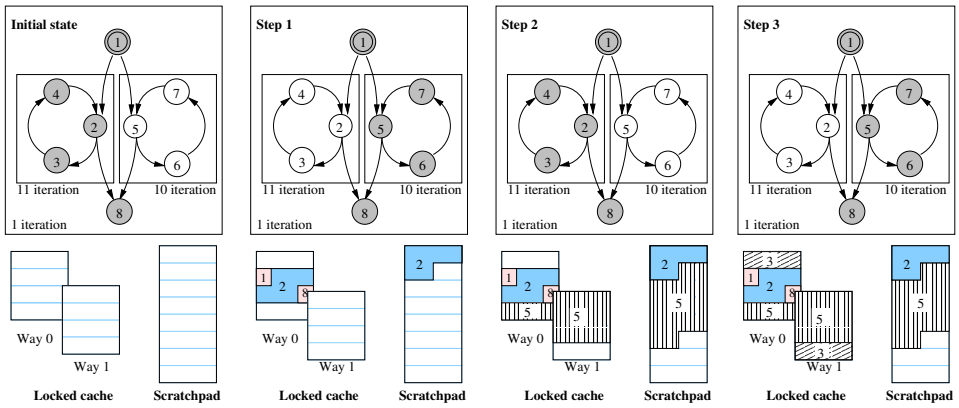


Figure 1: Content selection on an example

Figure 1 depicts three iterations of the algorithm on a toy example. We consider $N = 1$ (the WCEP is re-evaluated every time a basic block is considered for loading). The initial view depicts the locations where reload points are placed by the algorithm presented before

(here, for simplicity a single reload point at basic bloc number 1 is assumed). The initial WCEP, depicted as gray circles, is assumed to traverse the loop at the left. In the example, we consider a 2 ways set-associative cache with 4 cache blocks of 16 bytes each per way (total cache size = 128B) and a scratchpad memory of the same size.

At the first iteration, the most frequently executed basic block along the WCEP, here basic block number 2, is loaded into on-chip memory. In the case of a locked cache (left part of the subfigure), all program lines containing basic block 2, are locked, meaning that some instructions from the neighbour basic blocks (blocks 1 and 8) are locked as well. In the case of the scratchpad memory (right part of the subfigure) only the instructions of basic block 2 are loaded into the scratchpad memory. In both cases, loading basic block 2 makes the WCEP changes and now traverses the loop at the right.

At iteration 2, the most frequently executed basic block is basic block number 5. It is loaded on a per-cache-block or entire basic block basis depending on whether a locked cache or a scratchpad memory is considered. Note the first-fit allocation strategy in the case of scratchpad memory allocation.

Due to the effect of locking basic block 5, the WCEP then changes at iteration 3 again, and basic block 3 is then the mostly executed block along the WCEP. Assuming basic block 3 is more than 32 bytes large, it can not be loaded entirely into the scratchpad memory, but some of its program lines can be locked in the case of the locked cache.

This process iterates until the WCET estimate does not improve anymore. In the case of the locked cache, the process stops at iteration 3, because the locked cache is entirely filled-in. More iterations may be required in the case of the scratchpad memory, since the loading of more (small) basic blocks may still improve the WCET esimate.

3 Dynamically locked caches vs scratchpad memories: a qualitative comparison

As far as the contents of the locked cache or scratchpad is selected at compile time, both schemes are predictable. The outcome of every memory access (on-chip access or off-chip access) is known off-line. One interesting consequence of this aspect is that timing anomalies as defined in [LS99b] (a cache miss may in some cases result in a shorter execution time than a hit) do not occur anymore. Several factors related to the nature of the on-chip memory (scratchpad vs locked cache) are expected to impact the worst-case performance of tasks:

Volume of "predictable" memory available. As for processors with both on-chip cache(s) and on-chip scratchpad memory, the volume of memory available respectively for scratchpad and cache(s) is in general of the same order of magnitude (see Table 1 for an illustration on a small set of processors).

