# Optimizing a combined WCET-WCEC problem in instruction fetching for real-time systems

R. Gran[†‡], J. Segarra[†‡], C. Rodriguez[§‡], L. C. Aparicio[†‡] and V.Viñals[†‡]

[†] Dpt. Informática e Ingeniería de Sistemas, Universidad de Zaragoza,
C/María de Luna, 1, 50018, Zaragoza, Spain, Tlf (+34)976760001 Fax.(+34)976761914

[†] I3A Instituto de Investigación en Ingeniería de Aragón (I3A), 50018, Universidad de Zaragoza

[§] Facultad de Informática, Universidad del País Vasco,
Paseo Manuel de Lardizábal, 1, 20018 San Sebastián, Spain, Tlf.(+34)943015093 Fax.(+34)943015590

[‡] HiPEAC NoE (High-Performance Embedded Architectures and Compilers)

E-mail: {rgran, jsegarra, luisapa, victor}@unizar.es, clemente.rodriguez@ehu.es

February 6, 2013

**Abstract**

In real-time systems, time is usually so critical that other parameters such as energy consumption are often not even considered. However, optimizing the worst energy consumption case can be a key factor in systems with severe power-supply limitations. In this paper we study several memory architectures using combined time and energy optimization models for real-time multitasking systems. Each task is modeled using Lock-MS, a method to optimize the WCET of a task, with an added set of constraints to model in the same way the WCEC (worst case energy consumption). Our tested hardware components focus on instruction fetching, including a lockable cache, a line buffer and a sequential prefetch buffer. We test a variety of instruction fetch alternatives optimizing time and energy consumption. Our results show that the accuracy of the estimation of the number of context switches in the worst case may affect very much the resulting WCEC (up to 8 times in our experiments) and that optimizing the WCEC may provide similar execution times than optimizing the WCET, with up to 5 times less energy consumption. Additionally optimization functions combining WCET and WCEC with different weights show very interesting WCET-WCEC trade-offs. This confirms that methodologies testing such optimizations at design time could be very helpful to provide a precise system set-up.

*Keywords*: hard real-time, embedded, WCET, energy-aware, instruction memory-hierarchy

## 1 Introduction

Design of embedded and real-time systems and processors for specific tasks is a complex work. Besides, for autonomous real-time systems operated with batteries or draining energy from limited sources, low energy consumption is critical. Such systems are becoming increasingly important, ranging from sensor networks, surveillance systems and satellite subsystems to search and rescue robotics. However, their energy consumption is not usually tightly bounded, i.e. their specifications do not include a worst case energy consumption (WCEC).

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 calculated 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 data-path pipelining belong to the second one. Analyzing and optimizing

the WCET in the presence of such speculative components is very hard, but it is the base of all further scheduling analysis.

Memory hierarchy is one of these challenging structures. In general, an on-chip memory hierarchy made up of one or more cache levels takes advantage of program locality and saves execution time and energy consumption. This is achieved by delivering data and instructions with an average latency of a few processor cycles and an energy consumption far below that of the external main memory. Unfortunately, the cache behavior depends on past references and its analysis is very difficult due to this lack of predictability [11, 15, 24]. This implies that, usually, tasks are analyzed in isolation, i.e. without considering inter-tasks interferences. Instead, alternative on-chip memory architectures are commonplace in real-time studies because they are easier to analyze. Such designs include lockable caches and scratchpad memories.

Cache locking disables the cache replacement, which fixes cache contents. This means that, for an instruction cache, hits and misses depend on whether each instruction belongs to a cached and locked memory line, and not on the previous accesses. There are several WCET analysis/optimization methods based on lockable instruction caches. These methods consider a restricted cache behavior disabling cache replacement, a possibility offered by many commercial processors.[1] With specific contents locked in cache, the timing calculations are easier, so these methods enable the full system to be analyzed, 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 is never replaced [12, 18]. Martí Campoy et al. use a genetic algorithm to obtain the selection [12], whereas Puaut and Decotigny propose two low-complexity selection algorithms: one to minimize the utilization and another to minimize the interferences [18]. None of these three algorithms studies the possibility of worst path changes depending on the selected cache lines. Instead, the worst path is determined only once, assuming an empty cache. Therefore, as stated in [8], these single-path analysis are not optimal. Dynamic cache locking, on the other hand, allows tasks to disable and enable the cache replacement. Although there are studies which allow instruction cache reloading at any point in order to manage instructions and data phases in programs [17, 25], such detail is usually not needed and most studies restrict reloading to context switches [3, 4, 13, 14]. These approaches require per-task selection of contents, with the drawback that preloading is performed every time a task starts/continues its execution. Despite this drawback, studies show that dynamic locking approaches provide a better worst case performance than static locking policies, except when almost all code fits in cache [4, 13]. Lockable caches can also be used on data [25]. However its complex analysis leads to a limited performance compared to instruction cache locking.

Alternatively, scratchpad memories are considered for servicing specific main-memory areas [26]. Scratchpad memories are small, fast and low-consumption on-chip SRAMs. The allocated memory chunks can be configured and modified at run-time. Indeed, a scratchpad memory can be seen as a locked direct-mapped cache, although scratchpad management usually focus on tuning the data location at compile time (coarse-grain data mapping), whereas cache analysis focus on accounting the run-time consequences of the existing mapping between data and cache lines [19].

Regarding energy consumption in real-time systems, there are works that make use of dynamic voltage scaling (DVS) for energy-efficient scheduling of tasks [6]. Such studies are essentially theoretical, and they study how to manage the processor speed (and thus its estimated energy consumption) when there is enough slack, i.e. based on the WCET of tasks. However, they usually do not include in such scheduling analysis the worst-case energy consumption (WCEC) of the system, since this value is generally unknown.

The combined WCET-WCEC problem has been partially addressed in a previous work [9]. It nicely describes the problem and shows that the WCEC cannot be calculated as $average\ power \times WCET$, since the path corresponding to the WCET may not coincide with the path with the highest energy consumption. So, a technique to estimate the WCEC of a processor executing a single task is introduced. The WCEC is computed by Integer Linear Programming (ILP) similarly to the WCET analysis techniques [11]. That is, modeling the energy consumption cost of basic blocks through linear constraints and obtaining the worst

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

case by finding the maximum. Although in our work we are also addressing the combined WCET-WCEC, three characteristics of our proposal are key to understand the substantial improvement of our work with respect to this previous proposal. First, our proposed instruction memory organization includes a lockable cache, whereas they use a conventional cache. Conventional caches are not designed for real-time systems, so they may not provide a good worst-case performance, and their behavior is difficult to analyze compared to lockable caches. Hence, they analyze a single-task system, whereas we test the worst case of several tasks running in a preemptive real-time scheduler. Second, this multitasking analysis allows us to take into account the interferences of context switches in comparison to their proposal. As we will show in the results section, this aspect is very important in multitask systems. Third, they simply calculate the WCET-WCEC, whereas our ILP formulation optimizes it. That is, our proposal is able to select the set of instructions to lock into the instruction cache so that the worst case of the system is minimized. Finally, the optimization of the combined WCET-WCEC problem is a significant milestone, since previous worst-case optimization studies focused on WCET only and they do not consider other factors such as energy consumption [3, 4]. This can be counterproductive when considering the WCEC, since the WCET optimization process would discard options having a few more execution cycles even if such options consume far less energy.

In this paper we study different hardware components and optimization approaches for solving a combined WCET-WCEC problem, focusing on a cache-based instruction memory hierarchy. That is, both latency and energy consumption are modeled as constraints in a single ILP problem. In order to study different instruction fetch components, our target system includes a lockable instruction cache that may be enhanced with instruction line buffering and sequential prefetching. We consider the *Lock-MS* method for WCET optimization [4] and extend it to compute and optimize the WCEC within a real-time multitasking system, taking into consideration the effects of context switches. As we will see, depending on the particular tasks and hardware components, not considering the WCEC analysis may result in extremely energy-wasting real-time systems. For instance, in one of our experiments with the benchmark *CRC*, when optimizing time it shows the same worst case execution time as when optimizing energy, but it uses up to 5 times more energy. Also, the accuracy of the estimated number of context switches affects very much the energy consumption (up to 8 times more energy consumption in our experiments), which, to the best of our knowledge, no previous work has pointed before. Finally, in many cases the best results are not achieved by optimizing the WCET of the WCEC, but by an optimization goal combining both parameters. Such combined analysis and optimization is possible through our proposed methodology. This allows a very precise system set-up in a design phase, which could result in more efficient products.

