Reconciling real-time guarantees and energy efficiency through unlocked-cache prefetching

Emilio Wuerges, Romulo S. de Oliveira
Dept. of Automation and Systems Engineering
Federal University of Santa Catarina, UFSC
Florianopolis, Brazil
emilio@inf.ufsc.br, romulo@das.ufsc.br

Luiz C. V. dos Santos
Department of Computer Sciences
Federal University of Santa Catarina, UFSC
Florianopolis, Brazil
santos@inf.ufsc.br

ABSTRACT
For real-time tasks, cache behavior must be constrained via cache locking or predicted by WCET analysis. Since the former gives up energy efficiency for predictability, this paper proposes a novel code optimization that reduces the miss rate of unlocked instruction caches and, provenly, does not increase the WCET. We optimized the 37 programs from the Mälardalen WCET benchmark for 36 cache configurations and two technologies. By exploiting software prefetching on top of on-demand fetching, we reduced the memory’s contribution to the energy consumption (by 11.2%), to the average case execution time (by 10.2%), and to the WCET (by 17.4%).

1. INTRODUCTION
With the rise of smartphones and tablets, Mobile Computing requires increasing energy efficiency to execute programs whose complexity keeps raising. Mobile devices are essentially a combination of two subsystems – a “PC” and a “radio”. The former runs the end-user interface and application programs on a multitasking environment supported by a conventional operating system, whereas the latter implements baseband, protocol-stack, and security processing by relying on a multi-tasking environment built on top of anRTOS. This scenario asks for techniques that do not jeopardize predictability when improving energy efficiency. Since an instruction cache may consume around 40% of an embedded processor’s energy [3] and it impacts predictability and throughput, it becomes a relevant optimization target.

Cache controllers exploit locality of reference through on-demand fetching. When it is fully exploited, further miss rate reductions can only be obtained by fetching in advance the items that will not be in cache before they are referenced. To keep the processor from stalling, such prefetching mechanism relies on a non-blocking cache port or prefetch buffer. A smaller miss rate not only decreases the dynamic consumption, but it also shrinks the static energy consumption as it shortens the average-case execution time (ACET).

Despite its impact on consumption and worst-case execution time (WCET), cache prefetching is underexploited in Real-Time Systems, although a solid basis for accurately predicting cache behavior [8, 21] has been laid. One work extended cache abstract semantics to take prefetching into account [22], another exploited it for optimizing the WCET [5], and a recent one combined it with cache locking [2], but their impact on energy efficiency was not evaluated.

This paper presents a novel technique that inserts prefetch instructions for improving the energy efficiency of instruction caches. In contrast with most real-time optimization techniques [5, 4, 14], which target the minimization of the WCET as a single objective, our algorithm relies on the results of preliminary WCET analysis to identify the most profitable prefetches and to determine their insertion points in the execution flow. We claim that our non-conventional use of static WCET analysis drives code optimization towards energy-efficient binaries for real-time applications. We provide theoretical guarantees that the new algorithm does not increase the memory’s contribution to the WCET. Our experiments show that, as compared to standard fetching alone, the technique can provide energy reductions up to 21% with cache capacities from 2 to 4 times smaller, while sustaining the same or superior performance.

The next section reviews conventional prefetching and its tailoring to low-power, real-time systems. Section 3 shows the models adopted for static analysis and formulates the target problem. Section 4 describes the proposed solution by tailoring to low-power, real-time systems. Section 6 draws the overall conclusions. Supplement S.1 formally describes our algorithms and their complexity analysis. Supplement S.2 provides proofs for our theoretical claim. Supplements S.3 to S.5 present an extra example, the detailed set up, and extra experimental results.

2. RELATED WORK AND CONTRIBUTION

Sequential prefetching [18] assumes that the line contiguous to the one containing the current instruction is likely to be referenced and deserves to be loaded to the cache in advance depending on some criterion (next-line always, next-line on miss, or next-line tagged). It can be extended to multiple lines (next-N-line prefetching). However, it does not handle branches efficiently, since a target instruction typically does not lie in a line contiguous to the one containing the branch instruction. This led to more sophisticated techniques. Target prefetching [19] keeps a reference
prediction table (RPT). When a branch is taken, its target address is stored in some RPT entry, which is tagged with the instruction’s own address. When a branch is executed anew, the matching of a tag at some RPT entry induces the prefetch of the block corresponding to the entry’s target address. Note that this implicitly assumes the branch as always taken. To exploit prefetching when the branch is not taken, wrong-path prefetching [13] stores two addresses (target and fall-through) for each branch in the RPT. Although it can be profitable regardless of the taken path, the number of ineffective prefetches may be increased.

As opposed to the techniques discussed above, whose mechanisms are hardwired, software prefetching relies on a special instruction to load a memory block into a cache line. It allows the preclusion of unnecessary prefetches, which pollute the cache and reduce its effective capacity. The use of dominance trees in control flow graphs was proposed as a way of exploiting static program analysis for prefetch placement [12]. By moving prefetch instructions earlier enough in the control flow, their latency is hidden and their potential of migrating out of loop bodies is raised.

Hardware mechanisms often guess the required prefetches, but they do not issue them early enough so as to produce the desired effect. To reduce cache pollution [9], cooperative prefetching [12] was proposed. Hardware control is limited to sequential prefetching while non-sequential flows are handled by software prefetching.

2.1 Prefetching for energy efficiency

Instead of wasting energy in hardware-controlled prefetch, the performance gain obtained by software prefetching can be directly translated into an increase of energy efficiency when software prefetching is combined with dynamic voltage scaling [1]. A recent work [20] confirms the energy inefficiency of hardware prefetching for old technologies, but indicates a distinct scenario for newer ones. Since hardware prefetching contributes to shortening the average execution time, the resulting static energy profit can be larger than the energy cost of hardware prefetching. Therefore, to completely rule out the need for hardware prefetching, a software prefetching technique should not increase the ACET.

