Combining Prefetch with Instruction Cache Locking in Multitasking Real-Time Systems

Luis C. Aparicio*, Juan Segarra*, Clemente Rodríguez† and Víctor Viñals*
*Dpt. Informática e Ingeniería de Sistemas, Universidad de Zaragoza, España
Instituto de Investigación en Ingeniería de Aragón (I3A), Universidad de Zaragoza
Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC)
Email: {luisapa, jsegarra, victor}@unizar.es
†Dpt. Arquitectura y Tecnología de Computadores, Universidad del País Vasco, España
Email: acprolac@ehu.es

***Abstract*—In multitasking real-time systems it is required to compute the WCET of each task and also the effects of interferences between tasks in the worst case. This is complex with variable latency hardware usually found in the fetch path of commercial processors. Some methods disable cache replacement so that it is easier to model the cache behavior. Lock-MS is an ILP based method to obtain the best selection of memory lines to be locked in a dynamic locking instruction cache. In this paper we first propose a simple memory architecture implementing the *next-line tagged* prefetch, specially designed for hard real-time systems. Then, we extend Lock-MS to add support for hardware instruction prefetch. Our results show that the WCET of a system with prefetch and an instruction cache with size 5% of the total code size is better than that of a system having no prefetch and cache size 80% of the code. We also evaluate the effects of the prefetch penalty on the resulting WCET, showing that a system without prefetch penalties has a worst-case performance 95% of the ideal case. This highlights the importance of a good prefetch design. Finally, the computation time of our analysis method is relatively short, analyzing tasks of 96 KB with 10^{65} paths in less than 3 minutes.**

***Keywords*-WCET; prefetch; instruction cache;**

I. Introduction

Real-time systems require that tasks complete their execution before specific deadlines. Given hardware components with a fixed latency, the worst case execution time (WCET) of a single task could be computed from the partial WCET of each basic block of the task. However, in order to improve performance, current processors perform many operations with a variable duration. This is mainly due to speculation (control or data) or to the use of hardware components with variable latency. Branch predictors fall in the first category, whereas memory hierarchy and datapath pipelining belong to the second one. A memory hierarchy made up of one or more cache levels takes advantage of program locality and saves execution time and energy consumption by delivering data and instructions with an average latency of a few processor cycles. Unfortunately, the cache behavior depends on past references and it is required to know the previous accesses sequence in order to compute the latency of a given access in advance. Resolving these *intra-task* interferences is a difficult problem its own. Anyway, real-time systems usually work with several tasks which may interrupt each other at any time. This makes the problem much more complex, since the cost of *inter-task* interferences must also be identified and bounded. Furthermore, both these problems cannot be accurately solved independently, since the path that leads to the worst case of an isolated task may change when considering interferences. Cache locking tackles the whole problem by disabling the cache replacement, so the cache content does not vary. Specifically, for an instruction cache, the instruction fetch hits and misses depend on whether each instruction belongs to a cached and locked memory line and not on the previous accesses.

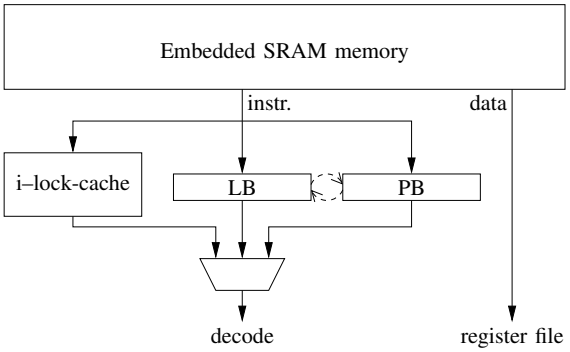


Figure 1. Memory architecture. The LB/PB roles are switched after issuing a prefetch.

In this paper we focus on the instruction fetch path. We introduce an implementation of *next-line tagged* sequential prefetching for real-time processors (Figure 1). It fetches the next memory line in physical order. This prefetch buffer (PB) is combined with a line buffer (LB) and a lockable instruction cache, and its hardware complexity is very low.

This work was supported in part by grants TIN2007-66423 (Spanish Government and European ERDF), gaZ: T48 research group (Aragón Government and European ESF), Consolider CSD2007-00050 (Spanish Government), and HiPEAC-2 NoE (European FP7/ICT 217068).

So, whereas the lockable instruction cache captures temporal locality, the LB captures spatial locality inside the current memory line and the PB extends it so that it is also captured outside the current line.

In order to use this architecture we extend Lock-MS (*Maximize Schedulability*), an ILP-based method that obtains the selection of memory lines to be loaded into the dynamic locking instruction cache that minimizes the worst overall execution cost (WCET plus preloading times at context switches) [1]. The obtained worst-case performance is compared with dynamic locking (Lock-MS) and static locking (Lock-MU, *Minimize Utilization*) methods without prefetch. Our results show that the usage of prefetch with a minimal instruction cache performs better than systems with very large instruction caches without prefetch.

This paper is organized as follows. In Section II we review the background and related work. Section III describes our proposed hardware architecture. The Lock-MS method is extended in Section IV to include the previous hardware architecture. Section V shows our results. Finally, Section VI presents our conclusions.

II. Related Work

Multitask preemptive real-time systems must be schedulable to guarantee their expected operation. That is, all tasks must complete their execution before their deadline. Considering a *fixed priority* scheduler, feasibility of periodic tasks can be tested in a number of ways [2]. Response Time analysis is one of these mathematical approaches, and fits very well as a schedulability test. This approach is based on the following equation for independent tasks:

$$R_i^{n+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \qquad (1)$$

where R_i is the response time, C_i is the WCET and T_i is the period of each task i, respectively. It is assumed that tasks are ordered by priority (the lower the i the higher the priority). This equation provides the response time of each task after a few iterations and it has been used in previous studies [3], [4], [5]. A task meets its real-time constraints if $R_i \leq D_i, \forall i$, being D_i the deadline.

The WCET of each task (C_i) is not easy to obtain in systems with cache memory. In the literature we find different methods to approach the WCET problem in the presence of caches [6]. These methods can be divided into those that analyze the normal behavior of the cache, and those which restrict its behavior to simplify the analysis.

