UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM
ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

EMILIO WUERGES

WCET-AWARE PREFETCHING OF UNLOCKED INSTRUCTION CACHES: A TECHNIQUE FOR RECONCILING REAL-TIME GUARANTEES AND ENERGY EFFICIENCY

Florianópolis, 2015

EMILIO WUERGES

WCET-AWARE PREFETCHING OF UNLOCKED INSTRUCTION CACHES: A TECHNIQUE FOR RECONCILING REAL-TIME GUARANTEES AND ENERGY EFFICIENCY

Tese de doutorado apresentada à Banca Examinadora do Programa de Pós-Graduação em Engenharia de Automação e Sistemas do Centro Tecnológico da Universidade Federal de Santa Catarina, como requisito parcial para a obtenção do título de Doutor em Engenharia de Automação e Sistemas, sob a orientação do Professor Doutor Luiz C. V. dos Santos e coorientação do Professor Doutor Rômulo Silva de Oliveira.

Florianópolis, 2015

EMILIO WUERGES

WCET-AWARE PREFETCHING OF UNLOCKED INSTRUCTION CACHES: A TECHNIQUE FOR RECONCILING REAL-TIME GUARANTEES AND ENERGY EFFICIENCY

Tese de doutorado apresentada ao Programa de Pós-Graduação em Engenharia de Automação e Sistemas do Centro Tecnológico da Universidade Federal de Santa Catarina em cumprimento a requisito parcial para a obtenção do título de Doutor em Engenharia de Automação e Sistemas.

APROVADA PELA COMISSÃO EXAMINADORA

EM FLORIANÓPOLIS, 26/02/2015

_____

Prof. Rômulo Silva de Oliveira, Dr. (Coordenador do Curso)

_____

Prof. Luiz C. V. dos Santos, Dr. – INE/UFSC (Orientador)

_____

Prof. Rômulo Silva de Oliveira, Dr. – DAS/UFSC (Coorientador)

_____

Prof. Flávio Rech Wagner, Dr. – Instituto de Informática/UFRGS

_____

Prof. Sandro Rigo, Dr. – Instituto de Computação/UNICAMP

_____

Profª Patricia Della Méa Plentz, Drª. – INE/UFSC

_____

Prof. Carlos Barros Montez, Dr. – DAS/UFSC

_____

Prof. Laércio Lima Pilla, Dr. – INE/UFSC

Wuerges, Emílio
WCET-AWARE PREFETCHING OF UNLOCKED INSTRUCTION CACHES : A TECHNIQUE FOR RECONCILING REAL-TIME GUARANTEES AND ENERGY EFFICIENCY / Emílio Wuerges ; orientador, Luiz C. V. dos Santos ; coorientador, Rômulo Silva de Oliveira. - Florianópolis, SC, 2015.
89 p.

Tese (doutorado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Engenharia de Automação e Sistemas.

Inclui referências

1. Engenharia de Automação e Sistemas. 2. Prefetching. 3. Compiler. 4. Energy Efficiency. 5. Real-time. I. Santos, Luiz C. V. dos . II. Oliveira, Rômulo Silva de. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia de Automação e Sistemas. IV. Título.

ACKNOWLEDGMENTS

This work was partially funded by the National Program of Microelectronics (PNM) of CNPq, by a scholarship (Process n. 141732/2010-5). Part of the costs of this work used resources from INCT NAMITEC: National Institute of Science and Technology of Micro and Nano-electronic Systems (Process CNPq. 573738/2008-4).

RESUMO

A computação embarcada requer crescente vazão sob baixa potência. Ela requer um aumento de eficiência energética quando se executam programas de crescente complexidade. Muitos sistemas embarcados são também sistemas de tempo real, cuja correção temporal precisa ser garantida através de análise de escalonabilidade, a qual costuma assumir que o WCET de uma tarefa é conhecido em tempo de projeto. Como resultado da crescente complexidade do software, uma quantidade significativa de energia é gasta ao se prover instruções através da hierarquia de memória. Como a cache de instruções consome cerca de 40% da energia gasta em um processador embarcado e afeta a energia consumida em memória principal, ela se torna um relevante alvo para otimização. Entretanto, como ela afeta substancialmente o WCET, o comportamento da cache precisa ser restrito via "cache locking" ou previsto via análise de WCET.