This paper is organized as follows. Section 2 presents our tested hardware architecture. The combined WCET-WCEC optimization model is described in Section 3. Sections 4 and 5 describe the experimentation environment and the obtained results. Finally, Section 6 presents our conclusions.

## 2 Hardware architecture

In this paper, we consider a simple processor model like the ones of family ARM7T (although the considered ISA in our experiments is ARMv7), a multicycle non-pipelined 1-issue processor. This allows us to focus on the instruction memory hierarchy for the computation of the WCET and WCEC.

Previous studies on WCET analysis have used or proposed memory hierarchies intended to be easily analyzable and to have a good worst case performance [4, 18, 26]. In this paper, additionally, our design aims at low energy consumption: low-power standby transistors, tag-only array reading in prefetch lookup, two separate SRAM banks instead of a dual-ported one, etc.

Figure 1 shows the chosen on-chip Harvard organization: the embedded SRAM main memory is split into two independent banks of 128 KB (iSRAM and dSRAM, for instructions and data, respectively). Having separate ports avoids contention, simplifies the timing analysis, and involves less energy consumption per access than in a dual-ported SRAM at the cost of some area. Load/store instructions use the dSRAM read/write port to transfer 32-bit words to/from the register file. Instruction fetch misses and prefetches read a 4-instruction cache line (128 bits) from iSRAM. Below the iSRAM, our architecture has a lockable instruction cache (IC), a line buffer (LB) and a prefetch buffer (PB). Between context switches the IC works

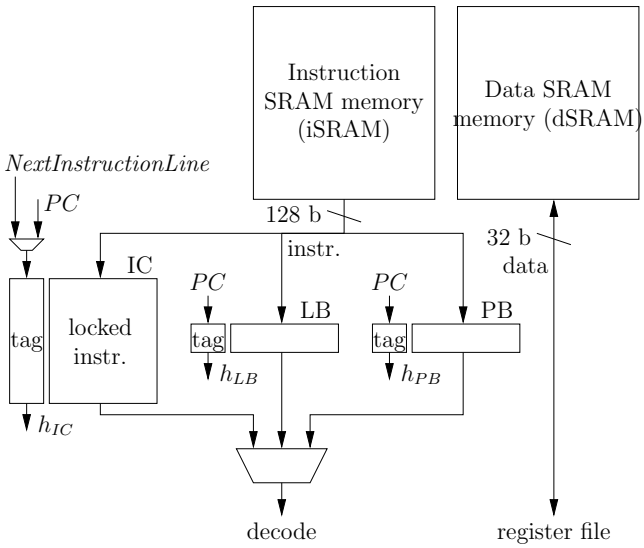



Figure 1: Memory hierarchy organization. PC stands for program counter; IC for instruction cache; LB for line buffer; PB for prefetch buffer; hIC, hLB and hPB for instruction hit in IC, LB or PB.

in a locked state, so no replacement is allowed. On each context switch, the IC is loaded with the instruction lines previously selected for the task being switched in. Even without replacement, IC may exploit significant amounts of temporal locality, because the instruction lines stored in IC are in general heavily reused. In order to exploit the spatial locality of instruction streams not cached in IC, a fetch miss brings to LB a whole cache line. Thus, short-range sequential fetching after a miss does not require iSRAM access. In order to better exploit sequential fetching, a prefetch mechanism is available [3].

The instruction fetch proceeds as follows. In a single cycle (fetch stage), IC, LB, and PB are concurrently accessed, providing the corresponding instruction in case of hit. On miss, the required instruction line is requested to iSRAM, which will fill LB after a few cycles (iSRAM access stages). If the instruction being fetched belongs to a memory line different from the previous PC, the prefetch mechanism is started. If so, it must be checked whether the instruction line to be prefetched is already in IC or not. This look up avoids redundant memory requests, which can be an important source of energy wasting. Since the tag array in IC is single ported, this action cannot be done during the fetch stage, so prefetch triggering is checked the first cycle just after the fetch stage (i.e., in parallel to the decode stage in case of fetch hit, and in parallel to the iSRAM access stages in case of fetch miss). If prefetch is triggered (i.e., if the next memory line is not cached), the next memory line will be requested to iSRAM. This will occur in the next cycle if the iSRAM is free, or after completing any ongoing instruction memory request. When the prefetched instruction line arrives, it is written in PB. If the next memory line required is the one speculatively prefetched, the roles of LB and PB are interchanged, so that the former LB will be recipient of the next prefetched instruction line. Note that the iSRAM may be busy reading a prefetched line when a fetch miss arises. If this occurs, the miss processing is delayed until the line gets read, and if this happens quite often performance may suffer.

Summarizing, the described memory system has three key advantages: it is simple, its prefetch system is suitable for a low-power locking cache and it is amenable for use in an ILP-based WCET analysis [3]. We assume that our system has no additional sources of latency variability (data cache, branch predictor, out-of-order execution, etc.). Many embedded processors can operate under these considerations, which are also assumed in previous studies [12, 18].

# 3 Combined WCET-WCEC optimization model

Using the memory organization described above, the instructions to be locked into the instruction cache must be selected so that a low WCET-WCEC is obtained. The *Lock-MS* method selects the set of instructions to be locked, so that the schedulability of the whole system is maximized [4]. In order to select such instructions, 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. Additionally to the instruction cache, it supports the instruction line buffer and the instruction prefetch buffer described above [3]. This method is based on *Integer Linear Programming* (ILP) [7, 20]. Specifically, all requirements must be modeled as a set of linear constraints and then the WCET of the resulting system is minimized. This allows not only to obtain a WCET bound, but the lowest possible one given the assumed memory organization. Next, we outline how to set the ILP constraints (see [3, 4] for further detail).

The WCET of a task $i$ must be equal or higher than the worst execution time of any of its paths $j$.

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

The execution time of each path $j$ can be calculated as the sum of the execution times of all the particular memory lines $k$ (subset of instructions, in a basic block, which fit in a cache line) it traverses.

$$pathTime_{i,j} = \sum_{k=1}^{NLines_{i,j}} lineTime_{i,j,k}$$

In turn, the worst execution time of each particular memory line depends on its hits and misses.

$$lineTime_{i,j,k} = hitT_{i,j,k} \cdot nhit_{i,j,k} + missT_{i,j,k} \cdot nmiss_{i,j,k}$$

The time cost of any hit or miss ($hitT$, $missT$) is a constant that depends on the timing of the hardware structures. Such constants consider the different instructions in the memory line $k$ and their execution times [3, 4]. The number of hits and misses of a memory line ($nhit$, $nmiss$) depends on the control flow graph and the memory lines locked in cache. For instance, if a memory line is traversed $n$ times and it has been cached and locked, it will hit $n$ times and miss 0 times.

The final worst execution time to be used in multitasking systems also accounts for the overhead of context switches.

$$wcetM_i = wcet_i + ncswitch_i \cdot switchTime_i \tag{2}$$

It requires an (external) estimation of the number of context switches in the worst case ($ncswitch_i$) depending on the selected scheduling policy. The estimation of such value is discussed in Section 4.1.

Thus, the ILP system is set to minimize eq. 2, and in this process it provides the selection of memory lines to be locked in the instruction cache in order to reach such result.

We apply the same methodology in order to calculate the WCEC. That is, additionally to model the timing requirements to obtain the worst time, we add a new set of constraints to model the worst energy consumption. The key point is that energy constraints can be constructed in the same way as timing constraints. That is, instead of execution time costs ($wcet$, $pathTime$, $lineTime$, $hitT$, $missT$, $wcetM$, $switchTime$, etc.) the new constraints will represent energy costs using the same control flow constraints and parameters ($nhit_{i,j,k}$, $nmiss_{i,j,k}$, $ncswitch_i$, etc.). Hence, the only changes between time and energy constraints are the specific coefficients of each operation, i.e., time coefficients (cycles) for the timing constraints and energy consumption coefficients (picojoules) for the energy constraints.