2.2 Prefetching under real-time constraints

There are two conflicting views on how to handle caches under real-time constraints. Those who prescribe cache locking [4, 14] (to trade-off performance for predictability) argue that cache-aware WCET analysis [8, 21] often neglects the interference between tasks [15]. They prescribe a combination of instruction prefetching and cache locking [16] [2]. Such works, however, target the minimization of WCET as a single objective and do not report the impact on energy efficiency. On the other hand, those who prescribe the accurate prediction [8, 21] of cache behavior (during WCET analysis) argue that cache locking may unnecessarily give up performance [2]. Under such assumption, the original cache abstract semantics proposed in [8] was extended in [22] to incorporate the effect of next-N-line prefetching. Based on such extension, a later work [5] exploited software prefetching for minimizing the WCET. Unfortunately, since it inserts a prefetch at the beginning of the basic block where the prefetched instruction belongs, the distance between them might be insufficient to hide the latency of the former.

To the best of our knowledge, none of the works advocating the use of prefetching has reported the impact on WCET, ACET, and energy consumption all together.

2.3 The intended reconciliation

Any reduction in performance is unwelcome, since it may impair energy efficiency through static consumption, especially for nanometer caches [17]. Cache locking tends to become less energy efficient as CMOS technology scales down, since it saves dynamic energy by increasing the ACET, i.e. it lengthens the interval during which static power is drained. Thus, as technology evolves and applications ask for higher throughputs under low power budgets, cache-aware WCET analysis seems more adequate for preserving real-time guarantees under increasing energy efficiency requirements.

To design a proper prefetching algorithm for unlocked caches, all the following conditions should hold for each prefetch: Condition 1: It does not increase the WCET (to preserve real-time guarantees); Condition 2: It reduces the miss rate (to save dynamic energy); Condition 3: It does not increase the ACET (to avoid wasting static energy).

To check Condition 1, we reuse classical static analysis [11, 8, 21]. To verify Condition 2, we propose a novel static analysis technique that visits instructions in reverse execution order and employs extended abstract semantics to identify prefetching points. As trace-based ACET estimates would be inefficient to check Condition 3, we propose an alternative approach relying on recent evidence that the ACET is often reduced when the WCET is decreased [6, 7]. This indicates that, despite their different magnitudes, the derivatives of the WCET and the ACET with respect to target technology and cache configuration tend to be correlated. Therefore, if Condition 1 holds, then Condition 3 is likely to hold as far as the optimization mechanism is based on iterative improvement (where the derivative is the optimization driver). That is why we propose an algorithm that transforms a program iteratively as far as an improvement can be observed. Indeed, the expected correlation between Conditions 1 and 3 is confirmed by the results in Section 5.

Although prefetching was already used to optimize the WCET [5], a criterion to select the most energy-efficient prefetches that preserve real-time guarantees was not proposed so far. Our new criterion is described in Section 4.3.

3. MODELING AND FORMULATION

3.1 Cache behavior

For self-containment, we review the main concepts from [8, 21]. The main storage and the cache are divided into blocks of equal capacity. A program item (instruction or data) always resides in a memory block and may also lie in a cache block. A memory block may contain one or more items. A group of a cache blocks is organized as a cache line (or set), where a is the cache’s associativity. A cache is represented by a set of lines \( L = \{l_1, \ldots, l_n\} \) and the main storage by a set of blocks \( S = \{s_1, \ldots, s_m\} \cup \{I\} \), where \( I \) represents an invalid block. A concrete cache state is a function \( c : L \rightarrow S \). The expression \( c(l_i) = s_j \) means that block \( s_j \) is in cache line \( l_i \). \( C_c \) denotes the set of all concrete cache states.

**Definition 1.** An update function \( U : C_c \times S \rightarrow C_c \) defines the new cache state from the state immediately before a reference to a memory block.

To represent the distinct concrete cache states leading to the WCET scenario, the notion of abstract state is used:

**Definition 2.** An abstract cache state is defined by \( \hat{c} : L \rightarrow \mathcal{P}(C_c) \). \( \hat{c} \) is the set of all possible abstract cache states. A state where all blocks are invalid is denoted as \( \hat{c}_I \).
An abstract update function \( \hat{U} : \hat{C} \times S \rightarrow \hat{C} \) handles abstract states. The abstract update functions used in this work are described in [8].

During the concrete execution of a program, when a path branches off, only one of the divergent paths is executed. In abstract interpretation, however, all paths are taken into account. That is why, a join function has to be defined to merge the abstract cache states prior to the convergence point into a single abstract state after it. The join functions used in this work for WCET analysis are described in [8].

Although we rely on such classical functions for preliminary WCET analysis, we propose novel update and join functions to drive code optimization in Sections 4 and S.1.

### 3.2 Conditional execution

**Definition 3.** Given a program, its control flow graph is a directed graph \( \text{CFG} = (B, F) \) where \( bb \in B \) represents a basic block and \( (bb_i, bb_j) \in F \) represents the precedence between \( bb_i \) and \( bb_j \) in a concrete execution of that program.

The Implicit Path Enumeration Technique (IPET) [11] casts the properties of execution paths into an integer linear programming (ILP) formulation, providing efficient static analysis [21] and accurate WCET bounds [8]. It encodes the conservation of execution flow on entry to and on exit from every basic block, instead of explicitly encoding execution paths. For instance, assume that a basic block \( bb_i \) reaches two mutually exclusive basic blocks \( bb_2 \) and \( bb_3 \) and let \( n_{bb} \) be the number of executions of a basic block. The corresponding ILP constraint is \( n_{bb_i} = n_{bb_2} + n_{bb_3} \). This implicitly encodes the fact that \( bb_2 \) and \( bb_3 \) cannot be executed simultaneously; i.e. if the WCET scenario corresponds to the execution through \( (bb_1, bb_2) \), then \( n_{bb_3} = 0 \) in such a scenario.