Para obter eficiência energética sob restrições de tempo real, é preciso estender a consciência que o compilador tem da plataforma de hardware. Entretanto, compiladores para tempo real ignoram a energia, embora determinem rapidamente limites superiores para o WCET, enquanto compiladores para sistemas embarcados estimem com precisão a energia, mas gastem muito tempo em "profiling". Por isso, esta tese propõe um método unificado para estimar a energia gasta em memória, o qual é baseado em Interpretação Abstrata, exatamente o mesmo substrato matemático usado para a análise de WCET em caches. As estimativas mostram derivadas que são tão precisas quanto as obtidas via "profiling", mas são computadas 1000 vezes mais rápido, sendo apropriadas para induzir otimização de código através de melhoria iterativa.

Como "cache locking" troca eficiência energética por previsibilidade, esta tese propõe uma nova otimização de código, baseada em pré-carga por software, a qual reduz a taxa de faltas de caches de instruções e, provadamente, não aumenta o WCET. A otimização proposta é comparada com o estado-da-arte em "cache locking" parcial para 37

programas do “Malardalen WCET benchmark” para 36 configurações de cache e duas tecnologias distintas (2664 casos de uso). Em média, para obter uma melhoria de 68% no WCET, “cache locking” parcial requer 8% mais energia. Por outro lado, a pré-carga por software diminui o consumo de energia em 11% enquanto melhora em 15% o WCET, reconciliando assim eficiência energética e garantias de tempo real.

ABSTRACT

Embedded computing requires increasing throughput at low power budgets. It asks for growing energy efficiency when executing programs of rising complexity. Many embedded systems are also real-time systems, whose temporal correctness is asserted through schedulability analysis, which often assumes that the WCET of each task is known at design-time. As a result of the growing software complexity, a significant amount of energy is spent in supplying instructions through the memory hierarchy. Since an instruction cache consumes around 40% of an embedded processor's energy and affects the energy spent in main memory, it becomes a relevant optimization target. However, since it largely impacts the WCET, cache behavior must be either constrained via cache locking or predicted by WCET analysis.

To achieve energy efficiency under real-time constraints, a compiler must have extended awareness of the hardware platform. However, real-time compilers ignore energy, although they quickly determine bounds for WCET, whereas embedded compilers accurately estimate energy but require time-consuming profiling. That is why this thesis proposes a unifying method to estimate memory energy consumption that is based on Abstract Interpretation, the very same mathematical framework employed for the WCET analysis of caches. The estimates exhibit derivatives that are as accurate as those obtained by profiling, but are computed 1000 times faster, being suitable for driving code optimization through iterative improvement.

Since cache locking gives up energy efficiency for predictability, this thesis proposes a novel code optimization, based on software prefetching, which reduces miss rate of unlocked instruction caches and, provenly, does not increase the WCET. The proposed optimization is compared with a state-of-the-art partial cache locking technique for the 37 programs of the Malardalen WCET benchmarks under 36 cache configurations and two distinct target technologies (2664 use cases). On average, to achieve an improvement of 68% in the WCET, partial cache locking required 8% more energy. On the other hand, software

prefetching decreased the energy consumption by 11% while leading to an improvement of 15% in the WCET, thereby reconciling energy efficiency and real-time guarantees.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

RTOS	Real-time operating system
ALU	Arithmetic logic unit
I-cache	Instruction cache
D-cache	Data cache
R-file	Register file
ACET	Average case execution time
WCET	Worst-case execution time
LRU	Least recently used
CFG	Control flow graph
AI	Abstract interpretation
BCET	Best-case execution time
ACEC	Energy consumption of the average-case execution time
SoC	System on Chip
ILP	Integer linear programming
IPET	Implicit path enumeration technique
IR	Intermediate representation
RPT	Reference prediction table
FCL	Full cache locking
PCL	Partial cache locking
WCEP	Worst-case execution path
ACFG	Abstract control flow graph

MRU	Most recently used
VIVU	Virtual interpretation of virtual unrolled (loops)
PFU	Prefetching unit

