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

Isabelle Puaut and Christophe Pais

Université de Rennes I/IRISA
Campus Universitaire de Beaulieu, 35042 RENNES Cedex - France
E-mail: {Isabelle.Puaut|Christophe.Pais}@irisa.fr

Abstract

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. 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).

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 to 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 [13].

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 [3]. 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 [10, 3, 6, 7]. 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) [3, 1]. Lastly, caches are sources of *timing anomalies* in dynamically scheduled processors [8] (a cache miss may in some cases result in a shorter execution time than a hit). In such situations, *cache 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 [11] 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 [4, 15, 5, 16]. Except [14], all these techniques aim at reducing the average execution time (ACET) of programs, through memory access profiles. Such ACET-oriented techniques are not necessarily suited for real-time systems, since the execution path followed in average may not be the worst-case execution path. Only [14] aims at optimizing tasks worst-case performance. However, in that study, scratchpad allocation 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:

978-3-9810801-2-4/DATE07 © 2007 EDAA

- 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 previously proposed in [11] for off-line selection of the contents of locked instruction caches.
- We give a 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 focuses on dynamic loading of *code* into scratchpad memories and locked caches only.

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 is devoted to a study of the impact of the type of on-chip memory (cache vs scratchpad memory) on the WCETs of tasks, as well as on the ratios of on-chip memory accesses along the worst-case execution path. Finally, we conclude in Section 4.

2 Selection of on-chip memory contents

This section presents an algorithm[1] 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 reloaded at run-time; (ii) *memory contents*, which are the pieces of code to be loaded at run-time when control reaches the reload point. As a result, the code of applications is divided into *regions* at the entry of which the contents of the on-chip memory is loaded. On-chip memory contents is selected thanks to the knowledge of execution frequencies of basic blocks along the worst-case execution path (WCEP), obtained through an external WCET estimation tool. Furthermore, since the worst-case execution path may vary when a piece of code is selected 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.

[1] An example of this algorithm as well as implementation issues can be found in the extended version of this paper *Scratchpad memories vs locked caches in hard real-time systems: a qualitative and quantitative comparison*. It can be downloaded at http://www.irisa.fr/centredoc/publis/PI/2006/irisapublication.2006-10-10.2735384409.

2.1 Notations and assumptions

We consider a CPU with 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 hardware-imposed 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 a piece of code of size sz into on-chip memory is a linear function of sz ($t_{reload} = a + b * sz$). Parameters t_{on}, t_{off}, b are taken from the hardware manuals (processor, system). b represents the cost per byte and a represents the software cost resulting from the call of the reload procedure.

Only reducible loops are currently supported.

2.2 Content selection algorithm

The algorithm is made of two independent parts: selection of reload points (§ 2.3) and selection of on-chip memory contents (§ 2.4).

2.3 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; $m_f(L)$ denotes the most frequently executed instructions of loop L along the WCEP, $instr(L)$ denotes the whole set of instructions of loop L, and $pre\_head(L)$ denotes the pre-headers of loop L.

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

$$WCET\_onchip(L) = \sum_{i \in m_f(L)} f(i) * t_{on} + \sum_{i \in instr(L) - m_f(L)} f(i) * t_{off} + \sum_{i \in pre\_head(L)} f(i) * (a + b * |m_f(L)|)$$

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

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).

2.4 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 on-chip memory may change the WCEP, it 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 § 2.3. 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 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*:

- In the case of a locked cache, 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 [11] for more details on implementation considerations).
- In the case of a scratchpad memory, function *Load* allocates information on a per basic block basis[2]. *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 runtime.

3 Dynamically locked caches vs scratchpad memories: a quantitative 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 the predictabiliy issues raised by timing anomalies as defined in [8] (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:

- **Addressing scheme.** When using a locked cache, the location of information in the cache is transparent to soft-

[2] One could consider allocating memory areas smaller than the basic block, by splitting basic blocks. We did not explore this direction in a first step because it results in extra complexity to decide which basic blocks should be splitted and where.

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 1. Task characteristics

ware. 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 there is no lower bound on the size of allocated blocks in scratchpad memory.
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, a entire basic block). 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.

These phenomenon are exhibited and quantified below, through a comparison of 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 previously published work [9, 12, 11].

3.1 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 1 KB, t_i=0, t_l accounts for one off-chip access per block of 16 bytes.

The WCETs of tasks are computed by the Heptane static WCET analysis tool [2].

Five benchmark tasks have been used (see Table 1 for a summary of the tasks features). All benchmarks except compress are maintained by the Mälardalen WCET research group[3]. Compress is from the UTDSP Benchmark suite[4].

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

3.2 Basic experiments

We study the number of on-chip and off-chip memory accesses for a direct-mapped locked instruction cache

[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

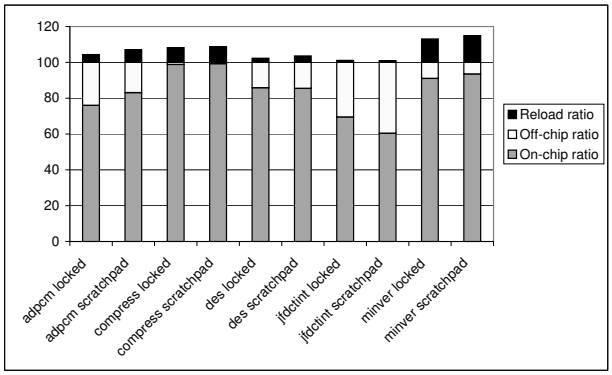

Figure 1. On-chip/Off-chip/reload ratios for locked caches & scratchpad memories

of 1 KB, compared with a scratchpad memory of 1 KB. Blocks in scratchpad memory are aligned on instruction boundaries (4 bytes). In the following, results are expressed 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 from the quantitative results, given in Figure 1, 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. 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 focus on the reasons why, in some cases, scratchpad may result in tigher WCET estimates than locked caches or vice-versa.

3.3 Impact of cache block size

Table 2 shows the impact of the cache block size. For that purpose, we used two different sizes for cache blocks: 8B, and 32B (the total cache capacity is kept to 1 KB). 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

Task	On-chip ratio	Off-chip ratio	Reload ratio	WCET (cycles)
adpcm locked (8B)	78.1%	21.9%	3.3%	40606
(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
(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
(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
(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
(32B)	94.1%	5.9%	19.7%	39770
minver scratchpad	93.5%	6.5%	14.9%	34938

Table 2. Impact of cache block size

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.

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

3.4 Impact of basic block size

Finally, we examine in Table 3 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 1 KB or a scratchpad of 1 KB as well. Two versions of the benchmark, with the same functionality, are studied: (i) 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) (ii) 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 1.5 KB.

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 3. Impact of Basic block size

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 3 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 with large basic blocks, 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.

4 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. 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 circumstances under which one type of on-chip memory is more appropriate than the other. On the one hand, worst-case performance with scratchpad memories may degrade when loading large information due to the scratchpad memory fragmentation. Splitting basic blocks may reduce fragmentation, at the cost of extra-complexity in the selection process. Exploring the impact of basic block splitting is left as future work. On the other hand, 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). Our future work will focus on allocation of data.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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)*, Dec. 2001.

[5] 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.

[6] 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, Dec. 1996.

[7] 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, Nov. 1999.

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

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

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

[11] 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.

[12] 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, Dec. 2002.

[13] 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.

[14] 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)*, Dec. 2005.

[15] 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*, Nov. 2003.

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