### 3.3 Determination of the WCET scenario

Given a program \( p \) and a referenced memory item \( r \), let \( t^p_{bb}(r) \) denote the time spent, in the WCET scenario, when accessing that item. Given a basic block \( bb \), let \( t^p_{bb}(bb) = \sum_{r \in b} t^p_{bb}(r) \) be the time spent, in the WCET scenario, when accessing all the memory items referenced in one execution of that basic block. The overall contribution to the WCET induced by all memory items referenced by \( bb \) in \( t^p_{bb}(bb) \times n_{bb} \).

The objective function for the ILP problem is:

\[
\text{maximize: } \sum_{bb \in B} t^p_{bb}(bb) \times n_{bb},
\]

which solution leads to the number of executions of each basic block \( bb \) in the WCET scenario, written \( n^w_{bb} \). Note that \( n^w_{bb} = 0 \) for every \( bb \) not belonging to the WCET path. The overall contribution of an item \( r \) to the WCET is:

\[
\tau^p_w(r) = t^p_{bb}(r) \times n^w_{bb},
\]

where \( B(r) \) represents the basic block to which \( r \) belongs. Given a program \( p \), the overall contribution of the memory system to the WCET is:

\[
\tau^p_w = \sum_{bb \in B} t^p_{bb}(bb) \times n^w_{bb}
\]

### 3.4 Problem formulation

**Definition 4.** The latency of a prefetch instruction, written \( \Lambda \), is the time it takes to place a block in cache.

**Definition 5.** Programs \( p \) and \( p' \) are prefetch-equivalent, written \( p \equiv p' \), if they are indistinguishable, except for their prefetch instructions.

**Problem 1.** Given a program \( p \), find a prefetch-equivalent program \( p' \) such that \( \tau^p_w \leq \tau^{p'}_w \) and it minimizes the number of cache misses, for a given prefetch latency \( \Lambda \), a given cache configuration, and a given process technology.

### 4. THE PROPOSED TECHNIQUE

To solve Problem 1, we deliberately adopted iterative improvement so as to increase the chances that program \( p' \) leads to higher energy efficiency than program \( p \) (as explained in Section 2.3). A joint improvement criterion was designed to evaluate the impact of each prefetch on both miss rate and WCET. From the original program, prefetch-equivalent programs are iteratively generated one after another as far as the joint improvement criterion is satisfied.

#### 4.1 Abstract program representation

As we target the memory subsystem, our program representation abstracts the references to memory items from the concrete instructions of the actual program. It assumes that loops were virtually unrolled beforehand, by applying the transformation (VIVU) proposed in [8], leading to an implicit loop representation where back edges are broken:

**Definition 6.** Given a program, its control abstract flow graph is a polar, directed cyclic graph \( \text{ACFG} = (R, E) \) where each vertex \( r_i \in R \) is a reference to a memory item and each edge \( (r_i, r_j) = r_i \) \( E \) represents the order of precedence between the references \( r_i \) and \( r_j \) in a concrete execution of that program. The poles are the source (●) and the sink (○).

To denote that \( r_i \) reaches \( r_j \) through a path in the ACFG, we write \( r_i \sim r_j \). Each edge defines a program point between successive references. Given two references belonging to convergent execution paths, to stress the precedence between each of them and a third post-dominating reference, we include special join vertices.

**Definition 7.** Given an ACFG \( = (R, E) \), its reverse abstract control flow graph is an undirected cyclic graph \( \text{ACFG}^r = (R, E^r) \) such that there exists an edge \( (r_j, r_i) \in E^r \) for every edge \( (r_i, r_j) \in E \) and vice-versa.

We define the predecessors and the successors of a given vertex \( r \) in \( \text{ACFG}^r \) as \( \text{PREDE}(r) = \{ r' \in R \mid (r', r) \in E^r \} \) and \( \text{SUCC}(r) = \{ r' \in R \mid (r, r') \in E^r \} \), respectively. A given predecessor of \( r \) is denoted as \( \text{pred}^r(r) \).

**Definition 8.** Given an item \( r_i \), we write \( S(r_i) \) to denote the memory block where \( r_i \) is stored. Conversely, given a memory block \( s \), we write \( R(s) \) to denote the reference to the item in \( s \) with the smallest address (i.e. the first item).

Let us now link the proposed representation with the classical model of cache behavior reviewed in Section 3.1.

**Definition 9.** The set of blocks in cache at a given state \( \hat{c} \), written \( B(\hat{c}) \), is \( \bigcup_{i=1}^{\hat{c}} \{ \hat{c}(i) \} \).

Let \( \hat{c}(r_i, r_j) \) be the cache state at program point \( (r_i, r_j) \). Let \( B(r_i, r_j) \) be a shorthand notation for \( B(\hat{c}(r_i, r_j)) \). Given three successive references \( r_{i-1}, r_i, \) and \( r_{i+1} \), the following properties hold:

- **Property 1.** When \( B(r_i, r_{i+1}) - B(r_{i-1}, r_i) = \emptyset \), the access to item \( r_i \) resulted in a hit.
- **Property 2.** When \( B(r_i, r_{i+1}) - B(r_{i-1}, r_i) = \{ s \} \), the access to item \( r_i \) resulted in a miss and \( r_i \) is stored in memory block \( s \).
- **Property 3.** When \( B(r_{i-1}, r_i) - B(r_i, r_{i+1}) = \{ s' \} \), the access to item \( r_i \) replaced the memory block \( s' \).
**4.2 Illustrative examples**

We show how our technique works by means of examples. For simplicity, we assume that all the references map to the same line of a 2-way LRU cache with 2 items per block.