Name	Instruction Cache	Data Cache	Scratchpad Memory
Motorola Coldfire 5206e	512B	None	512B
Motorola Coldfire 5272	1KB	None	4KB
Motorola Coldfire 5249	8KB	None	32 and 64KB
ARM 940T	4KB	4KB	None
ARM 1020	32KB	32KB	None
Infineon Tricore TC1912	8KB	8KB	24KB (code) + 24KB (data)
MPC5567	8KB (unified)		64KB
MPC5554	32KB (unified)		64KB
MPC7410	32KB	32KB	None
Raza MIPS64 XL7100	32KB	32KB	None

Table 1: On-chip memory available in some processors

Besides L1 caches, the memory hierarchy usually comprizes larger on-chip or off-chip caches (L2 and possibly L3 caches) as well. When the L2 and L3 caches have locking capabilities (for instance the Raza MIPS64 XL7100 mentioned in the Table above the L2 and L3 cache may be locked on a per cache block basis), a large volume of information may be locked, as compared to the smaller amount of information usually storable in on-chip scratchpad memories. Note that the quantitative comparison given in next section focuses on on-chip L1 caches and scratchpad memories only.

Worst-case performance. Besides predictability considerations, it is crucial in hard real-time systems to have software with good worst-case performance (as low as possible WCET estimates without sacrificing safety). Assuming the algorithm presented in section 2 is used for selecting on-chip memory contents, the nature of memory (scratchpad vs locked cache) impacts the worst-case performance of tasks because of several factors, listed below:

Name	Description	Code size (bytes)	Nb. of BBs	Average BB size (bytes)	Nb loops
adcpm	Adaptive differential pulse code modulation	8504	265	32	17
compress	Compression of a 128 x 128 pixel image using discrete cosine transform	3056	115	27	12
des	des and triple-des encryption/decryption algorithm	11068	229	48	13
jfdctint	JPEG slow-but-accurate integer implementation of the forward DCT (Discrete Cosine Transform)	3608	49	73	3
minver	Matrix inversion for 3x3 floating point matrices	4520	135	33	17

Table 2: Task characteristics

- **Adressing scheme.** When using a locked cache, the location of information in the cache is transparent to software. It is entirely under hardware control. The positive aspect is that no modification of the code layout is required when a basic block is locked into the instruction cache. The negative impact is that since the placement in the cache is under hardware control, there may be *conflicts* for cache locations. For instance, two basic blocks with the same address modulo the cache size for a direct-mapped cache cannot be locked simultaneously. This problem does not arise when using a scratchpad, since address selection is under software control.

- **Granularity of allocation.** The smallest locking unit in a locked cache is the cache block. If no modification of the code layout is done, basic blocks may not be aligned on cache block boundaries. Thus, when locking the program lines of a basic block, extra instructions may be locked as well. As these instructions are not necessarily on the WCEP, they are not necessarily the most interesting instructions to lock. This problem of *pollution* is expected to show up with large cache blocks. This issue does not arise when allocating code in scratchpad memory, since the size of allocated blocks in scratchpad memory is under software control.

  When allocating information in scratchpad memory, the location of the piece of information in memory is under software control. Thus, in order to keep the implementation cost low, the most natural allocation unit is a contiguous zone of code (here, we have selected the basic block as allocation unit, entire functions could have been selected instead). As a consequence, some space may be wasted when the basic blocks to be allocated are too large to be allocated in the left free space in scratchpad memory. This *fragmentation* issue is expected to arise in applications with large basic blocks. We expect the problem to be more acute when allocating data, because some big data structures may be candidate to scratchpad allocation. Note that the problem will not occur when using locked caches, since locking is done at the cache block granularity

with no need for changing the basic block addresses: for large basic blocks, only the program lines fitting into the locked caches are locked, even if the entire block does not fit into the cache.

The phenomenon having an impact on task WCET estimates are exhibited and quantified in the next section.

4 Dynamically locked caches vs scratchpad memories: a quantitative comparison

This section compares the worst-case performance (worst-case execution times, ratios of on-chip and off-chip memory accesses) of benchmark applications using respectively a locked cache and a scratchpad memory. No comparison with unlocked cache is made because this is part of our previously published work [Pua06].

4.1 Experimental setup

Our interest here is to evaluate the differences between dynamic allocation in locked caches and dynamic allocation in scratchpad memory. As we consider hard-real time systems, we focus on *worst-case* performance, estimated off-line without executing the code. Results are given on a per-task basis. The performance metrics we use are the task WCET and the ratios of on-chip and off-chip memory accesses along the *worst-case* execution path. To isolate the impact of the memory hierarchy, the WCET estimates given in the rest of this section only account for memory accesses (on-chip/off-chip), assuming that an off-chip access takes 10 cycles as compared to on-chip latencies of 1 cycle. The figures thus volontarily ignore architectural elements other than memory hierarchy (pipelining, branch-prediction) to be as architecture-independent as possible.