LIST OF SYMBOLS

I	An invalid cache block
r_i	An address of an instruction
s_i	A block in memory
π_{s_i}	A prefetch for the block s_i
$p \in P$	A program p and the set of programs P
$c \in C$	A memory configuration c and the set of memory configurations C
$c = (n, s)$	A memory configuration c, associativity s and cache size n
$t \in T$	A process technology t and the set of technologies T
a	Average-case execution scenario
w	Worst-case execution scenario
b	Best-case execution scenario
τ	A time estimate
ε	An estimate for the energy consumption
ν	Execution time normalized to the average-case execution time
η	Energy consumption normalized to the average-case energy consumption
T_i	The set of time estimates under an execution time scenario i
E_i	The set of energy estimates under an execution time scenario i

ρ	Pearson's coefficient
τ^+	Correlation between worst-case execution time estimate with average case estimates
τ^-	Correlation between best-case execution time estimate with average case estimates
ε^+	Correlation between worst-case execution energy estimate with average case estimates
ε^-	Correlation between best-case execution energy estimate with average case estimates
L	The set of cache lines
S	The set of memory blocks
$\mathscr{U}$	A cache update function
$\hat{C}$	The set of abstract cache states
B	The set of basic blocks
bb_i	A basic block of a program
n_{bb_i}	The number of times a basic block is executed
$t_i^p(r)$	The execution time of the program p under the scenario i
π_s	A prefetch to the block s
$\mathscr{J}$	A join function
$\mathscr{B}(\hat{c})$	The set of blocks in a cache state $\hat{c}$

1 INTRODUCTION

1.1 TRENDS IN EMBEDDED COMPUTING

Various embedded applications demand increasing energy efficiency, because they combine high throughput requirements with power constraints, ranging from low power control (for instance, hard driver controllers and automotive control units) to high speed and low power communication devices (for instance, baseband processing in wireless modems and mobile devices).

Many embedded systems are also real-time systems in the sense that they must meet real-time requirements. Results must be not only logically correct but they also must be produced at the right moment. The temporal correctness of a system is asserted through its schedulability analysis. Most schedulability analyses are based on the assumption that the worst-case execution time (WCET) of each task is known at design-time.

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 multi-threading 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 a Real-Time Operating System (RTOS). Baseband processing, for instance, involves increasingly energy efficiency, real-time constraints and growing software complexity. According to Dally et al. (DALLY et al., 2008), for a power constraint of around 1W, the energy efficiency had to increase from 25pJ/operations (for a 3G receiver) to 3-5pJ/operation (for a 4G receiver). This is an example of a scenario asking for techniques that do not jeopardize predictability when improving energy efficiency.

Figure 1 (DALLY et al., 2008) shows that around 70% of the energy spent in an embedded processor is due to data and instruction

supply (D-supply and I-supply). Figure 2 shows the contribution of the main hardware components to energy consumption[1]. Note that, while the consumption of pipeline registers (R-pipe) is marginal, the on-chip memories (I-cache and D-cache) and the register file (R-file) are responsible for more than 60% of the total energy consumption of the processor: 39% of the energy is spent in the instruction cache, 14% in data cache, and 11% in the register file. Since the contribution of the arithmetic-logical unit (ALU) is marginal (6%), code optimizations that reduce the energy spent in arithmetic operations have little impact on energy efficiency. The same reasoning applies to the pipeline registers. Although, the energy spent in control logic (clock+control) is significant, it could hardly be reduced through embedded code optimization. Fortunately, the energy spent in storage components (I-cache, D-cache, and R-file) can be shrinked with the help of code optimizations (e.g. register allocation, basic-block placement, procedure sorting, etc.). The major energy consumer is the instruction cache. Part of the energy represented by the shaded area in Figure 2 corresponds to dynamic consumption, which is proportional to the number of accesses to the instruction cache. Another part corresponds to static consumption, which is mainly due to leakage.

Figure 2, however, does not account for the energy spent in lower levels of the memory hierarchy. For instance, part of such energy corresponds to the dynamic consumption in main memory, which is proportional to the global miss rate in higher hierarchical levels. Another part corresponds to static consumption, which is proportional to the execution time and to the static power consumption in main memory.