In Figure 1, our technique is applied to a simple straight-line program. From the ACFG of the original program (1a), it shows the ACFG* representing the intermediate optimization steps for each visited vertex (1b) until the ACFG of the optimized program is obtained (1c). Each memory block is represented as a dotted box. The cache states at each program point are displayed at the right-hand side. They help track either the number of misses in program or the replaced blocks in reverse order (1b). The blocks of a cache line are denoted as [MRU, LRU], to indicate the most and the least recently used blocks.

Figure 1a shows the states at each edge of the ACFG. By applying Properties 1 and 2 to every successive pair of edges, we obtain whether the outcome was a hit or a miss.

Figure 1b presents our reverse analysis step-by-step from sink to source. Initially, the edge \((0, r_2)\) is assigned a state where all blocks are invalid. By applying Property 3 to each successive pair of edges, a replaced block can be identified. When \(r_3\), \(r_4\), \(r_5\) and \(r_6\) are visited, since no cache item is replaced, no action is taken but visiting the next vertex. However, when \(r_1\) is visited, the technique detects that the cached item \(s_3\) is replaced. Therefore, a prefetch for the replaced item, denoted as \(\pi_{s_3}\), is inserted at the program point \((r_2, r_1)\). This is done by removing the edge \((r_2, r_1)\) and adding the edges \((r_2, \pi_{s_3})\) and \((\pi_{s_3}, r_1)\). When \(\pi_{s_3}\) is visited, the effect of the prefetch to the cache is merely recalculated (despite the detection of \(s_3\) as a replaced block, which was already treated by \(\pi_{s_3}\) itself). Then the vertex \(r_1\), which is the successor of the inserted prefetch, is revisited. The analysis ends when the node \(\bullet\) is reached.

As shown in Figure 1c, the optimized program is obtained by simply reversing the edges of the resulting ACFG*. Note that, although the references to \(r_1\) and \(r_2\) induce cache misses, the accesses to \(r_3\), \(r_4\), and \(r_5\) do not.

A second example handles conditional constructs with the help of join functions. When a reference \(r\) is reached from distinct paths, the state at its leaving edge depends on the taken path. To derive a single output state from multiple input states, join vertices are added to the ACFG (ACFG*) and their behaviors are modeled by join functions, as illustrated in Figure 2. Figure 2a shows that a vertex \(J\) performing a conventional join function determines the state at its leaving edge as the intersection of the states of its entering edges [8]. Figure 2b shows that a vertex \(J_e\) performing a join function tailored to prefetching simply propagates to its leaving edge the state of the entering edge that belongs to the WCET path. Figures 2c and 2d directly show the resulting ACFG* and ACFG.

**4.3 The joint improvement criterion**

Let us denote the contribution to the WCET of all items referenced on a path starting at \(r_1\) and ending at \(r_j\) as:

\[
\tau^P_{w}(r_1, r_j) = \sum_{r \in \{s \in R \mid r_1 \prec s \prec r_j\}} t^P_w(r) \times n^w_B(r) 
\]  

(4)

Let \(p_{n-1}\) and \(p_n\) denote programs containing \(n-1\) and \(n\) prefetches, respectively, such that \(p_{n-1} \equiv p_n\). Let \(r_j\) be a reference to an item stored in some memory block \(s'\). We denote an instruction that prefetches the block \(s'\) into cache as \(\pi_{s'}\). Finally, let \((r_1, r_{i+1})\) denote some program point such that \(r_1 \prec r_j\). To check if the insertion of \(\pi_{s'}\) at program point \((r_1, r_{i+1})\) precludes the miss on access to \(r_j\) without increasing the WCET, five notions are required.

The first notion tracks prefetch effectiveness, i.e. the guarantee that the prefetched block is in cache before it is referenced, despite the prefetch latency (Definition 4). Given
a program $p_{n-1}$, the time spent, in the WCET scenario, to perform all memory access in between $r_i$ and $r_j$ is:

$$t^{p_{n-1}}_w(r_{i+1}, r_{j-1}) = \sum_{r \in \{x | x \in R \land r_{i+1} \leq x \leq r_{j-1}\}} t^{p_{n-1}}_w(r) \quad (5)$$

**Definition 10.** A prefetch instruction inserted at some program point $(r_i, r_{i+1})$ is effective if $\Lambda \leq t^{p_{n-1}}_w(r_{i+1}, r_{j-1})$.

The second notion tracks the contribution to the WCET of a reference $r_j$ to an item missing in cache at a given point, say $(r_{j-1}, r_j)$, of a program $p_{n-1}$:

$$mcost(r_j) = t^{p_{n-1}}_w(r_j) \quad (6)$$

The third notion tracks the contribution to the WCET of a reference $r_j$ to an item hitting in cache as a result of an effective prefetch instruction $\pi_x$ inserted at point $(r_i, r_{i+1})$ in program $p_{n-1}$, leading to a new program $p_n$:

$$pcost(r_j) = t^{p_n}_w(\pi_x) + t^{p_{n-1}}_w(r_j) \quad (7)$$

The fourth notion tracks the contribution to the WCET from the relocation of all references preceding $r_i$ in the address space, as a result of the insertion of a prefetch instruction $\pi_x$ at point $(r_i, r_{i+1})$ in program $p_{n-1}$, turning it into a new program $p_n$:

$$rcost(r_i) = \sum_{r \in \{x | x \in R \land x < r_i\}} t^{p_n}_w(\pi_x) - \sum_{r \in \{x | x \in R \land x > r_i\}} t^{p_{n-1}}_w(r) \quad (8)$$

The fifth notion combines all the previous concepts to define the profitability of a prefetch. Given two references $r_i$ and $r_j$ such that $r_i \prec r_j$ in the $ACFG$, the profit of inserting an instruction, at the program point $(r_i, r_{i+1})$, to prefetch the memory block storing the item $r_j$ is:

$$profit(r_i, r_j) = \begin{cases} 0 & \text{if } r_j \text{ is a prefetch} \\ 0 & \Lambda > t^{p_{n-1}}_w(r_{i+1}, r_{j-1}) \\ mcost(r_j) - pcost(r_j) & \text{otherwise} \end{cases} \quad (9)$$

A prefetch is profitable if and only if it is effective, the induced relocation does not increase the WCET, and the gain of suppressing a miss (induced by program $p_{n-1}$) is higher than the cost of inserting a prefetch to supress that miss (in a program $p_n$).