Our experiments were conducted on MIPS R2000/R3000 binary code, but we are actually independent of any specific MIPS-compatible processor since our focus is on instruction caches and scratchpad memory only. We consider an instruction cache with $S_B = 16$ bytes large blocks (4 instructions). The cache associativity degree can be parametrized (from a direct-mapped cache to a fully associative cache). By default, the cache size and the scratchpad size is 1KB, t_i=0 and t_l accounts for one off-chip access per block of 16B.

The WCETs of tasks are computed by the Heptane[2] static WCET analysis tool [CP01]. One may configure Heptane to estimate WCETs using either: a tree-based method, through a bottom-up traversal of the syntactic tree of the analyzed C programs; an Implicit Path Enumeration Technique (IPET) method [PK89], generating a set of linear constraints from the program control-flow graph. Here, the IPET WCET estimation method is used.

Heptane includes hardware modeling capabilities to estimate WCETs for programs running on architectures with unlocked instruction caches. Heptane may also import XML files describing the contents of locked instruction caches or the contents of a scratchpad memory, selected off-line.

The experiments were conducted on five benchmark tasks, whose features are summarized in Table 2. All benchmarks except compress are benchmarks maintained by the Mälardalen WCET research group[3]. Compress is from the UTDSP Benchmark suite[4].

[2]Heptane is an open-source static WCET estimation tool available at *http://www.irisa.fr/aces/software/software.html*.

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

[4]http://www.eecg.toronto.edu/

Task	On-chip ratio	Off-chip ratio	Reload ratio	WCET (cycles)
adpcm locked	76.0%	24.0%	4.4%	42861
adpcm scratchpad	83.1%	16.9%	7.1%	39769
compress locked	98.8%	1.2%	8.2%	26773754
compress scratchpad	99.2%	0.8%	8.8%	27039482
des locked	85.8%	14.2%	2.3%	10656840
des scratchpad	85.5%	14.5%	3.6%	11028095
jfdctint locked	69.5%	30.5%	1.1%	45278
jfdctint scratchpad	60.4%	39.6%	0.9%	54533
minver locked	91.0%	9.0%	13.1%	35392
minver scratchpad	93.5%	6.5%	14.9%	34938

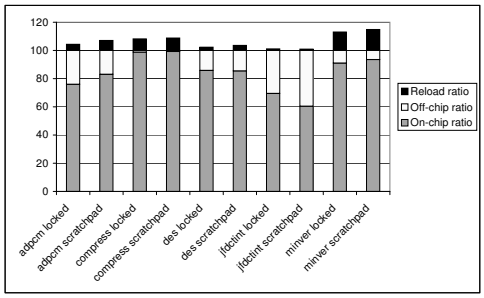


Figure 2: On-chip/Off-chip/reload ratios for locked caches & scratchpad memories (ratios=$\frac{n_{on-chip/off-chip/reload}}{n_{on-chip}+n_{off-chip}}$, %)

The content selection algorithm is run with $N = 10$, meaning that the WCEP is re-evaluated after placing 10% of the basic blocks.

4.2 Basic experiments

We study the number of on-chip and off-chip memory accesses for a direct-mapped locked instruction cache of 1KB, compared with a scratchpad memory of 1KB. Blocks in scratchpad memory are aligned on instruction boundaries (4 bytes). The results are expressed in Figure 2 in terms of ratios of categories of memory accesses ($\frac{n_{on-chip}/n_{off-chip}/n_{reload}}{n_{on-chip}+n_{off-chip}}$) and WCET estimate.

The major conclusion that can be drawn is that for most benchmarks, WCET estimates when using a locked instruction cache and a scratchpad memory are very close to each other. Furthermore, for most benchmarks, the amount of extra memory accesses for reloading the on-chip memory is acceptable. One may also remark that the ratio of on-chip memory accesses for both types of on-chip memory is good if applications exhibit temporal locality, which is the case for all the benchmarks considered in this paper[5]. In addition, task size does not have a direct impact on ratios of on-chip memory accesses: the biggest application *des*, whose code is 11 times the cache size, exhibit a better ratio of on-chip memory accesses than smaller applications (*adpcm, jfdctint*). This is because there are several contents of on-chip memory associated to each task (contents on-chip memory, while selected off-line change at run-time).