Therefore, for each task, both the worst time (WCET plus time overhead of context switches) and worst energy (WCEC plus energy overhead of context switches) are modeled together. Using this model, any linear combination of worst time and worst energy could be used as the optimization goal instead of eq. 2:

$$OptGoal = \alpha \cdot wcetM_i + (1 - \alpha) \cdot wcecM_i \tag{3}$$

This optimizes both parameters with the desired $\alpha \in [0, 1]$ and selects the memory lines to be locked into cache. Also, we normalize both $wcetM_i$ and $wcecM_i$ previously (not shown), in order to use straightforward values of $\alpha$.

Table 1: Worst-case execution time and energy consumption metrics

| Time | Energy | Description |
|---|---|---|
| WCET (Worst-Case Execution Time) | WCEC (Worst-Case Energy Consumption) | Time/energy required, at most, for completing the execution of a single task (without being interrupted) on a specific hardware platform |
| WCET-M (WCET-Multitask) | WCEC-M (WCEC-Multitask) | WCET/WCEC plus worst-case preemption delay overheads due to considering a bound on the number of context switches in the worst case |
| WCRT (Worst-Case Response Time) | WCRE (Worst-Case Response Energy) | WCET-M/WCEC-M of the task plus the WCET-M/WCEC-M of each interrupting task times its number of interruptions |
| WCSRT (Worst-Case System Response Time) | WCSRE (Worst-Case System Response Energy) | The highest WCRT/WCRE among the tasks in the system |

# 4 Experimentation environment

Experiments in this paper comprise worst-case time and energy values for multitasking systems. Although temporal metrics are common and well-known in real-time systems, energy metrics are uncommon in this area. Table 1 describes the used metrics, both regarding time and energy, and their relations. WCET ($wcet_i$ in eq. 1) and WCEC are the basic worst-case single task metrics. In multitask systems, it is required to know the worst-case preemption delay too, especially the cache related preemption delay. With conventional caches, this value usually depends on the interrupting task and may be different for each specific preemption, which implies a complex analysis (e.g. [2]). In contrast, using lockable caches this value is proportional to the cache lines used, and remains constant through the whole system lifetime. Hence, we can easily group the WCET with the overheads due to context switches into a new WCET-M metrics ($wcetM_i$ in eq. 2). The same applies to WCEC-M.

All our experiments consist of modeling each task and system as linear constraints, optimizing the model and simulating the results in a *Rate Monotonic* scheduler. Considering a *fixed priority* scheduler, the feasibility of periodic tasks can be tested in a number of ways [22]. Worst-case response time (WCRT, or simply response time) analysis is one of these mathematical approaches. It is based on the following equation for independent tasks:

$$R_i^{n+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \tag{4}$$

where $R_i$ is the WCRT, $C_i$ in our case is the WCET-M 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 [5, 10, 23]. A task $i$ meets its real-time constraints if $R_i \leq T_i$. The worst-case response energy (WCRE) can be calculated in the same way. Finally, in order to test whole multitasking systems, we use the response time or response energy of the task with the lowest priority (WCSRT or WCSRE).

## 4.1 Estimation of the number of context switches

A priory, the exact number of task switches (preemptions plus starting a task execution when the processor is idle) in the system is not known. However, it can be overestimated in many ways [10, 23], and then, included in the ILP. So that, the solver assumes that there will be as many context switches as estimated. In our experiments, we use two different estimations for the number of context switches. First, we use a trivial and pessimistic bound: the number of context switches for a task i, during $T_i$, cannot be higher than

Table 2: Task sets "small" and "medium"

| Set | Task | WCET (c) | WCEC (pJ) | Period (c) |
|---|---|---|---|---|
| small | jfdctint | 18704 | 93935.12 | 84168 |
| | crc | 203880 | 1004335.82 | 917460 |
| | matmul | 834226 | 4337954.64 | 3754017 |
| | integral | 1281414 | 6629432.60 | 5766363 |
| medium | minver | 14352 | 72799.72 | 57408 |
| | qurt | 10117 | 86466.51 | 68104 |
| | jfdctint | 18704 | 93935.12 | 74816 |
| | fft | 4737168 | 24074762.80 | 19422389 |

the sum of the maximum number of invocations of any higher priority task:

$$ncswitch_i \leq \sum_{j=1}^{i-1} \left\lceil \frac{T_i}{T_j} \right\rceil \tag{5}$$

By using this estimation in eq. 2, valid but pessimistic WCET-M and WCRT (eq. 4) will be obtained. In our second estimation method, we use these (pessimistic) response times and we refine the estimation of the maximum number of context switches as:

$$ncswitch_i \leq \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil \tag{6}$$

Using this new estimation, we solve again the ILP problem, which results in a more accurate system (which may have a different selection of instruction lines to lock into cache). This process could be repeated until the number of context switches converges but, as we will see, few iterations provide enough accuracy.

## 4.2 Benchmarks

We assume periodic tasks with fixed priorities managed by a *Rate Monotonic* scheduler. Table 2 lists the two sets of 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, from the *SNU-RT Benchmark Suite for Worst Case Timing Analysis* [21]. Sources have been compiled with GCC 2.95.2 -O2 without relocating the text segment, i.e., the starting address of the code of each task map to cache set 0. In this table, both WCET (in CPU cycles) and WCEC (in pJ) refer to the task being executed in a system without cache, LB or prefetch, and they have been computed without context switches. Task periods have been set so that all configurations are schedulable, which allows us to better compare the obtained results. The "small" and "medium" task sets and the relation between the periods for each task have already been used in previous studies [4, 18].

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. Periods grow approximately by an 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 much more context switches than the small task set for a given interval.

## 4.3 Architectural details

In this section we detail the timing and energy consumptions for the memory organization shown in Figure 1. We assume a multi-cycle non-pipelined 1-issue processor with the ARMv7 instruction set architecture, which has 4 bytes of instruction size. The LB/PB size (and memory/cache line size) is 16 bytes, or 4 instructions.

Table 3: Energy consumptions for Read, Write and Lookup operations (measured in picojoules, pJ), static power (leak) and area cost.

| | 32 bit | | 128 bit | | | Leak | Area |
|---|---|---|---|---|---|---|---|
| | Rd | Wr | Rd | Wr | Lkup | (pJ/cycle) | ($\mu$m$^2$) |
| iSRAM | — | — | 147.92 | — | — | 0.05 | 1216188 |
| dSRAM | 38.50 | 38.37 | — | — | — | 0.05 | 1216188 |
| 1 KB IC | 1.07 | — | — | 1.44 | 0.09 | 0.07 | 7395 |
| 512 B IC | 0.67 | — | — | 1.27 | 0.06 | 0.04 | 5928 |
| 256 B IC | 0.46 | — | — | 1.16 | 0.04 | 0.02 | 4933 |
| 128 B IC | 0.36 | — | — | 1.11 | 0.03 | 0.011 | 1587 |
| 64 B IC | 0.31 | — | — | 1.08 | 0.02 | 0.007 | 584 |
| LB/PB | 0.08 | — | — | 0.27 | — | 0.002 | 36 |

The instruction cache size is varied from 64 bytes to 1 KB. Main memory separates instructions and data into two 128 KB embedded SRAM memories (iSRAM and dSRAM). In order to compute memory circuit delays we have used Cacti v6.5 [16], a memory modeling tool at microarchitectural level, assuming an embedded processor built in 32 nm technology and running at a processor cycle equivalent to 36 FO4[2]. All memory circuits have been modelled with low-standby power transistors in order to save as much energy as possible (further details in Appendix A). We have verified that all the tested caches meet the cycle time constraint. Further, the access time of each 128 KB embedded SRAM is 6 cycles if we choose to implement it with low-standby power transistors. Therefore, instruction fetch costs are 1 cycle on cache or LB/PB instruction hit, 7 cycles on LB miss (6 cycles of memory access plus 1 cycle of LB access) and a specific value between 1 and 6 cycles for the first hit to PB/LB being filled by prefetch, depending on prefetch timeliness. All data accesses are delivered directly from the dSRAM in 6 cycles. So, once an instruction is fetched, the modeled execution costs are 1 cycle for non-memory instructions, and 1+6 cycles for loads and stores (1 cycle for address computation plus access latency).