### 4.4 The novel optimization algorithm

As a precondition, our algorithm assumes that traditional WCET analysis (to determine $t^{p}_{w}(r)$ for each $r \in R$ and $m^{p}_{w}$ for every $bb \in B$) and classical VIVU analysis (to transform a cyclic CFG into an acyclic ACFG) [8] were performed beforehand. Our algorithm, which is formally described in Supplement S.1, runs a non-conventional static analysis in reverse execution order to find the profitable prefixes that do not increase the WCET. The optimization algorithm relies on the novel update function informally described in Figure 1 and the novel join function illustrated in Figure 2. Their formal descriptions are available in Supplement S.1.

When the program order is preserved for the memory operations at execution time, our optimization algorithm provenly does not increase the contribution of the memory system to the WCET (see Theorem 1 in Supplement S.2).

### 5. Experimental Evaluation

We ran the proposed optimization on all 37 programs of the Mälardalen WCET benchmark [10], each one under 36 cache configurations and two technologies (45nm and 32nm), leading to 2664 use cases. For reproducibility, Supplement S.4 provides the detailed set up. We assume that each program fully owns the instruction cache. Since in practice many programs compete for the cache, the adopted sizes should be interpreted as effective cache capacities allocated to individual programs (not as the overall capacity). We selected them so that the average miss rate lies in a large span from 1% to 10% before the proposed optimization is applied. A 128MB DRAM was employed as level-two memory.

Figure 3 shows average improvements for distinct cache sizes. The overall average improvement was 10.2% for the ACET and 11.2% for energy consumption. Indeed, energy savings were obtained for all use cases without increasing the memory’s contribution to the ACET. To reach such energy efficiency, the maximal increase in the number of executed instructions was 1.32% (see Supplement S.5). The non-increasing ACET has two consequences: the memory’s static energy and the average number of cycles per instruction are not increased (if time anomalies are second-order effects). As the amount of inserted instructions is negligible, our optimization of the memory subsystem may only marginally increase the static consumption of the rest of the system.

Figure 3 also plots the average improvement in the WCET. It confirms the correlation between WCET and ACET variations, which was a premise for the design of our algorithm. Note that, to preserve real-time guarantees, our technique employs the WCET as a constraint and therefore does not try to optimize it. However, an average improvement of 17.4% was observed (see Supplement S.5 for an illustration).

![Figure 3: Impact on energy efficiency](image-url)

![Figure 4: Impact on miss rate](image-url)

As our technique enables the use of smaller caches, it can exploit the resulting reduction in static and dynamic consumption for further improving the energy efficiency, as follows. Figure 5 plots average reductions, but the cache size used to run the optimized programs was set to 1/4 and 1/2 of the cache size used to run the original programs. Note that, within the shaded areas, the optimized programs sustained ACETs less or equal to the unoptimized ones with...
1/2 and 1/4 of the original cache size. Although our technique provides WCET guarantees when the original and the optimized program run on the same cache configuration, it cannot keep such guarantee when comparing their behaviors on configurations with arbitrarily selected sizes. However, as Figure 5 indicates, the WCET did not grow for any use case when cache sizes were reduced. This is an evidence that our technique, by enabling the use of smaller caches, can lead to energy reductions up to 21% while sustaining the same or superior performance and preserving real-time guarantees.

6. CONCLUSIONS AND PERSPECTIVES

We showed that, since conventional WCET analysis should be run anyway to provide for real-time guarantees, a polynomial algorithm can exploit the analysis’ outcome to increase the energy efficiency of a program for a given cache configuration and process technology, while preserving the WCET.

Being independent from locality, prefetching can reduce the required cache size to reach the same performance level as obtained through on-demand fetching on larger caches. This diminishes static and dynamic consumption at level-1 caches. Besides, prefetching seems to harmonize with future hierarchies, where the sensitivity of power consumption to associativity is likely to be reduced [17] to the point of enabling level-2 caches with higher associativities than the current ones. This allows for smaller level-1 caches, and raises the potential of prefetching for energy efficiency.

As the impact of cache locking on energy efficiency has not been reported (to the best of our knowledge), we intend to bridge such gap by implementing techniques like [16] and [2] within our experimental environment. We also intend to generalize our algorithms for handling unlocked data caches.

7. REFERENCES

SUPPLEMENTAL MATERIAL

S.1. ALGORITHMS AND COMPLEXITY

We propose the prefetching update function \( \hat{U}_e : \hat{C} \times R \rightarrow \hat{C} \) defined in Algorithm 1. It detects the need for a prefetch (line 2) and checks if it is profitable (line 4). If so, it inserts the prefetch in the \( ACFG^* \) (lines 5-7) and relocates all memory items affected by such insertion (new block boundaries up to the source vertex). Then it is applied recursively to the inserted prefetch (line 9). If it detects no need for prefetching or an unprofitable prefetch, the conventional update function is applied and the resulting state is returned (line 10).

We also propose the prefetching join function \( J_{SE} : \hat{C} \times \hat{C} \rightarrow \hat{C} \) defined in Algorithm 2. Essentially, it propagates, to the edge leaving a join, the cache state from the entering edge that belongs to the WCET path.