The reminder of this section further details the reasons why, in some cases, scratchpad may result in tigher WCET estimates than locked caches or vice-versa.

4.3 Impact of cache block size

Table 3 shows the impact of the cache block size. For that purpose, we used three different sizes for cache blocks: 8B, 16B and 32B (the total cache capacity is kept to 1KB). In this section, we have used a fully-associative cache, in order to focus on the impact of cache block size only (conflicts for cache block locations do not exist).

What can be seen from the results is that an increase of the cache block size always results in an increase of the WCET estimates. This phenomenon comes from the fact that the cache is locked on a cache-block basis, resulting in the loading of program lines belonging to basic blocks which are not necessarily on the WCEP (pollution issue raised in the previous section). For some benchmarks (*adpcm, compress*) pollution increases the ratio of on-chip memory accesses. The reason is that extra instructions locked because of pollution do not prevent more interesting instructions to be locked. Anyway, in all situations, the reload cost gets higher when increasing cache block size, because more instructions than strictly necessary are actually locked. All in all, the pollution issue arising with locked caches, although easy to exhibit, only has a slim impact on WCET estimates.

[5]In [Pua06], it is shown that worst-case performance can degrade significantly for applications spatial locality only and no temporal locality.

Task	On-chip ratio	Off-chip ratio	Reload ratio	WCET (cycles)
adpcm locked (8B)	78.1%	21.9%	3.3%	40606
(16B)	78.5%	21.5%	3.6%	40591
(32B)	78.6%	21.4%	3.8%	40660
adpcm scratchpad	83.1%	16.9%	7.1%	39769
compress locked (8B)	99.21%	0.79%	9.03%	27393624
(16B)	99.23%	0.77%	9.41%	27894260
(32B)	99.25%	0.75%	9.77%	28372360
compress scratchpad	99.2%	0.8%	8.8%	27039482
des locked (8B)	88.6%	11.4%	3.7%	9918470
(16B)	88.4%	11.7%	3.8%	10043796
(32B)	87.8%	12.4%	4.1%	10360504
des scratchpad	85.5%	14.5%	3.6%	11028095
jfdctint locked (8B)	70.4%	29.4%	1.2%	44080
(16B)	69.5%	30.5%	1.2%	45278
(32B)	69.3%	30.7%	1.2%	45530
jfdctint scratchpad	60.4%	39.6%	0.9%	54533
minver locked (8B)	93.2%	6.8%	15.6%	36058
(16B)	93.6%	6.4%	16.9%	37073
(32B)	94.1%	5.9%	19.7%	39770
minver scratchpad	93.5%	6.5%	14.9%	34938

Table 3: Impact of cache block size (ratios=$\frac{n_{on-chip/off-chip/reload}}{n_{on-chip}+n_{off-chip}}$, %)

The pollution problem could be (partially) removed by aligning basic blocks on cache block boundaries, at the cost of a larger code size.

4.4 Impact of associativity degree

Table 4 shows the impact of the associativity degree on the ratio of off-chip and on-chip memory accesses.

The table contents shows that, similarly to unlocked caches, locked caches are subject to conflicts for cache block locations. This is because the placement of a program line into the cache is under hardware control and is constrained by the cache structure.

4.5 Impact of basic block size

Finally, we examine in Table 5 the impact of basic blocks size on worst-case performance, which turned out to be the factor with the biggest impact on worst-case performance. The study is done on the *jfdctint* benchmark, using either a fully-associative cache of 1KB or a scratchpad of 1KB as well. Two versions of the benchmark, with the same functionality, are studied:

- a version with small basic blocks (original version), in which the code of the two inner loops are mainly made of calls to a function with a very small body (a couple of C statements)