The first kind of methods try to model each task and system as accurately as possible considering the dynamic behavior of the cache [7], [8], [9], [10], [11]. This analysis is very hard, since the worst path depends on the cache outcome, and the cache outcome affects the cost of each path. Due to this complexity, interferences among tasks are not usually considered and tasks are analyzed in isolation. So, complementary methods are needed to adapt the WCET of each isolated task to multitasking systems. This may be done by further analysis to add the number of inter-task interferences and their cost to the cost of each task [4], [5].

On the other hand, cache locking methods restrict the cache behavior by using the ability to disable the cache replacement, present in many commercial processors[1]. Having specific contents fixed in cache, the timing analysis is easier, so these methods can afford a full system analysis, i.e. several tasks on a real-time scheduler. Cache-locking techniques can also be divided into *static* and *dynamic cache locking*. Static locking methods preload the cache content at system start-up and fix this content for the whole system lifetime so that it never gets replaced [12], [13]. On the other hand, dynamic cache locking allows the tasks to disable and enable the cache replacement at will [14], [15], [16], [17]. Although there are studies which allow instruction cache reloading at any point [16], most of them restrict reloading to context switches [1], [14], [15]. These approaches call for a per-task selection of contents, with the drawback of preloading every time that a task enters the CPU. Our base method, Lock-MS (Maximize Schedulability) belongs to this type of methods. Lock-MS models can be set both in compact and path-explicit versions. Whereas the path-explicit version may be easier to understand and necessary to include execution-history dependent timings, the compact version offers a much better performance due to the model description being much shorter [1].

Most of the methods dealing with the instruction cache do not consider other variable-latency component in the instruction fetch path. Only a few of them include a line buffer, which improves performance and does not present relevant difficulties in its analysis (e.g. [1], [16]). Instruction prefetching involves predicting future program counter addresses and sending them to the next memory level in order to overlap execution and miss processing time. Much research has been done on the subject for general purpose processors, focusing on coverage, precision and timeliness, both from software and hardware point of view (see for instance [18], [19]). However, prefetch is not usually considered in real time frameworks, since its timing is not straightforward to model and it pollutes the cache. Sequential prefetching is the simplest and oldest hardware alternative. Three simple algorithms have been proposed, differing in the conditions that launch the prefetch. *Next-line always* launches the prefetch for line $i + 1$ every time that there is a reference to line i [20]. *Next-line on miss* launches it only if the access to the current line misses. The *next-line tagged* scheme issues a prefetch if the current fetch misses or it hits in a previously prefetched line [21]. All these schemes

[1]For instance, Motorola (ColdFire, PowerPC, MPC7451, MPC7400), MIPS32, ARM (904, 946E-S), Integrated Device Technology (79R4650, 79RC64574), Intel 960, etc.

can be extended in degree or distance (prefetch the next n lines or prefetch the line at a distance n, respectively) [22].

It is important to notice that one of the main difficulties in the WCET analysis in the presence of prefetch is that the cache gets polluted. However, since we work with a prefetch buffer and a locked cache, our cache content remains unchanged.

III. Hardware Architecture

Our memory hierarchy consists of a lockable instruction cache, a line buffer (LB) and a prefetch buffer (PB), as can be seen in Figure 1. These two single-line address-tagged buffers capture spatial locality and support the on-miss tagged sequential prefetching. The fetch lookup proceeds in parallel in the instruction cache, LB and PB. A hit in either structure delivers the instruction in a processor cycle, while a miss in the three structures leads to requesting the line to the next memory level (an embedded SRAM in our model), filling later the LB with the incoming line, and scheduling a sequential prefetching directed to the PB. In contrast to conventional tagged prefetching where the cache cannot be locked, in order to support the tagged prefetching, both buffers have a bit-tag to inform the prefetch controller of the first use of a prefetched line. Then, on such a first use, the prefetch controller looks up the cache (we assume a dedicated lookup port) and, if it misses, the prefetch is issued to the next level and the current PB/LB role is interchanged, being the current LB the new PB, and the current PB the new LB, which invalidates its content. Summarizing, the described memory system has three key advantages: it is simple, it is a suitable implementation of prefetching for a locking cache and, as we will see, it is amenable for use in an ILP-based WCET analysis. The only specific behavior to consider is that, on an LB/PB miss and cache hit, the buffers invalidate their content. This removes the potential dependencies on the previous path. Finally, we consider that our system has no additional sources of latency variability (data cache, branch predictor, out-of-order execution, etc.). Most embedded processors can operate under these considerations, which are also assumed in previous studies [12], [13].

IV. Supporting Hardware Prefetch in Lock-MS

We consider a multitask system with the memory architecture described in the previous section, so its behavior is completely predictable. Having a dynamic locking approach, the cache content is preloaded every time a task enters the CPU. In this way each task can take profit of the whole cache with the drawback of the preloading costs in context switches. We also consider that all loop bounds are known.

The aim of Lock-MS is to provide a selection of lines such that, when locked, the schedulability of the whole system is maximal. To obtain this selection of lines, Lock-MS considers the resulting WCET of each task, the effects of interferences between tasks and the cost of preloading the selected lines into cache. This method is based on *Integer Linear Programming* (ILP) [23], [24]. Thus, all requirements must be modeled as a set of linear constraints and then the resulting system is minimized.

The constraints in Lock-MS can be either path-explicit or compact. Whereas the compact model offers a superior performance, it may be harder to understand. So, we first describe the integration of prefetch into the path-explicit model and then we show how to transform it into the compact model.

Path-explicit model overview: The main idea of constraints is to define the worst overall cost of a task as the function to minimize. This worst cost of task i is the WCET plus the number of context switches ($ncswitch$) times the switch cost:

$$Wcost_i = wcet_i + ncswitch_i \cdot switchCost_i \qquad (2)$$

The WCET of a task i must be equal to the cost of its worst path, i.e. it must be equal or greater than any of its paths j:

$$wcet_i \geq pathCost_{i,j} \quad \forall\, 1 \leq j \leq NPaths_i$$

where, using the path-explicit Lock-MS constraints, the cost of each path can be computed as the sum of the cost of its particular memory lines:

$$pathCost_{i,j} = \sum_{k=1}^{NLines_{i,j}} lineCost_{i,j,k}$$

In turn, the cost of each memory line can be computed by knowing the number of instruction cache hits ($nIChit$) and misses ($nICmiss$) to the line k in the specific path j of task i and the hit and miss costs ($IChitCost$ and $ICmissCost$). Since the cache is locked, the number of hits and misses will be either zero or the number of accesses. This can be easily computed by knowing the total number of times that this line is accessed ($nfetch$) and whether its corresponding physical memory line is cached ($cached_l = 1$) or not ($cached_l = 0$).