**Algorithm 1** The proposed update function \( \hat{U}_e(\hat{c}, r_i) \)

1. \( s = S(r_i) \)
2. if any \( S' \in S \) such that \( B(\hat{c}) - B(\hat{U}(\hat{c}, s)) = \{S'\} \neq \{I\} \) do
3. \( r_j = R(s') \)
4. if profit(r, r_j) > 0 do
5. \( R := R \cup \{\pi_s\} \)
6. \( E^* := E^* \cup \{(\text{pred}^*(r_i), \pi_s, (\text{succ}^*(r_i), r_i)\} \)
7. \( E^* := E^* - \{(\text{pred}^*(r_i), r_i)\} \)
8. relocate_upwards(r_i)
9. return \( \hat{U}_e(\hat{c}, \pi_s) \)
10. return \( \hat{U}(\hat{c}, s) \)

**Algorithm 2** The proposed join function \( J_{SE}(\hat{c}_1, \hat{c}_2) \)

1. let \( (r_x, \tilde{J}) \in E^* \) if \( \hat{c}(r_x, \tilde{J}) = \hat{c}_1 \)
2. let \( (r_y, \tilde{J}) \in E^* \) if \( \hat{c}(r_y, \tilde{J}) = \hat{c}_2 \)
3. if mcost(r_x) < mcost(r_y) do
4. return \( \hat{c}(r_x, \tilde{J}) \)
5. else
6. return \( \hat{c}(r_y, \tilde{J}) \)

Algorithm 3 builds the \( ACFG^* \) (line 1) and finds a topological ordering \( \prec_T \) of its vertices (line 2). Then it visits vertices in that order from sink (line 5) to source (line 6). If it visits a join, the proposed join function is invoked (line 10); otherwise, the proposed update function is called (line 13). Finally, the optimized \( ACFG \) is built from the \( ACFG^* \) that was modified by the proposed update and join functions (line 15).

Lines 1, 3, and 5–7 of Algorithm 1 take \( O(1) \). Lines 2 and 10 also take \( O(1) \) when cache states are precomputed during the preliminary WCET analysis and stored in a hash table. At line 4, the evaluation of Equations 6 and 7 takes \( O(1) \). Although the second summation of Equation 8 can benefit from precalculated values (and, therefore, takes \( O(1) \)), the first summation takes \( O(|R|) \). The relocation at line 8 also takes \( O(|R|) \). Therefore, Algorithm 1 takes \( O(|R|) \).

Algorithm 1 is called at most \( |R| \) times from the line 13 of Algorithm 3 and, recursively, at line 9, as many times as the number of inserted prefetches, which is at most \( |R| \). Therefore, the line 13 of Algorithm 3 contributes \( O(|R|^2) \) to the overall complexity. All lines of Algorithm 2 take \( O(1) \) due to the hash table and it is invoked at most \( |R| \) times.

As a result, lines 6–14 of Algorithm 3 contribute \( O(|R|^2) \) to the overall complexity, whereas lines 1, 2, and 15 take \( O(|R| + |E|) \). Thus, the overall worst case complexity of Algorithm 3 is \( O(|R|^2) \).

Besides, when generating the \( ACFG = (R, E) \) from the \( CCFG = (B, F) \), we bound the set \( R \) by virtually unrolling each loop at most once when applying the VIVU transformation [8].

**Algorithm 3** The proposed prefetching optimization

1. build \( ACFG^* = (R, E^*) \) from program \( p \)
2. \( \prec_T = \{(u, v) \in R \times R \mid (u, v) \in E \lor (v, u) \in \bar{E} \} \)
3. let \( \text{successor}(r) \) be the successor of \( r \) in \( \prec_T \)
4. \( c(\text{root}(\hat{S})) := c_1 \)
5. \( r := \emptyset \)
6. while \( (\text{successor}(r) \neq \emptyset) \) do
7. \( \{r_x, r_y\} := \text{SUCCESS}(r) \)
8. \( \hat{c}(r, r_x) := \hat{U}_e(\hat{c}(r, r_x)) \)
9. \( \hat{c}(r, r_y) := \hat{U}_e(\hat{c}(r, r_y)) \)
10. \( r := \text{successor}(r) \)
11. build \( ACFG = (R, E) \) for program \( p' \)

S.2. FORMAL GUARANTEES

To improve readability, this section adopts \( \sum_{r \in p_{n-1}} \tau^w_n(r) \) as a shorthand notation for \( \sum_{r \in p_{n-1}} \tau^w_n(r) \).

**Lemma 1.** Given the \( ACFG \) representing a program \( p_{n-1} \) and a path \( (r_i, r_{i+1}, \cdots, r_j) \), if Algorithm 1 inserts, at program point \( (r_i, r_{i+1}) \), a prefetch \( \pi_s \) for a block \( s' = S(r_i) \), then \( \text{block cost}(r_j) < \text{block cost}(r_j') \).

**Proof.** Line 4 of Algorithm 1 guarantees, via Equation 9, that a prefetch \( \pi_s \) is inserted only if \( \text{block cost}(r_j) < \text{block cost}(r_j') \), which from Equations 6 and 7 leads to \( \tau^w_n(r_j) < \tau^w_n(r_j') \). Since \( \pi_s \) is inserted immediately before \( r_{i+1} \) and every vertex \( s \) such that \( r_{i+1} \sim s \sim r_j \) is untouched by Algorithm 1, we can write \( \tau^w_n(r_{i+1}, r_j) = \tau^w_n(r_{i+1}, r_{i+1-1})(\Pi) \). Thus, from (I) and (II) we conclude that \( \tau^w_n(r_j) < \tau^w_n(r_{i+1}, r_j) + \tau^w_n(r_{i+1-1}) + \tau^w_n(r_{i+1}, r_j) \). Therefore, from Equations 2 and 4, we can write \( \tau^w_n(r_{i+1}) < \tau^w_n(r_{i+1}, r_j) \).

**Lemma 2.** Given the \( ACFG \) representing a program \( p_{n-1} \) and a path \( (r_i, r_{i+1}, \cdots, r_j) \), if Algorithm 1 inserts, at program point \( (r_i, r_{i+1}) \), a prefetch \( \pi_s \) for a block \( s' = S(r_i) \), then \( \text{cost}(p_{n-1}) \) of all memory items reaching \( r_i \) is not increased, i.e., \( \sum_{r \in p_{n-1}} \tau^w_n(r) \leq \sum_{r \in p}_{n-1} \tau^w_n(r) \).