Task	On-chip ratio	Off-chip ratio	Reload ratio	WCET (cycles)
adpcm locked (direct)	76.0%	24.0%	4.4%	42861
(2-ways)	78.5%	21.5%	3.6%	40504
(4-ways)	83.3%	16.1%	7.4%	39656
(fully-asso)	78.5%	21.5%	3.6%	40591
adpcm scratchpad	83.1%	16.9%	7.1%	39769
compress locked (direct)	98.8%	1.2%	8.2%	26773754
(2-ways)	99.2%	0.8%	9.4%	27893882
(4-ways)	99.2%	0.8%	9.4%	27893882
(fully-asso)	99.2%	0.8%	9.4%	27894260
compress scratchpad	99.2%	0.8%	8.8%	27039482
des locked (direct)	85.8%	14.2%	2.3%	10656840
(2-ways)	88.4%	11.5%	3.8%	10036326
(4-ways)	88.4%	11.5%	3.8%	10043796
(fully-asso)	88.4%	11.5%	3.8%	10043796
des scratchpad	85.5%	14.5%	3.6%	11028095
jfdctint locked (direct)	69.5%	30.5%	1.1%	45278
(2-ways)	69.5%	30.5%	1.1%	45278
(4-ways)	69.5%	30.5%	1.1%	45278
(fully-asso)	69.5%	30.5%	1.1%	45278
jfdctint scratchpad	60.4%	39.6%	0.9%	54533

Table 4: Impact of associativity degree (ratios=$\frac{n_{on-chip/off-chip/reload}}{n_{on-chip}+n_{off-chip}}$, %)

- a modified version with large basic blocks, in which the function bodies are inlined in the callees. As a consequence, the code of the two inner loops is now mainly composed of a big basic blocks of around 1.5KB.

Task	On-chip ratio	Off-chip ratio	Reload ratio	WCET (cycles)
jfdctint locked (small BB)	71.1%	28.9%	1.1%	43598
(big BB)	68.7%	31.3%	1.5%	35370
jfdctint scratch. (small BB)	60.4%	39.6%	0.9%	54533
(big BB)	32.2%	67.8%	0.5%	63689

Table 5: Impact of Basic block size (ratios=$\frac{n_{on-chip/off-chip/reload}}{n_{on-chip}+n_{off-chip}}$, %)

The results show that locked caches are not very sensitive to the size of basic blocks, since locking is done at a granularity which is independent of the size of basic blocks. The increase of WCET estimates when analyzing the version without inlining is explained by the times required for function calls and parameter passing, which do not exist with the other code version.

On the contrary, the results depicted in table 5 show that scratchpads are very sensitive to basic block size because of fragmentation. On this example, the ratio of on-chip memory accesses drops drastically (from 60.4% to 32.2%) because a single big basic block cannot be loaded into scratchpad memory because of memory fragmentation. This phenomenon

appears when loading code when basic blocks are large, which is rather rare in practice except when inlining is used for performance considerations. Fragmentation could be much more common if dynamically loading data structures such as big arrays.

5 Related work

A large amount of research has been undertaken to make an efficient use of the memory hierarchy both for general purpose and real-time and/or embedded systems.

In the field of real-time systems, the increasing use of caches has motivated research into cache-aware static WCET estimation methods, mainly for instruction caches. For architectures without timing anomalies, their objective is to determine for every memory access if it *will certainly* cause a cache hit or *may* cause a miss. Cache-aware WCET computation methods can be based on data-flow analysis [Mue00], abstract interpretation [HLTW03], integer linear programming techniques [LMW96], or symbolic execution [LS99a]. In comparison, much less work has tackled data caches because of the presence of dynamic references (i.e. arrays and pointers). Cache-aware WCET estimation techniques reach their limit for some cache replacement policies (e.g. random, pseudo round-robin, pseudo LRU), which cannot be tightly predicted, and for dynamically scheduled processors, for which timing anomalies arise. In such situations, a statically-decided software control of the cache, through cache locking techniques, is of interest. Work on locking techniques has been done for instruction caches [PD02, CPIM05] and data caches [VLX03]. [PD02, CPIM05] study static locking techniques. While such techniques provide good worst-case performance for small tasks, their performance decreases dramatically when the task working set exceeds the cache size. The work presented in [Pua06], which served as a basis for this paper, extends locking of instruction caches to a more dynamic locking, for which on a per-task basis, regions in the task code are detected off-line and associated to a given cache contents to be loaded at runtime when entering the region. Some other studies on locked caches have been undertaken by Vera et al (for instance see [VLX03]). In these studies, the focus is on data caches, and the authors make a combined use of cache analysis and cache locking.