In Table 3, we show the values of energy consumption according to Cacti. For each memory component, columns (from left to right) show energy consumption for Read (Rd) and Write (Wr) operations for different data widths (32 bit and 128 bit). Next column shows energy consumption for look-up accesses (Lkup) in ICs (always direct-mapped), already included in the read/write consumptions. In the Leak column, we show energy per cycle due to leakage (0.4 ns at 2.4 GHz). Whether a memory component is performing an operation or not, this energy consumption must be accounted. Finally, in the last column, we show area costs. Some values of this table have been intentionally left blank because none of the evaluated memory hierarchy organizations requires that value. First two rows show the energy consumption of the two eSRAM modules: iSRAM for instructions and dSRAM for data. iSRAM is accessed on prefetch, fetch miss or context switch, and all these cases imply reading complete memory lines (128 bit). On the other hand, data load and store operations always access dSRAM for a word (32 bit). Next six rows show the energy consumption for different direct-mapped instruction cache sizes and LB/PB circuits, which are always written with line granularity and read with word granularity.

# 5 Results

We consider four memory configurations, namely *Direct-eSRAM*, *LB-only*, *No-prefetch*, and *Prefetch*. In *Direct-eSRAM*, instructions and data values are serviced from the corresponding embedded iSRAM or dSRAM. *LB-only* adds the instruction line buffer (LB) to the previous configuration. In *No-prefetch* we add a direct-mapped lockable IC ranging from 4 to 64 sets (64 B to 1 KB) to the instruction memory datapath. Finally, *Prefetch* adds sequential prefetching to the previous configuration, with prior IC lookup.

[2]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 [1].

Table 4: Energy consumptions of each memory operation

| Operation | Memory Configuration | | | |
|---|---|---|---|---|
| | *Direct-eSRAM* (iSRAM + dSRAM) | *LB-only* (iSRAM + dSRAM+LB) | *No-prefetch* (iSRAM+dSRAM +IC+LB) | *Prefetch* (iSRAM+dSRAM +IC+LB+PB) |
| Fetch Stage | | Rd-LB(32b) | Rd-IC(32b) + Rd-LB(32b) | Rd-IC(32b) + Rd-LB(32b) + Rd-PB(32b) |
| Prefetch Lookup | | | | Lkup-IC |
| Prefetch Access | | | | Rd-iSRAM(128b) + Wr-LB/PB(128b) |
| Fetch Miss | Rd-iSRAM(32b) | Rd-iSRAM(128b) + Wr-LB(128b) + Rd-LB(32b) | | |
| Data Load | Rd-dSRAM(32b) | | | |
| Data Store | Wr-dSRAM(32b) | | | |
| Context Switch (flushed buffers) | | Rd-iSRAM(128b) + Wr-LB(128b) | | Rd-iSRAM(128b) + Wr-LB(128b) + Rd-iSRAM(128b) + Wr-PB(128b) |
| Context Switch (IC) | | | refill-energy = mem-lines-in-IC × (Rd-iSRAM(128b) + Wr-IC(128b)) | |
| Any Cycle | Leak-iSRAM + Leak-dSRAM | Leak-iSRAM + Leak-dSRAM + Leak-LB | Leak-iSRAM + Leak-dSRAM + Leak-IC + Leak-LB | Leak-iSRAM + Leak-dSRAM + Leak-IC + Leak-LB + Leak-PB |

Table 4 breaks down energy consumption for each memory operation in the four memory configurations. The last row of this table shows the memory components involved in the static energy consumption, which corresponds to leakage currents and must be considered even when the whole memory hierarchy is idle.

In sections 5.1, 5.2 and 5.3, for each memory configuration we optimize each task for either time (labeled *Opt. time*) or energy (labeled *Opt. energy*). The plots in these sections present these values in vertical bars, normalized to the *Direct-eSRAM* configuration. So, values below 1 represent that the tested system performs the same work in less time or using less energy. Results for the *LB-only* configuration are also normalized to the *Direct-eSRAM* system and are added to the same plots as horizontal lines. In Section 5.4 we focus on a specific benchmark on a concrete memory hierarchy in order to show how optimizations combining time and energy with different weights can be used for fine tuning particular systems.

An important point to consider is that our results do not deal with isolated benchmarks, but include the effects of the real-time multitasking scheduling. That is, they include the effects of the context switches in the worst case, and also the energy leakage for idle periods. To the best of our knowledge, no previous work has studied how the energy consumption varies depending on the estimated number of context switches. As we will see below, an accurate estimation of this value is far more critical for energy consumption than for time, especially when the number of context switches is high.

## 5.1 Multitasking

In Figure 2, we show the normalized WCSRT and WCSRE(see Table 1) for the small and medium task sets. The relative weight of each task can be seen in the periods of Table 2. The relation between weights determines the estimated maximum number of context switches of each task. So, the whole system behavior is mainly defined by the task with the lowest period, which is executed more times, and in a second term by the other tasks plus their context switch overheads.

Firstly, it can be observed that energy consumption has a much higher variability than time. That is, small changes in time usually result in important changes in energy consumption. This relation between time and energy can be observed in all our experiments. In general, Figure 2 shows that prefetch increases energy

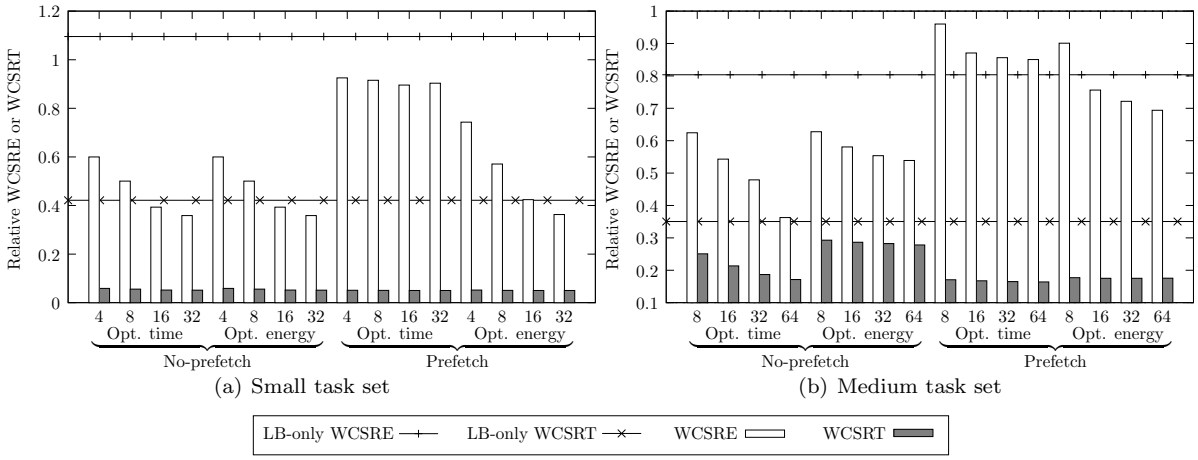



Figure 2: WCSRT and WCSRE (see Table 1) of *No-prefetch*, *Prefetch* and *LB-only* normalized to the *Direct-eSRAM* configuration for the small (a) and medium (b) task sets. The number of sets (of 16 bytes) of IC (always direct-mapped) varies from 4 to 64.

consumption and may reduce the system response time. Also, increasing the cache size usually benefits both time and energy. However, note that too much size would not be beneficial, specially when the number of context switches is high. In such cases, the cost (in terms of time and energy) of preloading the cache at each context switch could be higher than the cost of the actual misses, and the optimizer would choose not using the cache. Also, a larger cache implies a higher leakage and energy consumption per operation, so sizes larger than a given saturation point would provide worse results. This saturation point has already been observed in previous studies [3, 4]. In our experiments this saturation point in cache size is reached with the largest tested cache size, i.e. larger cache sizes (not shown in figures) perform worse.