1.2 MOTIVATION

Given a task of a real-time system, code optimizations should reduce the energy spent in supplying instructions through the whole memory hierarchy, while preserving real-time guarantees, but without

[1] The values reported in Figure 2 were obtained by combining the data available in Figures 1, 2, and 3 from (DALLY et al., 2008).

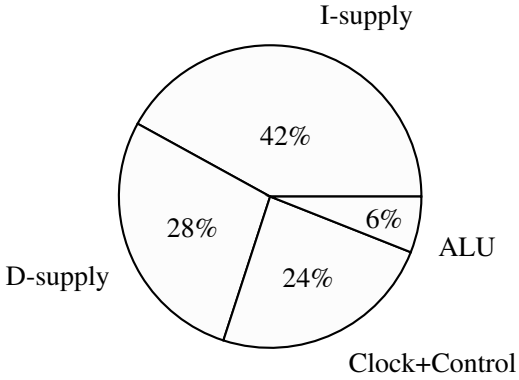


Figure 1 – Typical processor energy consumption (DALLY et al., 2008)

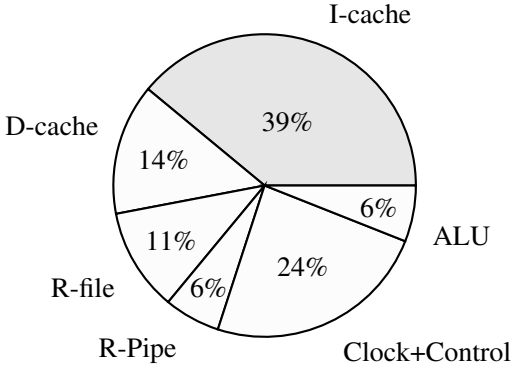


Figure 2 – Detailed energy consumption of a processor

decreasing the average throughput. Therefore, for energy-efficient instruction supply, code optimizations should:

- **Reduce the dynamic consumption in the instruction cache:** this can only be done by decreasing the number of executed instructions. Fortunately, this is accomplished by common optimizations available in conventional compilers (e.g. dead code elimination, constant/copy propagation, common subexpression elimination, code motion of the loop invariant).

- **Reduce the dynamic consumption in the main memory:** this

can be obtained by decreasing the miss rate. Although some optimizations address that goal in conventional compilers (e.g. basic block placement, procedure sorting), they do not provide real-time guarantees.

- **Reduce the static consumption in both the instruction cache and main memory:** this can be done by reducing the miss rate (since this decreases the average execution time of a task).

Since an instruction cache may consume around 40% of an embedded processor's energy (DALLY et al., 2008), since it affects the energy consumption in main memory, and since 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), instruction prefetching is underexploited in real-time systems, although a solid basis for accurately predicting cache behavior (FERDINAND et al., 1999; THEILING; FERDINAND; WILHELM, 2000) has been laid. One work extended cache abstract semantics to take hardware prefetching into account (YAN; ZHANG, 2007), another exploited it for optimizing the WCET (DING; YAN; ZHANG, 2009). On the other hand, *cache locking* (PUAUT; ARNAUD, 2006) (DING; LIANG; MITRA, 2012) increases the predictability of cache-based real-time systems. Although it allows a large reduction in the WCET, no concern is shown for the impact on energy consumption. Despite the proposition of a recent technique combining hardware prefetching with cache locking (APARICIO et al., 2010), the joint impact on energy ef-

ficiency was not evaluated. Besides, the use of software prefetching for real-time systems is even less exploited than hardware prefetching.

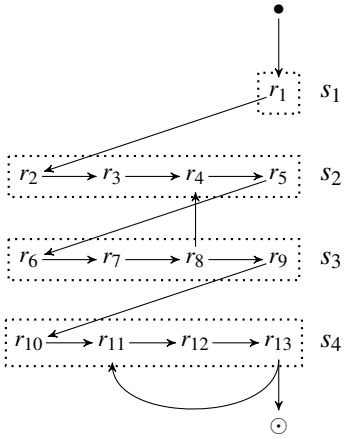
For these reasons this thesis focuses on WCET-aware software prefetching for instruction caches and compares it with cache locking in terms of WCET, ACET, and energy consumption.