As for scratchpad memories, they have been the subject of many researches in the context of embedded systems. The objective of existing studies were to make an efficient (static or dynamic) use of scratchpad memories to minimize energy consumption [BSL+02, SGW+02, VWM04] or average-case performance [UB03, KRI+01, LGX05]. The problem of optimizing the contents of scratchpad memory contents with respect to worst-case performance is subtly different to optimizing the contents of scratchpad memory contents with respect to average-case performance or memory consumption. Indeed, the worst-case execution path does not necessarily coincide with the average-case execution path. In addition, the effect of loading some information may cause the WCEP to change, which does not appear when optimizing with respect to average-case performance/energy consumption. To the best of our knowledge, only [SMRC05] addresses the problem of allocating information in scratchpad memory with the objective of reducing the WCET estimate. Their method allocates data for the whole lifetime of a program, which should result in scalability problems for program whose data size is larger than scratchpad memory size. Our proposal gives a more dynamic management of scratchpad contents for instruction, and quantitatively compares the resulting performance to the one of a dynamically locked cache.

6 Concluding remarks

We have proposed in this paper an algorithm for off-line selection of the contents of on-chip memory. The proposed algorithm supports both locked caches and scratchpad memories. On-chip memory contents are selected off-line for the sake of predictability. Moreover, the contents of on-chip memory, while selected off-line, is changed at run-time, for the sake of scalability with respect to code size.

Experimental results show that the algorithm yields to good ratios of on-chip memory accesses along the worst-case execution path, with a tolerable reload overhead, for both types of on-chip memory. Furthermore, we have highlighted the circonstances under which one type of on-chip memory is more appropriate than the other: worst-case performance with scratchpad memories may degrade when loading large information due to the scratchpad memory fragmentation; worst-case performance with locked caches may slightly degrade with large cache lines, due to a phenomenon of pollution (not-so frequent program lines may be locked because of the cache line locking granularity).

Extending our proposal to accesses to data is easy if the addresses of referenced data are known off-line. In our opinion, the extension is straightforward to achieve for accesses to global data and stack-allocated data with programs with no recursion, which is common in embedded programs. Dealing with dynamically-allocated data and pointers, especially when there is aliasing, is harder to implement and is left as future work.

References

[Ber06] C. Berg. PLRU cache domino effects. In *6th International Workshop on Worst-Case Execution Time Analysis, in conjunction with the 18th Euromicro Conference on Real-Time Systems*, Dresden, Germany, July 2006.

[BSL+02] L. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory : a design alternative for cache on-chip memory in embedded systems. In *Proceedings of Tenth International Workshop on Hardware/Software Codesign (CODES 2002)*, May 2002.

[CP01] A. Colin and I. Puaut. A modular and retargetable framework for tree-based WCET analysis. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, The Netherlands, June 2001.

[CPIM05] A. M. Campoy, I. Puaut, A. P. Ivars, and J. V. B. Mataix. Cache contents selection for statically-locked caches: An algorithm comparison. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 49–56, Palma de Mallorca, Spain, July 2005.

[HLTW03] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7), July 2003.

[KRI+01] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayil, and A. Parikh. Dynamic management of scratch-pad memory space. In *Proc. of the 38th Design Automation Conference (DAC'01)*, December 2001.

[LGX05] L. Li, L. Gao, and J. Xue. Memory coloring: A compiler approach for scratchpad memory management. In *Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[LMW96] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction cache. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS96)*, pages 254–263. IEEE, December 1996.

[LS99a] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, November 1999.

[LS99b] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *IEEE Real-Time Systems Symposium*, pages 12–21, 1999.

[Mue00] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2):217–247, May 2000.

[PB00] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2-3):115–128, May 2000. Guest Editorial.

[PD02] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS02)*, pages 114–123, Austin, Texas, December 2002.

[PK89] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, September 1989.

[Pua06] I. Puaut. WCET-centric software-controlled instruction caches for hard real-time systems. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, Dresden, Germany, July 2006.

[SGW+02] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS 2002)*, pages 213–218, Kyoto, Japan, 2002.

[SMRC05] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET centric data allocation to scratchpad memory. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS05)*, December 2005.

[UB03] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, November 2003.

[VLX03] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *ACM International Conference on Measurement and Modeling of Computer Systems (Sigmetrics 2003)*, 2003.

[VWM04] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. In *Proceedings of Design Automation and Test in Europe (DATE)*, Paris, France, February 2004.