When considering the memory configuration without prefetch, a seemingly contradictory result shows up in Figure 2(b). Namely, when optimizing task energies, both the system response time and the overall energy consumption become higher than when optimizing task times. The following analysis explains these facts. Since optimizing energy may increase the task execution times, a higher number of context switches may occur, which in turn raises the energy consumption. On the contrary, optimizing time may reduce the number of context switches, which may reduce the system energy consumptions.

Further, note that simple components such as the LB are commonplace in real-time studies because they always reduce the execution time. For instance, the response time of *LB-only time* in Figure 2 is around 0.4 times the time in *Direct-eSRAM* systems. Intuitively, it would be expected the LB to reduce the energy consumption too, since it reduces the number of memory accesses. However, our results show that in some cases it may increase energy consumption (*LB-only energy* above 1 in Figure 2(a)). This is a very interesting result, since it shows that even trivial hardware components can behave differently for time and energy. In this case, the line buffer reduces the number of memory accesses by fetching an instruction line (4 instructions), which consumes almost the same energy (0.27 pJ) as performing 4 smaller memory accesses (0.08 pJ) according to Table 3. However, some of the fetched instructions may not be required, so in these cases the energy consumed would be higher.

Although multitasking experiments show the general behavior, they depend very much on the selected periods and the real-time scheduler used, i.e. results are very dependent on the combination of tasks. Thus, a detailed per-task analysis is desirable for better understanding how the memory architectures and optimization goals affect each particular task. Next sections show such detailed results, both for the small and medium task sets.

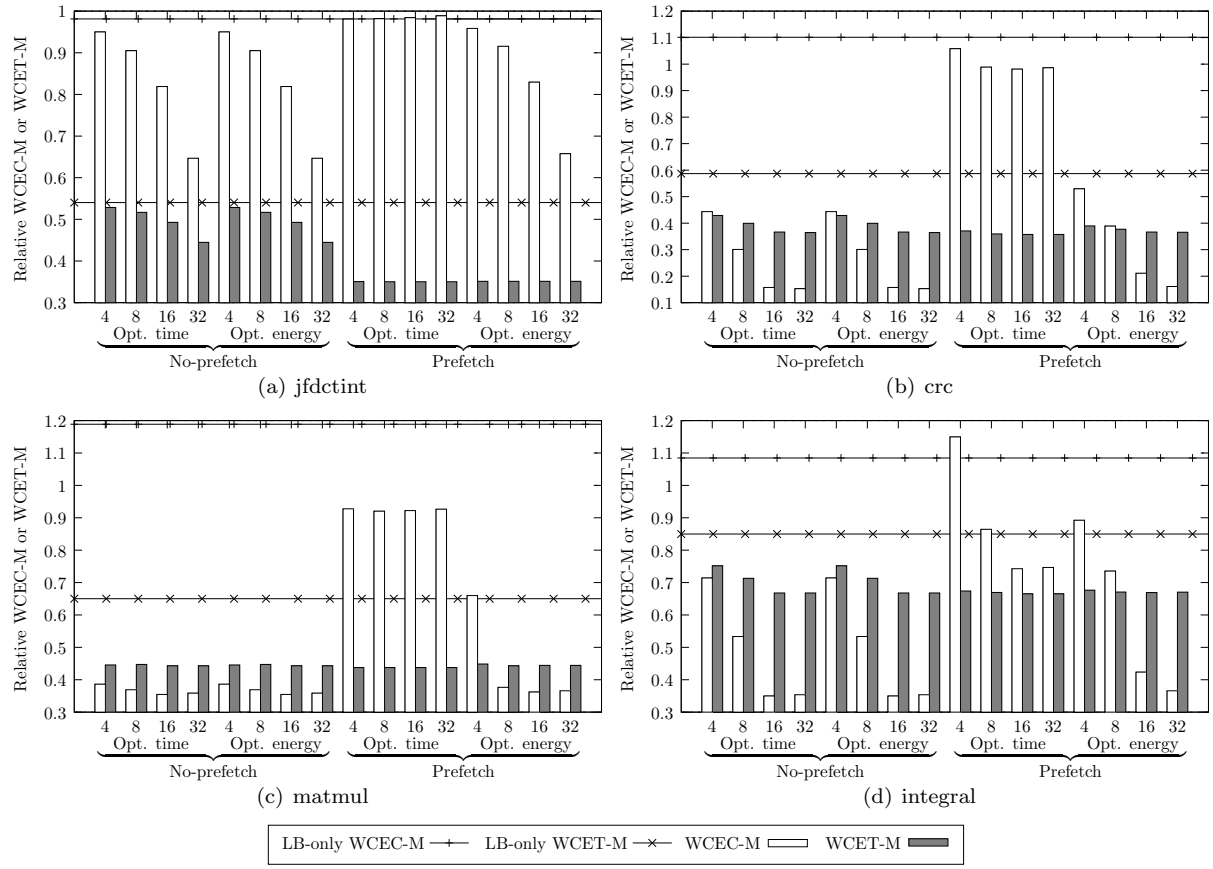



Figure 3: Normalized WCET-M and WCEC-M (see Table 1) of the considered memory configurations for each task in the small task set.

## 5.2 Small task set

Figure 3 focus on the individual tasks in the small task set, showing time and energy normalized to the *Direct-eSRAM* configuration. Its bars and lines correspond to the normalized WCET-M or WCEC-M (see Table 1) of each task.

The leftmost half of each plot shows the configurations without prefetch. Such configurations are not sensitive to the optimization goal, either time or energy. Using a prefetchless configuration, both time and energy decrease as the IC size increases. However, the larger the IC, the higher the cost of preloading at every context switch. This cost refers to the time needed to refill a larger IC (containing more instruction lines) and to the individual energy cost of each refilled instruction line (see Table 3). In the search for an optimal solution, the ILP solver can decide not to preload an instruction line if that cost is not offset by the cost of its misses in the worst case at runtime. As a result occupancy of the IC may be very small for large IC sizes. This can be seen in values 16 to 32 IC sets, in figures 3(b), 3(c) and 3(d), where time and energy no longer decrease and, in some cases, energy increases. Indeed, Figure 3(c) becomes saturated with even fewer sets, since this benchmark is essentially a tiny loop.

The rightmost half of each plot shows configurations with IC, LB and prefetch. In this configuration, results do depend on the optimization goal. When optimizing time, the contents to lock into IC are selected so that prefetch is used somehow more aggressively. On *flat codes*, where the control flow follows large sequences of different basic blocks in most of the code (usually found as loops with a very large body), prefetch improves very much the execution time. This can be seen in Figure 3(a), where the time bars reach much lower values

with prefetch configurations. On the other hand, iterative codes (figures 3(b), 3(c) and 3(d)) do not take so much advantage of prefetch, and time values remain very similar to those without prefetch. However, energy presents opposite values. Using prefetch with the goal of minimizing time generally results in an important waste of energy, which may be even higher than the *Direct-eSRAM* configuration (e.g. see Figure 3(b) and Figure 3(d) for a 4-set IC). If the goal is to minimize the energy consumption and the IC is large enough, in general, prefetching works similarly to not having prefetch, both for time and energy. This means that IC is large enough to keep the most useful instruction lines and prefetch adds a marginal performance gain. On flat codes (Figure 3(a)), as above, prefetching presents better time results and adds no additional energy consumption.

Therefore, for our small task set, we can conclude that prefetch is only interesting for flat codes and, if present, the goal of optimizations should focus on energy and not on time. A more practical approach would be to be able to enable/disable prefetching on a per-task basis. In this way, after performing a time/energy analysis and optimization, each particular task could enable/disable prefetching. Regarding the IC size, in general the same approach could be taken: provide a large enough direct mapped lockable cache with the possibility of adapting/fitting its size to the executed task size (for example by disabling some sets). This would allow to use the best suited size, i.e. having the required sets and disabling those unused to avoid their energy leakage.