$$\begin{aligned} lineCost_{i,j,k} &= IChitCost_{i,j,k} \cdot nIChit_{i,j,k} + \\ &\quad ICmissCost_{i,j,k} \cdot nICmiss_{i,j,k} \qquad (3) \\ nIChit_{i,j,k} &= nfetch_{i,j,k} \cdot cached_l \\ nICmiss_{i,j,k} &= nfetch_{i,j,k} - nIChit_{i,j,k} \end{aligned}$$

In a similar way, the switch cost to task i depends on the number of memory lines to preload in each context switch ($numcached$) times the preload cost of each single line ($ICpreloadCost$). Also, it depends on the possible penalties for flushing the LB/PB if they contain useful data.

$$\begin{aligned} numcached_i &= \sum_{l=0}^{Mlines-1} cached_l \\ switchCost_i &= ICpreloadCost \cdot numcached_i + \\ &\quad (tmiss_{LB} - thit_{LB}) + (tmiss_{PB} - thit_{PB}) \end{aligned}$$

Finally, the cache configuration (sets S and ways W) must be also specified:

$$W \geq cached_s + cached_{s+S} + cached_{s+2S} + \cdots \\ \cdots + cached_{s+nS} \quad \forall\, 0 \leq s < S,\ s \in \text{Integer}$$

So, minimizing $Wcost_i$ we get the selection of lines $cached_l$ (and the values of $wcet$, $Wcost$, $switchCost$, etc.) such that when preloaded and locked into cache, the worst overall cost of task i is minimal. For further details not related to our prefetch contribution, please refer to the paper describing Lock-MS [1].

Introducing prefetch to the cost of hits and misses: In order to compute the execution costs we assume a simple processor where instruction fetch and execution proceed sequentially. For a cached memory line containing one or more ($nIns$) instructions we can compute the total cost as the fetch cost plus the execution cost ($texec$). The fetch cost is the instruction cache memory hit time ($thit_{CM}$) times the number of instructions ($nIns$) in the line k.

$$IChitCost_{i,j,k} = texec_{i,j,k} + thit_{CM} \cdot nIns_{i,j,k}$$

Depending on the memory architecture, the penalties on cache misses may vary. Let us start assuming that we have a locked cache (replacements disabled) and no other component is present. In this case, the miss cost is the execution time of the instructions in the memory line, plus one cache miss time for each of them:

$$ICmissCost_{i,j,k} = texec_{i,j,k} + tmiss_{CM} \cdot nIns_{i,j,k}$$

Adding a line buffer, the cache miss cost is the sum of the execution time ($texec$), a single LB miss ($tmiss_{LB}$) and one LB hit ($thit_{LB}$) for the remaining instructions in the memory line ($(nIns - 1)$):

$$ICmissCost_{i,j,k} = texec_{i,j,k} + tmiss_{LB} + \\ thit_{LB} \cdot (nIns_{i,j,k} - 1)$$

Having cache memory, line buffer and hardware prefetch buffer, the computation of the cache miss cost is more complex. Prefetching implies fetching a memory line while the previous line is being executed, trying to overlap the current fetch with the execution of the preceding line, and thus hiding the fetch latency of the new memory line. According to *next-line tagged* prefetch, we assume that misses and first references to prefetched lines start prefetching the next memory line. Being a speculative action, the prefetch may succeed or not. Within a straight-code sequence sequential prefetching always gets the correct line, but on jumps or taken branches it fetches an incorrect line (except for jumps to the next physical line). In this latter case the correct memory line will be demanded to the next level and stored in the LB as in a system without prefetch. Thus, we cannot compute the cost of a memory line as above (eq. 3: cache hits plus cache misses), but we need to consider two cache miss cases: cache miss with prefetch hit and cache miss with prefetch miss.

$$lineCost_{i,j,k} = IChitCost_{i,j,k} \cdot nIChit_{i,j,k} + \\ PBCost_{i,j,k} \cdot nICmissPBhit_{i,j,k} + \\ LBCost_{i,j,k} \cdot nICmissPBmiss_{i,j,k}$$

So, instead of accounting the number of fetches of each memory line as above, it is needed to separate this account into sequential fetches, coming after the previous line in physical order ($nfetchInSequence$), and fetches after jumps ($nfetchAfterJump$). These constants can be easily obtained by statically parsing the code, as the number of fetches in the original Lock-MS method.

$$\begin{aligned} nfetch_{i,j,k} &= nfetchInSequence_{i,j,k} + \\ &\quad nfetchAfterJump_{i,j,k} \\ nIChit_{i,j,k} &= nfetch_{i,j,k} \cdot cached_l \\ nCmissPBhit_{i,j,k} &= nfetchInSequence_{i,j,k} \cdot \\ &\quad (1 - cached_l) \\ nCmissPBmiss_{i,j,k} &= nfetchAfterJump_{i,j,k} \cdot \\ &\quad (1 - cached_l) \end{aligned}$$

The cache miss cost with prefetch miss is the same as not having hardware prefetch, as above:

$$LBCost_{i,j,k} = texec_{i,j,k} + tmiss_{LB} + \\ thit_{LB} \cdot (nIns_{i,j,k} - 1)$$

Having a prefetch hit, the prefetch penalty not hidden ($PBPenalty$) is the time required to service the memory line from memory ($tmiss_{PB}$) minus the service (from LB or cache) and execution cost of the preceding memory line, i.e. the $IChitCost$ of the preceding line.[2] Since this penalty must be equal or greater than zero we translate it into two constraints:

$$\begin{aligned} PBPenalty_{i,j,k} &\geq tmiss_{PB} - IChitCost_{i,j,k-1} \\ PBPenalty_{i,j,k} &\geq 0 \\ PBCost_{i,j,k} &= texec_{i,j,k} + PBPenalty_{i,j,k} + \\ &\quad thit_{LB} \cdot (nIns_{i,j,k} - 1) \end{aligned}$$

Note that once a prefetched memory line is already stored in PB and the processor starts fetching from it, the PB role switches to the LB role. That is why all instructions have an access cost of $thit_{LB}$.