**Proof.** Line 4 of Algorithm 1 guarantees, via Equation 9, that a prefetch \( \pi_s \) is inserted only if \( \text{cost}(r_j) < \text{cost}(r_j') \), which from Equation 8 leads to \( \tau^w_n(r_{i+1}, r_j) - \tau^w_n(r_{i+1}, r_{i+1-1}) \leq 0 \). Therefore, from Equations 2 and 4, we can write \( \tau^w_n(r_{i+1}) < \tau^w_n(r_{i+1}, r_j) \).

**Theorem 1.** Given a program \( p \), Algorithm 3 produces a program \( p' \) such that \( p' \equiv p \) and \( \tau^w_n(r) \leq \tau^w_n(r') \) if all memory operations are kept in program order at execution time.
Proof. Let $p_{n-1}$ denote the program generated by Algorithm 3 after inserting $n-1$ prefetches prior to some invocation of Algorithm 1 in which the condition in line 4 holds. This means that a prefetch $\pi'$ for a block $s' = S(r_j)$ will be inserted at point $r_{i+1}$ of program $p_{n-1}$, thereby generating a program $p_n$ with $n$ prefetches. From Equations 2 and 3, we can write:

$$\tau_{p_{n-1}} = \sum_{bb \in B} t_{bb}^{p_{n-1}}(bb) \times n_{bb} = \sum_{r \in R} \tau_{p_{n-1}}^{p_{n-1}}(r),$$

$$\tau_{p_{n}} = \sum_{bb \in B} t_{bb}^{p_{n}}(bb) \times n_{bb} = \sum_{r \in R} \tau_{p_{n}}^{p_{n}}(r),$$

which can be rewritten, with the help of Equation 4, as follows:

$$\tau_{p_{n-1}} = \sum_{r=t_{j+1}} \tau_{p_{n-1}}^{p_{n-1}}(r) + \tau_{p_{n}}^{p_{n-1}}(r_{j+1}, r_j) + \sum_{r \in R, r \neq r_{j+1}} \tau_{p_{n-1}}^{p_{n-1}}(r)$$

$$\tau_{p_{n}} = \sum_{r=t_{j+1}} \tau_{p_{n}}^{p_{n}}(r) + \tau_{p_{n}}^{p_{n}}(r_{j+1}, r_j) + \sum_{r \in R, r \neq r_{j+1}} \tau_{p_{n}}^{p_{n}}(r)$$

When $p_{n-1}$ is turned into $p_n$, all paths starting at $r_{j+1}$ are untouched by Algorithm 1 and Lemma 2 holds. Therefore, we can write:

$$\sum_{r=t_{j+1}} \tau_{p_{n}}^{p_{n}}(r) + \sum_{r \neq r_{j+1}} \tau_{p_{n}}^{p_{n}}(r) \leq \sum_{r=t_{j+1}} \tau_{p_{n-1}}^{p_{n-1}}(r) + \sum_{r \neq r_{j+1}} \tau_{p_{n-1}}^{p_{n-1}}(r)$$

Therefore, we conclude that:

$$\tau_{p_{n}}^{p_{n}} - \sum_{r \neq r_{j+1}} \tau_{p_{n}}^{p_{n}}(r) \leq \tau_{p_{n}}^{p_{n-1}}(r_{j+1}+1, r_j) \leq \tau_{p_{n}}^{p_{n}}(r_{j+1}, r_j)$$

where $K = \tau_{p_{n-1}}^{p_{n-1}}(r_{i+1}, r_j) - \tau_{p_{n}}^{p_{n}}(r_{j+1}, r_j) \leq \tau_{p_{n}}^{p_{n}}(r_{j+1}, r_j)$.

Since $p_n$ and $p_{n-1}$ are indistinguishable except for $\pi'$ and we know from Lemma 1 that $K > 0$, we conclude that $p_n \equiv p_{n-1}$ and $\tau_{p_n}^{p_{n}} \leq \tau_{p_{n-1}}^{p_{n-1}}$ hold for any integer $n > 1$, i.e. $p' \equiv p$ and $\tau_{p'}^{p'} \leq \tau_{p}^{p}$ hold for any program $p'$ produced by Algorithm 3. If, however, Algorithm 3 does not insert any prefetches ($n = 0$), i.e. $p' = p = p_0$, we obviously have $p \equiv p'$ and $\tau_{p_0}^{p_0} \leq \tau_{p}^{p}$ hold for any program $p'$ produced by Algorithm 3 from $p$.

S.3. HOW LOOPS ARE HANDLED

This supplemental example illustrates that, to handle loops, our technique relies on the VIVU transformation [8] (which is often employed by conventional WCET analysis) to derive an acyclic ACFG from a cyclic CFG. Figure 6a shows a CFG prior to the VIVU transformation, where a back edge closes a loop. Figure 6b shows the transformation’s effect: the back edge is broken and the loop body is instantiated twice, leading to an ACFG where the effect of loop iteration is implicitly encoded in the conditional flow. In that figure, $r^2_j$ denotes the reference to an item $r_2$ in the first loop iteration and $r^3_j$ denotes the reference to the same item in other loop iterations. From the ACFG in Figure 6b, our technique obtains the ACFG* in Figure 6c, according to the mechanisms already illustrated in the previous examples. Figure 6d shows the resulting CFG for the optimized program.

S.4. DETAILED EXPERIMENTAL SETUP

Table 1 labels all the adopted benchmark programs.

The cache configurations employed in our experiments are denoted as $k = (a, b, c)$ in Table 2, where $a$ is the associativity, $b$ is the block size (in bytes), and $c$ is the cache capacity (in bytes).