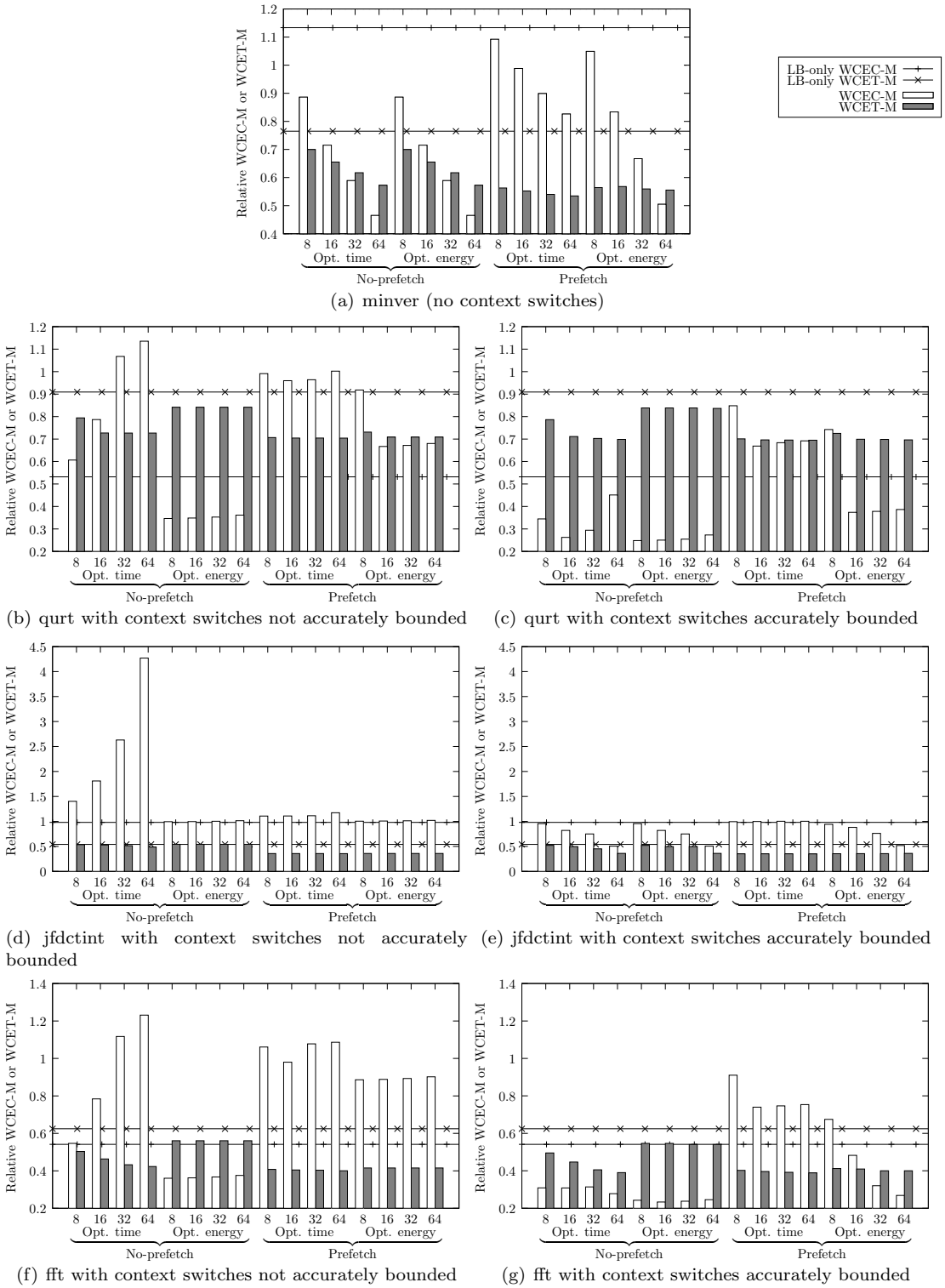
### 5.3 Medium task set

When the maximum number of context switches is small, as in the small task set, it has little impact on results. However, when this number grows, its impact can be much bigger than expected, especially on energy consumption. This is the case for the medium task set. Since the number of context switches is high, the optimizer will be more restrictive on the selection of instruction lines to be locked into cache, due to they must be reloaded at each context switch. Thus, the selected lines and in turn the resulting system behavior will depend very much on the accuracy of the estimated maximum number of context switches. In order to show its effects, Figure 4 includes plots (right side) using an accurately bounded number of context switches (eq. 6), as the small task set, and also (left side) using a trivial overestimated bound (eq. 5). Obviously, the task with the highest priority (Figure 4(a)) is never evicted, so it has no context switches (apart from those when starting its own execution).

For each task, when comparing the resulting values as the accuracy in bounding the number of context switches increases (left vs. right plots), we observe a large reduction in energy, especially when there is no prefetching. That is, the energy consumption for refilling the IC at each context switch plus the energy required in the worst case to refill the flushed contents of the line and prefetch buffers may be a dominant component in the energy accounting when the number on context switches is largely overestimated. Thus, the accuracy on the estimation of the number of context switches is critical. For instance, for the 64 set IC without prefetch running *jfdctint*, as the bound in the number of context switches is tightened, the WCEC-M gets divided by more than eight (see Figure 4(d) and 4(e)). However, time decrements are much lower. This is a very important result, since it shows that real-time systems mainly focusing on time may not be aware of the important energy consumption effects of the accuracy on the estimation of the number of context switches, so they might present extremely overestimated energy values.

Note also that, using an overestimated number of context switches, energy consumption has a clear upwards trend along the leftmost four energy values (i.e. no prefetch and time optimization). That is, as the cache size increases, optimizing time tries to use it as effectively as possible and this means selecting many instruction lines to lock into cache. However, the energy required to preload such contents on every context switch may result in an extremely high energy consumption, which may not worth the savings in execution time. In all other cases (optimizations with prefetch and optimizing energy without prefetch), in general both energy and time values with an overestimated number of context switches remain relatively flat when increasing cache size. Thus, when the number of context switches is too much overestimated, small caches are in general a safer option.

Finally, take into account that in general one does not decide whether to use an accurate bound on the number of context switches or not, but each estimation method provides a particular bound [10, 23]. Thus,




(a) minver (no context switches)

(b) qurt with context switches not accurately bounded

(c) qurt with context switches accurately bounded

(d) jfdctint with context switches not accurately bounded

(e) jfdctint with context switches accurately bounded

(f) fft with context switches not accurately bounded

(g) fft with context switches accurately bounded

Figure 4: Normalized WCET-M and WCEC-M (see Table 1) of the considered memory configurations for each task in the medium task set. The left side plots use a trivial bound on the maximum number of context switches (eq. 5), whereas the plots on the right side correspond to an accurate bound for the number of context switches (eq. 6).

given the huge differences that such estimations may cause, it is important to work with the method that best fits a given system.

Focusing now on the results with an accurate number of context switches and without prefetch (leftmost half of plots on the right in Figure 4), they show a time-energy trade-off depending on the optimization goal. Although in Figure 4(a) (no context switches), and Figure 4(e) (flat code) differences are minimal, in general the behavior should be similar to that of figures 4(c) and 4(g), which show the expected trade-off. Assuming that in real-time systems, time is more critical than energy consumption, systems without prefetch should probably optimize execution time. Further, note in Figure 4(c) the IC size effect described above: when increasing the IC size, the energy consumption grows and the time presents minor variations. Since this benchmark requires very few cache sets, increasing the cache size means a higher energy consumption without improvement in time.

Let us focus now on configurations with prefetch and an accurate number of context switches (rightmost half of plots on the right in Figure 4). Systems with prefetch present minor variations in time independently of their optimization goal. That is, their time results are almost identical when optimizing time and when optimizing energy. This means that prefetch acts as the the main time-reduction factor, and the optimization goal has little effect on time. On the contrary, energy consumption depends very much on the optimization goal. In general, optimizing time implies a much higher energy consumption than optimizing energy. This means that, in order to reduce a few processor cycles, the system uses much more energy. See for instance figures 4(c) and 4(g), which present similar times but much lower energy consumption values when optimizing energy. Thus, on real-time systems with prefetch, even assuming that time is critical, optimizing energy provides better global trade-offs between energy and time than optimizing time.