Let us see graphically how the timings on the first access to memory lines are set. Figure 2 shows an example of basic blocks and memory lines (L_x) covering all the prefetch cases that may appear. Let us assume that none of these memory lines is cached, since otherwise the fetch cost would

[2]We consider that a cache hit has the same cost than an LB hit. Technologically, this is the usual situation and avoids detailing where the previous memory line was, which would require to replicate this constraint.

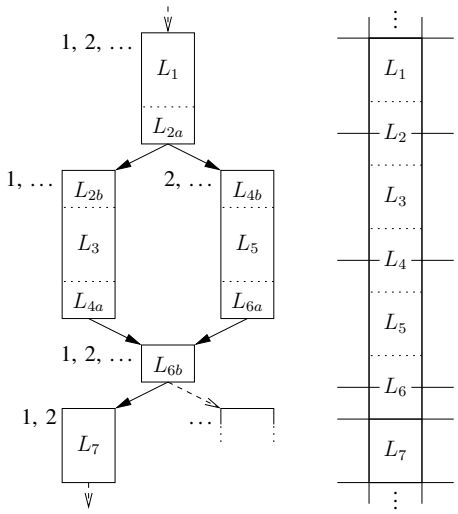


Figure 2. Example of prefetch cases.

Table I
FETCH COST OF THE FIRST ACCESS TO MEMORY LINES IN FIGURE 2 FOR SYSTEMS WITHOUT AND WITH PREFETCH.

Line	No Prefetch			Prefetch		
	Path-expl.		Cm	Path-expl.		Cm
	1	2		1	2	
L_1	LB	LB	LB	LB	LB	LB
L_{2a}	LB	LB	LB	PB$-L_1$	PB$-L_1$	PB$-L_1$
L_{2b}	0	-	0	0	-	0
L_3	LB	-	LB	PB$-L_2$	-	PB$-L_2$
L_{4a}	LB	-	LB	PB$-L_3$	-	PB$-L_3$
L_{4b}	-	LB	LB	-	LB	LB
L_5	-	LB	LB	-	PB$-L_{4b}$	PB$-L_{4b}$
L_{6a}	-	LB	0	-	PB$-L_5$	PB$-L_5-$LB
L_{6b}	LB	0	LB	LB	0	LB
L_7	LB	LB	LB	PB$-L_{6b}$	PB$-L_6$	PB$-L_{6b}$

be always a hit cost. If there is LB and no prefetch, the fetch cost of the first access to the line (i.e. discarding the execution costs and the hit costs once the line is already in LB) is the LB cost ($tmiss_{LB}$) for every accessed line. If there is LB and PB, the fetch cost of the first access to the line may be the LB cost (after a jump) or the maximum PB fetch cost ($tmiss_{PB}$) minus the execution of the previous memory line ($IChitCost_{i,j,k-1}$). Table I summarizes these costs for paths 1 and 2 of Figure 2. Columns are grouped into *No prefetch* and *Prefetch*. Using the path-explicit models for paths 1 and 2, there can be seen that divided memory lines (e.g. L_{2a}, L_{2b}) consecutively accessed have a single fetch cost. Also, having prefetch, the cost to be subtracted may be that of a full line or a divided one, depending on whether the whole line is executed previously or not. All this information can be obtained by statically analyzing the code. Fetch costs for the compact model (*Cm* columns) are discussed below.

A. Compact Model

As outlined above, the previous constraints refer to the path-explicit Lock-MS model. This model is relatively easy to understand and has an acceptable analysis time. Without prefetch this path-explicit model can be transformed into a compact model whose analysis is not dependent on the number of paths [1]. This can reduce very much the analysis time for benchmarks. In the columns labeled as *Cm*, Table I shows the fetch cost of the first access to each line using the compact model. As it can be seen, the transformation without prefetch is straightforward, with the only detail of setting the fetch cost of divided memory lines to the part in the common path (e.g. L_{2a} and L_{6b}).

However, this transformation is not directly applicable to a system with prefetch without allowing certain WCET overestimation. In general the compact model would maintain the prefetch accuracy when the prefetch is more effective, since many of the memory lines are accessed sequentially independently of the execution path. In the other cases, the WCET overestimation would be bounded by the number of accesses to a given memory line times the maximum overestimation on the memory line fetch cost. The last column in Table I shows the application of the compact model with this overestimation. In general, the fetch cost of most memory lines is also straightforward to set. The previous detail on the divided memory lines can be also extended to prefetch (L_{2a}, L_{2b}), but in some cases it may imply a more elaborated (yet straightforward) computation (e.g. L_{6a}, L_{6b}). The overestimation can be seen in the cost of line L_7. The prefetch cost in this line depends on the path previously taken and, since not all paths go through L_7, there is no way of reorganizing the associated costs to get a precise cost computation. In this case, the fetch cost of the first access to this line must be overestimated by assuming the worst possible case: PB$-L_{6b}$.

As an intermediate solution, an hybrid path-explicit/compact model is also possible [1]. This option has been proposed to deal with high level functional control flow information or execution-history dependent instruction timings, as in other ILP methods [8]. So, path-explicit constraints can be used inside a compact model to define specific execution behaviors. In a similar way, chunks of code where prefetch involves overestimations could be analyzed using path-explicit constraints. In the example in Figure 2 this would imply to use path-explicit constraints from a common point above the path fork (L_1) to the non-compactable memory line L_7, so that the cost of L_7 can be accurately set depending on the path. The partial cost computed in this way would be added to the cost of the rest of the task computed with the compact model.

V. RESULTS

We assume periodic tasks with fixed priorities managed by a *Rate Monotonic* scheduler. Table II shows the two sets of

Table II
TASK SETS "SMALL" AND "MEDIUM".

Set	Task	LB-only WCET	Period	Size
small	jfdctint	10108	23248	1072 B
	crc	109696	329088	536 B
	matmul	542229	2440031	208 B
	integral	716633	3583165	400 B
medium	minver	8522	19601	1360 B
	qurt	10117	30351	752 B
	jfdctint	10108	44475	1072 B
	fft	2886680	15010736	1016 B

tasks used in our experiments. Benchmarks include JPEG integer implementation of the forward DCT, CRC, matrix multiplication, integral computation by intervals, matrix inversion, computation of roots of quadratic equations and FFT. Sources have been compiled with GCC 2.95.2 -O2 and there has been no address tuning of the tasks, i.e. the code of each task begins at an address mapped to cache set 0. The WCET in this table refers to the LB-only system and it has been computed without context switches. Periods have been set so that the CPU utilization of each task in the LB-only system is 1.2. The "small" and "medium" task sets and the relation between the periods for each task have already been used in previous studies [1], [13].