Our technique was integrated into the GNU compiler (version 4.6.1). We used the '-O2' optimization level and targeted ARMv7. First, the source code of each program was compiled by disabling the proposed technique, leading to a single non-optimized executable file for each program $p$, denoted as $e_p$, to be used as a reference. Then each source file was compiled by applying our technique to every cache configuration $k$ and each process technology $t$, resulting in a set of optimized executable files for each program $p$, denoted as $\{e_{p,kt}\}$.

For each executable file $e$, we estimated the contributions of the memory system to the WCET and to the ACET, written $\tau_{e}(c)$ and $\tau_{e}(c)$, respectively, and the energy consumption of the memory system in the ACET scenario, written $e_{\text{ac}}(c)$. The first estimate was obtained with conventional WCET analysis, whereas the last two were obtained through a traditional trace-based approach. For trace generation, we employed an instruction-set simulator available within the GEM5 simulation environment. Since our optimization heavily relies on WCET analysis, we implemented our own WCET analyzer based on [8, 21] and integrated its components into the optimizing tool prototype.

We employed the CACTI 6.5 power/energy model to obtain energy and access times for the primary cache and the level-two memory. Since we did not model the processor’s micro-architecture, we did not estimate the impact of the instruction overhead on the processor’s energy consumption and execution time. However, since the measured increase in the number of executed instructions was negligible, the impact is likely to be marginal.
Table 1: Program identification

<table>
<thead>
<tr>
<th>program ID</th>
<th>program ID</th>
<th>program ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>p1</td>
<td>bs</td>
</tr>
<tr>
<td>cnt</td>
<td>p4</td>
<td>compress</td>
</tr>
<tr>
<td>crc</td>
<td>p7</td>
<td>duff</td>
</tr>
<tr>
<td>expint</td>
<td>p10</td>
<td>fac</td>
</tr>
<tr>
<td>ft1</td>
<td>p13</td>
<td>icall</td>
</tr>
<tr>
<td>insertsort</td>
<td>p16</td>
<td>anne_complex</td>
</tr>
<tr>
<td>lctnum</td>
<td>p19</td>
<td>ins</td>
</tr>
<tr>
<td>ldcmap</td>
<td>p22</td>
<td>matmult</td>
</tr>
<tr>
<td>minver</td>
<td>p25</td>
<td>udes</td>
</tr>
<tr>
<td>nsichneu</td>
<td>p28</td>
<td>prime</td>
</tr>
<tr>
<td>qurt</td>
<td>p31</td>
<td>recursion</td>
</tr>
<tr>
<td>sqrt</td>
<td>p34</td>
<td>statemate</td>
</tr>
<tr>
<td>ucl</td>
<td>p37</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Cache configurations

<table>
<thead>
<tr>
<th>(a, b, c) ID</th>
<th>(a, b, c) ID</th>
<th>(a, b, c) ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 16, 256) k1</td>
<td>(2, 16, 256) k2</td>
<td>(4, 16, 256) k3</td>
</tr>
<tr>
<td>(1, 32, 256) k4</td>
<td>(2, 32, 256) k5</td>
<td>(4, 32, 256) k6</td>
</tr>
<tr>
<td>(1, 16, 512) k7</td>
<td>(2, 16, 512) k8</td>
<td>(4, 16, 512) k9</td>
</tr>
<tr>
<td>(1, 32, 512) k10</td>
<td>(2, 32, 512) k11</td>
<td>(4, 32, 512) k12</td>
</tr>
<tr>
<td>(1, 16, 1024) k13</td>
<td>(2, 16, 1024) k14</td>
<td>(4, 16, 1024) k15</td>
</tr>
<tr>
<td>(1, 32, 1024) k16</td>
<td>(2, 32, 1024) k17</td>
<td>(4, 32, 1024) k18</td>
</tr>
<tr>
<td>(1, 16, 2048) k19</td>
<td>(2, 16, 2048) k20</td>
<td>(4, 16, 2048) k21</td>
</tr>
<tr>
<td>(1, 32, 2048) k22</td>
<td>(2, 32, 2048) k23</td>
<td>(4, 32, 2048) k24</td>
</tr>
<tr>
<td>(1, 16, 4096) k25</td>
<td>(2, 16, 4096) k26</td>
<td>(4, 16, 4096) k27</td>
</tr>
<tr>
<td>(1, 32, 4096) k28</td>
<td>(2, 32, 4096) k29</td>
<td>(4, 32, 4096) k30</td>
</tr>
<tr>
<td>(1, 16, 8192) k31</td>
<td>(2, 16, 8192) k32</td>
<td>(4, 16, 8192) k33</td>
</tr>
<tr>
<td>(1, 32, 8192) k34</td>
<td>(2, 32, 8192) k35</td>
<td>(4, 32, 8192) k36</td>
</tr>
</tbody>
</table>

To evaluate the improvement in energy efficiency (ACET and energy consumption), we used the inequations:

\[
\frac{1}{|\{e_o^{p,k,t}\}|} \sum_{p,k,t} e_o(e_{p,k,t}) e_a(e_p) < 1 \quad (10)
\]

\[
\frac{1}{|\{e_o^{p,k,t}\}|} \sum_{p,k,t} \tau_o(e_{p,k,t}) \tau_a(e_p) \leq 1 \quad (11)
\]

To evaluate the reduction in the WCET, we used the inequation:

\[
\forall(p,k,t) : \frac{\tau_w(e_{p,k,t})}{\tau_w(e_p)} < 1 \quad (12)
\]

First, such inequations were used to check all use cases individually. Then, by restricting them to a given cache size and taking the average, we obtained the values plotted in the Figures 3 and 5. Inequation 12 was also used to plot Figure 7.

S.5. EXTRA RESULTS

Figure 7 illustrates the WCET reduction for every use-case scenario, according to Inequation 12, when targeting 32nm.

Figure 8 plots the average ratio between the number of executed instructions of the optimized program as compared to the original one. It shows a maximal increase of 1.32%.