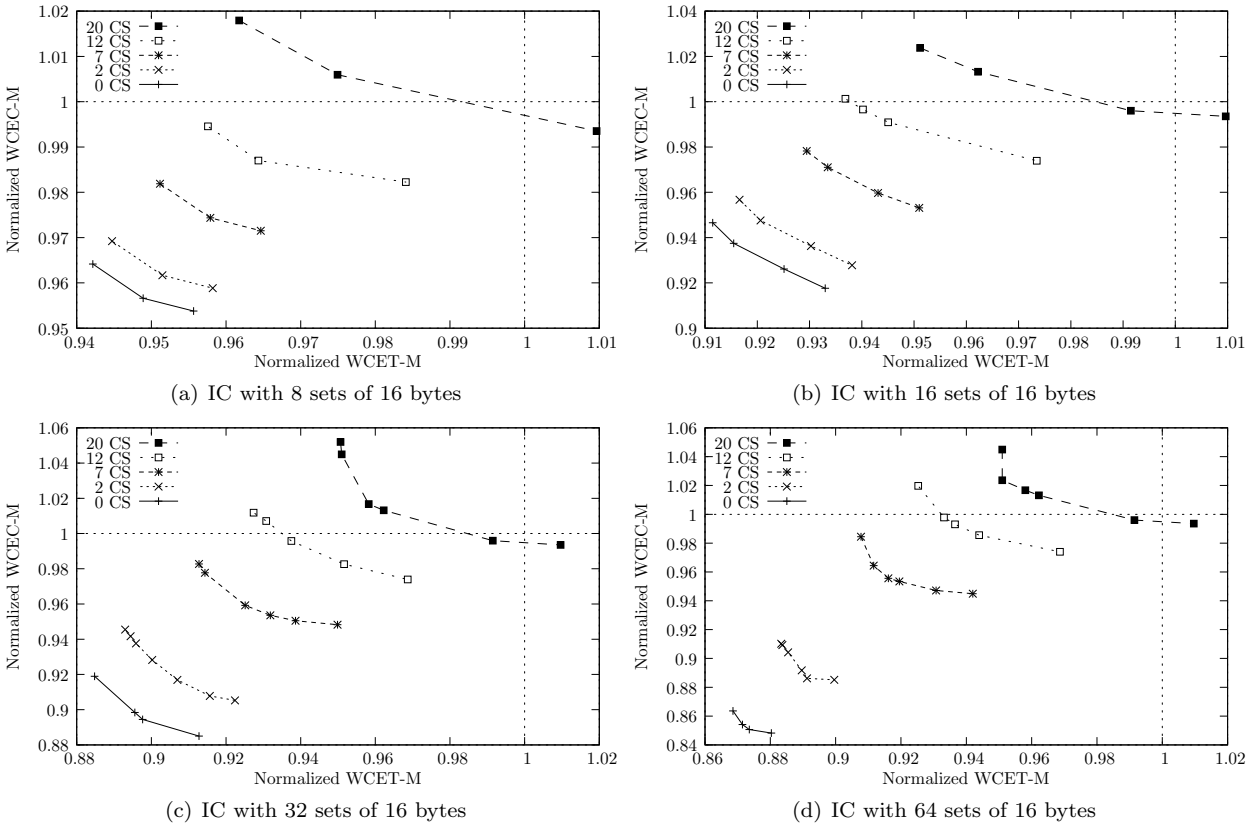
Next, we compare the tasks without prefetch and optimizing time to those with prefetch and optimizing energy when the number of context switches is accurate (i.e. left-most and right-most 4 pairs of bars of plots on the right in Figure 4). For flat codes (Figure 4(e)) prefetch is in general the best option as stated above, since it reduces time very much. For the other tasks, times show lower or very similar values for systems with prefetch. Regarding energy, prefetch usually implies more energy consumption, since it performs memory accesses that may not be required. Thus, disabling prefetch should be recommended except for flat codes. However, note that the trivially bounded bars of energy consumption are higher without prefetch, which means that if the number of context switches is not estimated accurately, the resulting values for energy consumption may be higher without prefetch than with prefetch.

## 5.4 Gradual optimization

In this section we perform a gradual optimization of a particular task on a specific memory organization. That is, apart from showing the extreme values of optimizing time or energy (*Opt. time* and *Opt. energy*) as above, we use several values for $\alpha$ (eq. 3).

The specific memory hierarchy organization selected includes prefetch, line buffer and instruction cache, and the experiments show the worst-case performance (time and energy) of the matrix inversion benchmark (*minver*). Several scenarios with a different number of context switches are used, and results are normalized against the worst one. That is, a system with so much context switches that the instruction cache is present, but not used at all. Although this normalization prevents direct comparisons with previous experiments, it allows us to focus on the details of this particular memory organization.

Figure 5 shows the results of these experiments, representing time in the horizontal axis and energy in the vertical one. That is, the lower left, the better. Although both axis show small steps, note that the tested benchmark is not very large, and its behavior has a limited variability. That is, larger tasks implementing more complex algorithms are expected to show more points spreading over a wider range. The normalization point (coordinates $(1, 1)$) represents that the cost of preloading contents at each context switch equals that of always missing in cache. Each chart corresponds to a particular IC size, and shows several lines representing the worst case behavior with a different estimation of the number of context switches. Extreme points in such lines represent the optimization of either time ($\alpha = 1$, upper left) or energy ($\alpha = 0$, lower right). Intermediate points represent different values of $\alpha \in [0, 1]$. As expected, optimizing time requires more energy, and optimizing energy consumption results in a longer execution time. Also, the lower the number




(a) IC with 8 sets of 16 bytes

(b) IC with 16 sets of 16 bytes

(c) IC with 32 sets of 16 bytes

(d) IC with 64 sets of 16 bytes

Figure 5: Gradual optimization of the matrix inversion benchmark (*minver*) with instruction cache, line buffer and prefetch (*Prefetch* in Table 4), showing normalized WCET-M and WCEC-M. Extreme points in lines represent the optimization of either time ($\alpha = 1$, upper left) or energy ($\alpha = 0$, lower right). Intermediate points represent different values of $\alpha$ (eq. 3).

of context switches expected, the better the system performs. Thus, depending on the schedulability margin and the number of context switches in the worst case, one could choose the most adequate optimization for each task, i.e. choosing an optimization that, guaranteeing schedulability, uses the lowest possible energy. Also, depending on the situation, interesting points may appear. For instance, in this benchmark, having an IC size of 64 sets (Figure 5(d)) and 20 context switches, blindly minimizing time would not be advisable, since the next point (with $\alpha = 0.85$) achieves an almost identical WCET-M with a much lower WCEC-M. Similarly, if there are no context switches (i.e. this task being the one with the highest priority), optimizing WCEC-M with a 64-set IC would not be advisable, since the next point ($\alpha = 0.5$) achieves a very similar WCEC-M with a much lower WCET-M.

In conclusion, once a specific system has been chosen, this gradual analysis could be applied to its tasks in order to look for the most adequate optimization combining time and energy. This would allow a very precise system set-up in a design phase, which in turn could result in more efficient products.

# 6 Conclusions

In real-time systems, time is usually so critical that other parameters such as energy consumption are often not even considered. When energy is considered, it remains as a secondary factor, i.e. execution time is analyzed/minimized first and then the worst energy consumption of the resulting system is calculated.

However, optimizing the worst energy consumption case can be a key factor in real-time and embedded systems that are limited by their batteries or those not powered by mains, such as satellites. In this work, we look into a combined WCET-WCEC analysis and optimization method in the presence of a lockable instruction cache. That is, it chooses the optimal set of instructions to be locked into the cache of our memory hierarchy.

We present results for several memory architectures using this combined WCET-WCEC optimization model for real-time multitasking systems. To model the WCET of a task, we use the Lock-MS optimization method. In order to add the WCEC computation, we apply the previous method but using energy coefficients instead of time coefficients. So, the combined WCET-WCEC model of each task can be used to optimize any linear combination of both WCET and WCEC.