Note that the WCETs and periods of each task set follow different patterns. In the small task set, WCETs and periods grow as the priority in tasks decreases. Their maximum growth can be seen in the period, being around one order of magnitude in most cases. However, the medium task set has relatively small WCETs and periods for all tasks but the one with the lowest priority, which is around three orders of magnitude larger. This means that the lowest priority task in the medium task set will be interrupted many times. So, in general, the medium task set will have more context switches than the small task set for a given time period.

The target architecture considered in our experiments is an ARM7 processor with instructions of 4 bytes. The LB/PB size (and memory/cache line size) is 16 bytes, or 4 instructions. The instruction caches are varied in size from 128 bytes to 4 KB and the eSRAM is kept fixed at 256 KB. In order to compute memory delays we have used Cacti v5.3, a cache circuit modeling tool [25], assuming a future high-performance embedded processor built in 32 nm technology and running at processor cycle equivalent to 36 FO4[3]. We have verified that all the tested caches, excluding the fully associative ones, meet the cycle time constraint. Besides, the access time of the 256 KB eSRAM is 7 cycles if we choose to implement it with low standby power transistors. Therefore, instruction fetch costs are 1 cycle on cache or LB instruction hit, 7 cycles on LB miss and a specific value between 1 and 7 cycles on PB hit, depending on prefetch timeliness. All data accesses are delivered directly from the eSRAM. The modeled execution costs are 1 cycle for non-executed predicated instructions[4], 2 cycles for non-memory instructions, and 1+7 cycles for loads and stores.

Next we characterize separately each single task by computing its WCET in four fetch systems, namely, Direct-eSRAM fetch (upper bound), LB-only fetch (upper bound), LB+PB fetch and Single-cycle fetch (lower bound). Then, we compare the whole multitasking system (LB + PB + iCache system) with systems without prefetch using Lock-MS (dynamic locking) [1] and Lock-MU (static locking) [13]. Next, to avoid the influence of the particular task code, we study the effects on the WCET of several forced prefetch penalties. Finally, we discuss the analysis cost.

A. Spatial Locality

Figure 3 shows, for each benchmark, the WCET of the LB-only, LB+PB and Single-cycle systems relative to a Direct-eSRAM system. A Single-cycle fetch system would correspond to a system with an ideal instruction cache, whereas a Direct-eSRAM fetch system would correspond to an easily predictable system with the instruction cache disabled. As it can be seen, WCETs are significantly lowered just by exploiting the spatial locality inside the current memory line. On average, an LB-only system reduces the Direct-eSRAM WCET by a $0.53\times$ factor. When exploiting the spatial locality both inside and outside the current line, LB+PB reduces the Direct-eSRAM WCET by a $0.40\times$ factor. Finally, a Single-cycle system reduces the Direct-eSRAM WCET by a $0.33\times$ factor on average. Obviously this Single-cycle system would need the whole program preloaded and fitting into cache, but it gives an idea of how far the ideal case is. So, the WCET reduction achieved by enhancing an LB+PB system with an instruction cache (temporal locality) could be up to an additional 7% at most (from 0.40 to 0.33, assuming that the ideal 0.33 is reachable). Results for individual tasks depart somewhat from the average case, but the trends are quite similar.

Utilization (Figure 4) can be used to test the spatial locality effects on multitasking systems. It is computed as the fraction of time the CPU is busy executing the task set in the worst case:

$$U = \sum_{i=1}^{NTasks} \frac{Wcost_i}{T_i}$$

Utilization values above 1 indicate the system is not executable (and not schedulable). On values below 1, the system may (or may not) be schedulable, depending on the response

[3]A fan-out-of-4 (FO4) represents the delay of an inverter driving four copies of itself. A processor cycle of 36 FO4 at 32 nm technology would result in a clock frequency around 2.4 GHz, which is in line with the market trends [26].

[4]Predicated instructions are those general (not jump) instructions that include several test bits, so that the instruction is executed as long as the test is true.

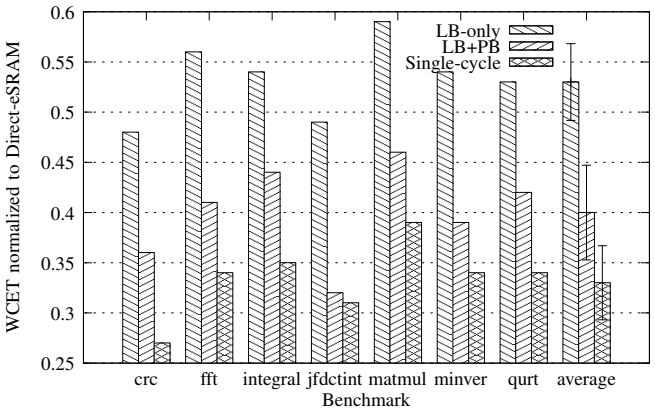


Figure 3. WCET values without cache.

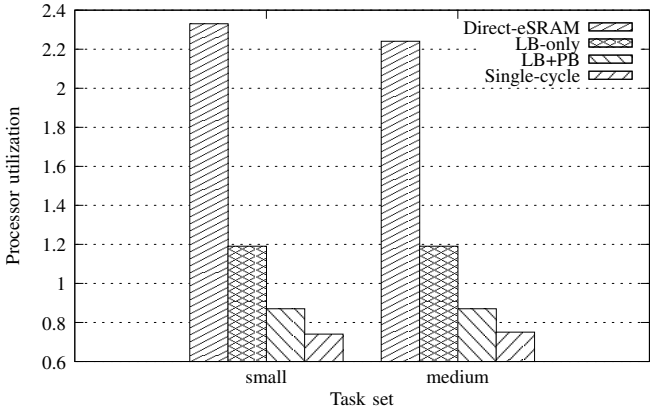
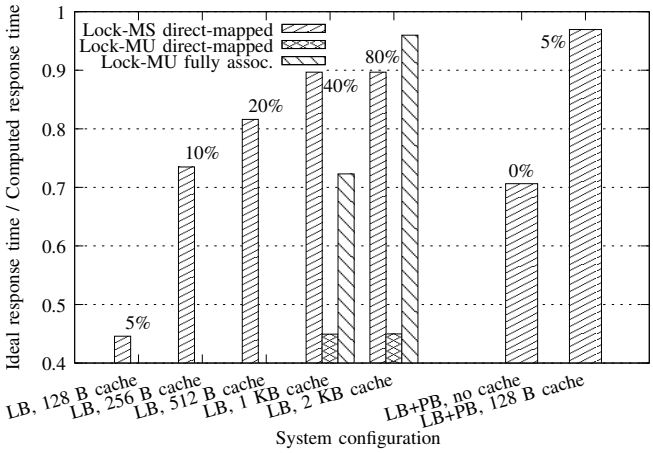
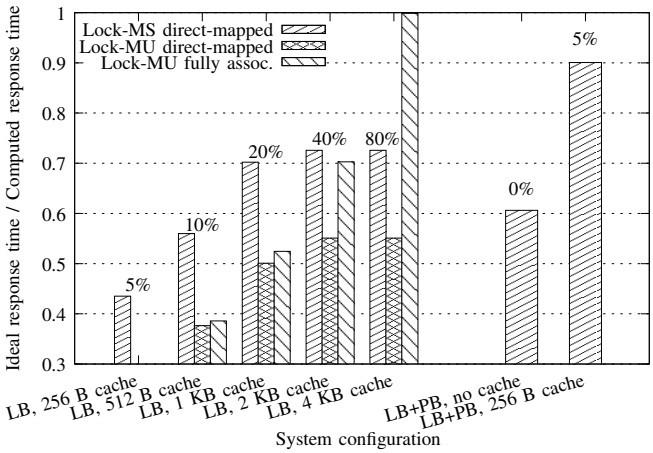


Figure 4. System utilization.



(a) Small task set



(b) Medium task set

Figure 5. Response times normalized in respect of the ideal case (Single-cycle fetch).

times and deadlines ($R_i \leq D_i, \forall i$). We use the periods as the deadlines. As indicated above, the periods have been set for both task sets in order to get an utilization value of 1.2 in the LB-only system. Using line buffer and hardware prefetch (without cache), the system utilization is below 0.9. So, the LB+PB captures spatial locality pretty well, almost halving the utilization of a system without LB. In this case, the performance gain in these task sets by an instruction cache can be no more than 6%.

These results show that instruction prefetch is very effective in worst-case performance, improving both single task and multitasking systems.

B. Dynamic Locking with Hardware Prefetch

As outlined above, hardware prefetch can increase worst-case performance significantly. With the timings considered, the average cycles per instruction (CPI) is one fetch hit cycle plus two execution cycles. This means that it is required to consume at least three instructions (of the four that may be stored in each memory line) to pay no prefetch penalties (3 cycles $\times$ 3 instr. $>$ 7 memory fetch cycles). The possibility of prefetching with no penalties is common in systems including prefetch (e.g. PIC32MX3XX/4XX flash microcontrollers).

Figure 5 shows the results comparing several configurations of cache plus LB using dynamic (Lock-MS) and static locking (Lock-MU) with those also using prefetch. For the experiments with instruction cache and prefetch we use dynamic locking (Lock-MS), since in general they will require few lines to be cached and locked, and it is well known that dynamic locking performs better than static on small caches. Instruction cache sizes range from 5% to 80% of the task set size (percentages shown over the bars). Using dynamic locking their configuration is always direct mapped since other options present very similar results. For static locking, both direct mapped and full associative configurations are presented, since they may differ very much. Results without prefetch are similar to those in previous studies [1], [14], [15], [16]. One detail to consider is that the medium task set presents much more context switches than the small task set, which benefits static locking.

When introducing the prefetch buffer, the LB+PB combination is very effective, reducing drastically the cache requirements, both for the small and the medium task sets. This can be easily explained because, in the general case, the only memory lines to be cached are those at jump destinations, which would never be correctly prefetched. Obviously there are more details to consider, such as memory lines with few instructions, but the ILP solver handles them all automatically. In the small task set, prefetch with 5% of cached contents outperforms both the static and dynamic locking approaches without prefetch, even having 16 times more cache size (80% of the task set size). In the medium task set the combination with prefetch is also the best one, except for the fully associative 4 KB static locking instruction cache. However, note that such an instruction cache is not realistic since it would imply hit times much higher than 1 cycle because the look-up procedure would require several cycles. Also, remember that the medium task set has a high number of context switches, which penalizes dynamic locking policies but not static ones. So, under certain conditions, it is possible that a large cache having a significant associativity degree using static locking may outperform a combination of prefetch with a very small instruction cache using dynamic locking. Nevertheless, a small instruction cache with prefetch seems the most adequate option.

These results show that the combination of line buffer with prefetch and a (minimal) dynamic locking cache outperforms systems with line buffer and larger caches without prefetch.

C. Effects of prefetch penalties

For the prefetch to be effective, the fetch time of the prefetched memory line must be hidden by the execution time of the previous line. So, the effectiveness will depend on the number of instructions in the previous line and how much they take to be executed. This depends on the hardware timings and on the particular code to analyze. To evaluate the effects of the prefetch penalty (fetch cycles not hidden using prefetch) we have forced several prefetch penalties. The forced prefetch penalties are 4 cycles (PB4) and 0 cycles (PB0). A 0-cycle penalty means that the prefetch works in an ideal way, i.e. on prefetch hits it fetches memory lines before they are needed. A 4-cycle prefetch penalty means that every fetch (by prefetch hit) of a new memory line costs 4 cycles. The worst prefetch case would be a prefetch taking the same time than not prefetching, i.e. 7 cycles, which is equivalent to not having PB.

Additionally, since these results will not be comparable to the previous ones, we perform this analysis on a collection of large synthetic tasks. These tasks range between 16 KB and 96 KB of code size, with 512 to 10^{65} possible paths through consecutive conditional statements.

Figure 6 shows the relative worst-case performance of several memory architectures without cache. There can be seen that the average prefetch penalty is decisive in the performance that the final system can provide. Whereas a system able to prefetch without penalties (LB+PB0) performs really close to the ideal Single-cycle case, the performance of a system with intermediate penalties (LB+PB4) is at a distance 0.13 of that of a system without prefetch (LB-only).