In our experiments we assume a Harvard architecture aiming at low-energy consumption, with different combinations of a lockable instruction cache, a line buffer and sequential prefetching. The inherent predictability of such hardware has allowed us to study the WCEC on a multitasking system for the first time. Such study includes the influence of the estimation of the maximum number of context switches. Our results show that, when this value is not accurately bounded, WCEC values can reach up to 8 times the actual WCEC. This effect is much lower on the WCET. Thus, when accuracy is not possible, the basic recommendation would be to use very small caches (which use less energy) and optimize the system for energy consumption in order to minimize the overestimation.

When the maximum number of context switches can be estimated accurately, the type of task determines its best memory hierarchy configuration and optimization goal. For tasks based on flat codes (i.e. large sequences of different basic blocks, usually found as loops with a very large body), prefetch is specially beneficial, reducing very much the execution time in the worst case with minor energy increments. In such tasks, the instruction cache size does not affect the execution time, but it helps reducing the energy consumption. For these codes, optimization should focus on energy consumption, which provides similar time results to optimizing time, with much lower energy consumption. On the other hand, tasks based on iterative control flow obtain more benefits from a large enough lockable cache, since it provides temporal locality. For these cases, conservative memory hierarchies without prefetch and optimizing time should be a better option in general. Nevertheless, performing a combined optimization with different weights of WCET and WCET for each task on a particular hardware is always recommended, since the most adequate results may be located on such combined optimization weights.

Finally, in order to make the most of the processor in terms of time and energy, it would be very interesting some reconfiguration ability at task granularity. First, by fitting the size of the IC to requirements of the executed task, since sizes of the IC larger than the optimal one do not harm performance but increase energy consumption. Second, by activating or deactivating sequential prefetching. We can provide this mechanism for tasks that really benefit from it or for tasks that really need it for achieving time constraints. Otherwise prefetch implies an increasing energy consumption.

Further research would be needed in order to include the WCET-WCEC values of tasks into existing real-time schedulers (apart from *Rate Monotonic*, used in this paper). This would allow us to provide a precise bound on the lifetime of a real-time system running on a limited energy supply. Also, studies combining WCET, WCEC and dynamic voltage scaling could lead to much more energy-efficient systems.

# 7 Acknowledgement

This work was supported in part by grants TIN2007-60625 and TIN2010-21291-C02-01 (Spanish Government and European ERDF), gaZ: T48 research group (Aragon Government and European ESF), Consolider CSD2007-00050 (Spanish Government), and HiPEAC-2 NoE (European FP7/ICT 217068).

Table 5: Cacti Parameters

| Flag | Value | | | |
|---|---|---|---|---|
| | SRAMS | 1KB IC | 512B IC | 256B |
| -size (bytes) | 131072 | 1024 | 512 | 256 |
| -block size (bytes)(line size) | 16 | 16 | 16 | 16 |
| -read-write port | 1 | 1 | 1 | 1 |
| -UCA bank count | 1 | 1 | 1 | 1 |
| -technology (u) | 0.032 | 0.032 | 0.032 | 0.032 |
| -Data array cell type | itrs-lstp | itrs-lop | itrs-lop | itrs-lop |
| -Data array peripheral type | itrs-lstp | itrs-lop | itrs-lop | itrs-lop |
| -Tag array cell type | itrs-lstp | itrs-lop | itrs-lop | itrs-lop |
| -Tag array peripheral type | itrs-lstp | itrs-lop | itrs-lop | itrs-lop |
| -output/input bus width | 128/32 | 128/32 | 128/32 | 128/32 |
| -operating temperature (K) | 350 | 350 | 350 | 350 |
| -cache type | ram | cache | cache | cache |
| -tag size (bits) | 16 | 10 | 11 | 12 |
| -access mode | fast | fast | fast | fast |
| -design objective | 100:0:0:100:0 | 100:0:0:0:0 | 100:0:0:0:0 | 100:0:0:0:0 |
| -deviate | 100000:100000:100000:100000:1000000 | 0:100000:100000:100000:1000000 | | |
| -Optimize ED or ED^2 | NONE | NONE | NONE | NONE |
| -Cache model (NUCA, UCA) | UCA | UCA | UCA | UCA |
| -Wire signalling | default | default | default | default |
| -Wire inside mat | global | global | global | global |
| -Wire outside mat | global | global | global | global |
| -Interconnect projection | conservative | conservative | conservative | conservative |
| -Add ECC | yes | yes | yes | yes |
| -Force cache config | false | false | false | false |

# A Cacti parameters

In order to obtain energy consumption estimations for our memory hierarchy components, we use the Cacti 6.5 toolset. This tool models memory circuits at microarchitectural level. From the microarchitectural model, Cacti calculates an estimation of energy consumption, power dissipation and area of the memory circuit. The operational of Cacti is based on a configuration file with the description of the memory circuit to be modeled. In Table 5, we show the value of the most important parameters of our memory hierarchy components.

In the memory hierarchy model of this work, read and write operations to the different memory circuits can require access to the complete line (128 bits) or to a single element (32 bits). Thus, we simulate both 128 and 32 bits of bus width, without varying the rest of the parameters.

We only show parameter values for SRAMs and some IC sizes (1 KB, 512 B and 256 B). Cacti was not able to find a microarchitectural model for smaller memory circuits (128 B, 64 B and 16 B). So that, we approximate the corresponding values from the ones obtained from the larger caches. In order to do that, we perform a linear interpolation of 128 B and 64 B ICs from 512 B and 256 B ICs. In case of 16 B, a linear approximation results in an inconsistent value for leakage ($< 0$). So that, just in this case, we divide by 4 the energy consumption values of the 64 B IC.

# References

[1] Chart watch: High-performance embedded processor cores. *Microprocessor Report*, 22:26–27, March 2008.

[2] S. Altmeyer, R.I. Davis, and C. Maiza. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *Proc. of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, pages 261–271, December 2011.

[3] L. C. Aparicio, J. Segarra, C. Rodríguez, and V. Viñals. Combining prefetch with instruction cache locking in multitasking real-time systems. In *Proceedings of the IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 319–328, Macau SAR, China, August 2010. IEEE Computer Society Press.

[4] L. C. Aparicio, J. Segarra, C. Rodríguez, and V. Viñals. Improving the WCET computation in the presence of a lockable instruction cache in multitasking real-time systems. *Journal of Systems Architecture*, 57(7):695–706, August 2010.

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

[6] Jian-Jia Chen and Chin-Fu Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (DVS) platforms. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 28–38, Washington, DC, USA, 2007. IEEE Computer Society.

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

[8] Heiko Falk, Sascha Plazar, and Henrik Theiling. Compile-time decided instruction cache locking using worst-case execution paths. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 143–148, New York, NY, USA, 2007. ACM.

[9] R. Jayaseelan, T. Mitra, and Xianfeng Li. Estimating the worst-case energy consumption of embedded software. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 81–90, April 2006.

[10] 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*, 27(9), September 2001.

[11] Y. T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 254–264, December 1996.

[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] 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.

[14] 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.

[15] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2-3):217–247, May 2000.

[16] N. Muralimanohar, T. Balasubramonian, and N. P. Jouppi. Cacti 6.0: A tool to understand large caches. Technical report, University of Utah and Hewlett Packard Laboratories, 2007.

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

[18] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proc. of the IEEE Real-Time Systems Symp.*, December 2002.

[19] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proc. of the Design, Automation Test in Europe Conference Exhibition*, pages 1–6, April 2007.

[20] F. Rossi, P. Van Beek, and T. Walsh. *Handbook Of Constraint Programming.* Elsevier, 2006.

[21] Seoul National University Real-Time Research Group. SNU-RT benchmark suite for worst case timing analysis, 2008.

[22] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28:101–155, 2004.

[23] 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*, pages 41–48, July 2005.

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

[25] Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for tight timing calculations. *ACM Transactions on Embedded Computing Systems*, 7(1):1–38, December 2007.

[26] Jack Whitham and Neil Audsley. Studying the applicability of the scratchpad memory management unit. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 205–214, 2010.