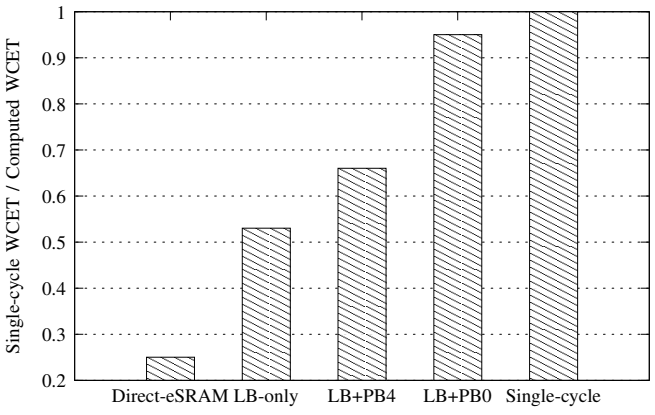


Figure 6. WCET without cache.

Adding a dynamic-locking direct-mapped instruction cache, some prefetch misses can be avoided by caching the memory lines after jumps. This can be seen in Figure 7. The smaller the task size, the more it can be cached, so the WCET is nearer to the ideal one. In the same way, the larger the cache size, the better the results. However, the performance gain because of the cache size depends on the available margin. That is, the performance gain when adding an instruction cache to the LB+PB0 system is much lower than that of adding it to a system with just an LB.

So, a hardware design avoiding prefetch penalties is a key factor to reach the best worst-case performance.

D. Analysis Cost

The analysis of the initial non-synthetic benchmarks takes a few milliseconds, so the discussion of their analysis is not relevant. In this section we test the analysis time of Lock-MS with compact models using the previous synthetic tasks. For these experiments we have used *lp_solve* version 5.5.0.13 using the default branch-and-bound rule (lowest indexed non-integer column) on a 2.33 GHz Intel Core 2 (32-bit).

First of all, we have observed that the solver gets the optimal (or a very close) solution of the minimization function $Wcost$ (eq. 2) in a very short time, and then spends the rest of the time testing similar solutions. Thus, using the first integer solution can save us much time and its accuracy can be easily tested by the real (not integer) solution. Although the real solution is not necessarily a valid ILP solution, it is faster (the solution space is continuous) and there cannot exist any better solution to the problem, i.e. it is a lower bound. Thus, the difference between any

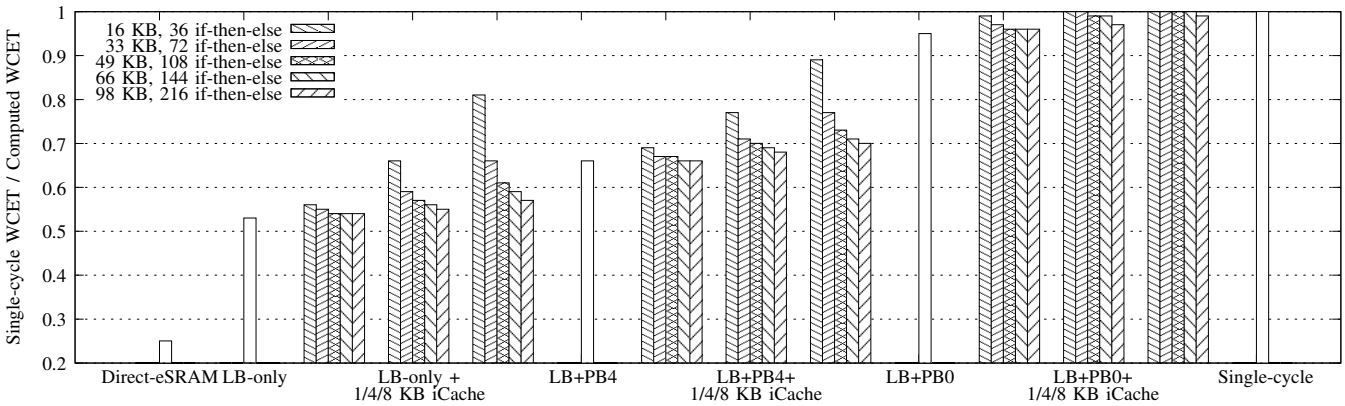


Figure 7. WCET of synthetic tasks combining instruction cache, line buffer and prefetch with forced prefetch penalties.

valid integer solution and the real one gives an idea of the goodness of the integer solution. Considering all experiments in this section, our results show that in 28% of cases the difference is 0, i.e. the real solution is also integer and both solutions coincide. Also, we have found no differences above 0.45%. This means that the obtained result is exact in 28% of cases, and in the other 72% the potential overestimation (probably not reachable because of the real solution being a lower bound) when using the first integer solution differs at most in five CPU cycles per thousand.

Figure 8 depicts the analysis time (both for the real and first integer solutions) as the complexity of the analysis grows. The x axis shows the program size, but both the cache size and the number of possible paths grow along this axis. Each code has been analyzed for three cache sizes, which can be seen in the plot as three marks for the same x value, both for real and integer solutions. The basic instruction cache has 64 sets and 4, 8 and 12 ways with sizes of 4, 8 and 12 KB respectively, and have a multiplicative factor (on the associativity degree and size) along the x axis (see labels in Figure 8). The resulting values have been fitted to a potential function, also shown in the figure. It can be seen that the analysis time grows approximately in a quadratic way when the complexity (code size, cache size and number of paths simultaneously) grows. The analysis of such large benchmarks (96 KB, 10^{65} paths, 72 way set-associative cache) takes no longer than 3 minutes.

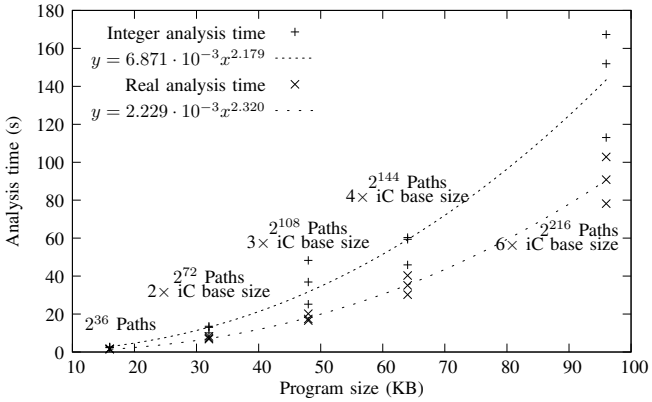


Figure 8. Analysis time as the complexity grows (iCache base sizes of 4, 8 and 12 KB).

VI. Conclusions

In this paper we propose a simple memory architecture for real time systems. It is composed of a lockable instruction cache, a line buffer and a prefetch buffer implementing the *next-line tagged* prefetch. This architecture is easy to implement and highly predictable.

To analyze this architecture we extend Lock-MS, an ILP-based method that obtains the selection of memory lines to be loaded into the dynamic locking instruction cache that minimizes the worst overall execution cost (WCET plus preloading times at context switches). The prefetch extension of the path-explicit Lock-MS model is possible by adding new constraints to compute the prefetch penalty of the memory lines. However, this path-explicit model cannot be transformed directly into a compact Lock-MS model without introducing certain overestimations. We show that these possible overestimations are relatively small and they can be avoided by an hybrid path-explicit/compact Lock-MS model.

Our results shows that hardware prefetch reduces very much the resulting WCET. A system with prefetch and without cache has a WCET comparable to a system without prefetch and a dynamic locking cache with size 10% of the task set code. Comparing to static locking, this cacheless system with prefetch performs better than a static locking direct-mapped cache with size 80% of the task set code, and very near to a full associative cache with size 40% of the task set code. Adding just a dynamic locking cache of size 5% of the task set code to the system with prefetch, the resulting WCET outperforms all other realistic systems. We also evaluate the effects of the prefetch penalty on

the resulting WCET showing that, without cache, a system without prefetch penalties has a performance 95% of the ideal case. These results state the importance of a good prefetch design and its benefits when used in combination with a small dynamic locking cache. Finally, the analysis cost of the method is very low, being able to analyze large benchmarks (96 KB, 10^{65} paths, 72 way set-associative cache) in less than 3 minutes.

REFERENCES

[1] L. C. Aparicio, J. Segarra, V. Viñals, C.Rodríguez, and J. L. Villarroel, "Improving the use of instruction cache-locking methods in multitasking real-time systems," Universidad de Zaragoza, Tech. Rep. RR-09-01, Mar. 2009.

[2] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok, "Real time scheduling theory: A historical perspective," *Real-Time Systems*, vol. 28, pp. 101–155, 2004.

[3] J. V. Busquets and A. Wellings, "Adding instruction cache effect to schedulability analysis of preemptive real-time systems," in *Proceedings of RTAS96*, 1996.

[4] C. Lee, K. Lee, J. Hahn, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim, "Bounding cache-related preemption delay for real.time systems," *IEEE Transactions on Software Engineering*, vol. 27, no. 9, Sep. 2001.

[5] J. Staschulat, S. Schliecker, and R. Ernst, "Scheduling analysis of real-time systems with precise modeling of cache related preemption delay," in *Proceedings of the Euromicro Conference on Real-Time Systems*, Jul. 2005, pp. 41–48.

[6] R. Wilhelm *et al.*, "The determination of worst-case execution times-overview of the methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, 2007.

[7] L. C. Aparicio, J. Segarra, C. Rodríguez, J. L. Villarroel, and V. Viñals, "Avoiding the WCET overestimation on LRU instruction cache," in *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug. 2008.

[8] Y. T. S. Li, S. Malik, and A. Wolfe, "Cache modeling for real-time software: beyond direct mapped instruction caches," in *IEEE Real-Time Systems Symposium*, Dec. 1996, pp. 254–264.

[9] T. Lundqvist and P. Stenström, "An integrated path and timing analysis method based on cycle-level symbolic execution," *Real-Time Systems*, vol. 17, no. 2-3, pp. 183–207, November 1999.

[10] F. Mueller, "Timing analysis for instruction caches," *Real-Time Systems*, vol. 18, no. 2-3, pp. 217–247, May 2000.

[11] H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and precise WCET prediction by separated cache and path analyses," *Real-Time Systems*, vol. 18, no. 2-3, pp. 157–179, May 2000.

[12] A. Martí Campoy, Á. Perles Ivars, and J. V. Busquets Mataix, "Static use of locking caches in multitask preemptive real-time systems," in *IEEE Real-Time Embedded System Workshop*, December 2001.

[13] I. Puaut and D. Decotigny, "Low-complexity algorithms for static cache locking in multitasking hard real-time systems," in *IEEE Real-Time Systems Symposium*, December 2002.

[14] A. Martí Campoy, Á. Perles Ivars, F. Rodríguez, and J. V. Busquets Mataix, "Static use of locking caches vs. dynamic use of locking caches for real-time systems," in *Canadian Conference on Electrical and Computer Engineering*, May 2003.

[15] A. Martí Campoy, S. Sáez, Á. Perles Ivars, and J. V. Busquets Mataix, "Performance comparison of locking caches under static and dynamic schedulers," in *Proceedings of the 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming*, Lagow, 2003.

[16] I. Puaut, "WCET-centric software-controlled instruction caches for hard real-time systems," in *Euromicro Conference on Real-Time Systems*. Los Alamitos, CA, USA: IEEE Computer Society, 2006, pp. 217–226.

[17] X. Vera, B. Lisper, and J. Xue, "Data cache locking for tight timing calculations," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 1, pp. 1–38, December 2007.

[18] C. Luk and T. C. Mowry, "Cooperative prefetching: compiler and hardware support for effective instruction prefetching in modern processors," in *Proceedings of the ACM/IEEE International symposium on Microarchitecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998, pp. 182–194.

[19] G. Reinman, B. Calder, and T. M. Austin, "Fetch directed instruction prefetching," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, 1999, pp. 16–27.

[20] W.-C. Hsu and J. E. Smith, "A performance study of instruction cache prefetching methods," *IEEE Transactions on Computers*, vol. 47, no. 5, pp. 497–508, 1998.

[21] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, 1978.

[22] G.-H. Park, O.-Y. Kwon, T.-D. Han, S.-D. Kim, and S.-B. Yang, "An improved lookahead instruction prefetching," in *HPC-ASIA '97: Proceedings of the High-Performance Computing on the Information Superhighway, HPC-Asia '97*. Washington, DC, USA: IEEE Computer Society, 1997, p. 712.

[23] V. Chvátal, *Linear Programming*. W.H. Freeman & Company, 1983.

[24] F. Rossi, P. V. Beek, and T. Walsh, *Handbook Of Constraint Programming*, F. Rossi, P. V. Beek, and T. Walsh, Eds. Elsevier, 2006.

[25] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.3," http://www.hpl.hp.com/research/cacti/.

[26] "Chart watch: High-performance embedded processor cores," *Microprocessor Report*, vol. 22, pp. 26–27, Mar. 2008.