1.3 ILLUSTRATIVE EXAMPLES

This section employs a chain of simple examples to illustrate the basic mechanisms of cache locking and software prefetching when applied to instruction caches. It allows us to clearly pinpoint their sources of improvements and limitations, which will be the keys to the analysis of related work (Chapter 3) and to the interpretation of the experimental results (Chapter 6).

In the examples, a given program is run on the same cache configuration under four distinct scenarios: unlocked cache with on-demand fetching, full cache locking, partial cache locking, and unlocked cache with software prefetch. After showing the resulting behavior of the cache in each scenario, we analyze the impact of each technique on miss rate (which affects energy consumption and ACET) and on predictability (which affects the accuracy of an upper bound for the WCET). Although in real-life scenarios, the cache state space may be too large to explore (WILHELM et al., 2008), for our examples, we assume that WCET analysis is able to determine every cache state. For a simple estimation of the WCET, we suppose that the access time of a cache is one cycle, and the access time of main memory is 2 cycles.

For simplicity, all the examples assume a 2-way set associative cache with only two blocks (note that this is equivalent to assuming that we are observing one of the sets of a larger cache, but all the references in the example map to that same set). We adopt the least recently used (LRU) replacement policy and denote cache states as $[X,Y]$, where X and Y represent the most and the least recently used memory blocks residing in cache, respectively. We use I to denote an invalid block. Therefore, $[I,I]$ represents the initial state.



(a) CFG

First reference			Subsequent references		
State		Result	State		Result
$[I,I]$	r_1	*Miss*			
$[s_1,I]$	r_2	*Miss*			
$[s_2,s_1]$	r_3	*Hit*			
$[s_2,s_1]$	r_4	*Hit*	$[s_3,s_2]$	r_4	*Hit*
$[s_2,s_1]$	r_5	*Hit*	$[s_2,s_3]$	r_5	*Hit*
$[s_2,s_1]$	r_6	*Miss*	$[s_2,s_3]$	r_6	*Hit*
$[s_3,s_2]$	r_7	*Hit*	$[s_3,s_2]$	r_7	*Hit*
$[s_3,s_2]$	r_8	*Hit*	$[s_3,s_2]$	r_8	*Hit*
$[s_3,s_2]$	r_9	*Hit*			
$[s_3,s_2]$	r_{10}	*Miss*			
$[s_4,s_3]$	r_{11}	*Hit*	$[s_4,s_3]$	r_{11}	*Hit*
$[s_4,s_3]$	r_{12}	*Hit*	$[s_4,s_3]$	r_{12}	*Hit*
$[s_4,s_3]$	r_{13}	*Hit*	$[s_4,s_3]$	r_{13}	*Hit*

(b) Cache states

Figure 3 – Behavior of an unlocked cache

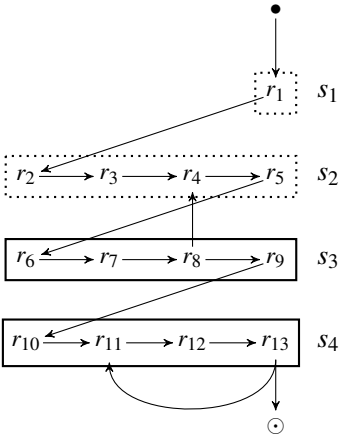
Figure 3a shows a control-flow graph (CFG) representation for the program, where edges denote the flow of control and vertices represent references to memory locations. Each reference r_i represents the address of an instruction. Each dotted rectangle represents the boundaries of a block s_j in memory. The example assumes that each block contains four instructions. For instance, r_2, r_3, r_4, and r_5 refer to instructions belonging to memory block s_2. Note that the program has two loops, which are indicated by the back edges (r_8, r_4) and (r_{13}, r_{11}).

For all the examples in this section, we suppose that each loop executes exactly ten iterations.

Figure 3b shows the resulting cache behavior when conventional on-demand fetching is used to update cache blocks. The table has two partitions, one showing the behavior induced by the references in the first iteration of each loop, another showing the behavior for the subsequent iterations. Each line represents the cache state *before* a given reference and the outcome *after* that reference.

Since the cache is initially empty, r_1 induces a miss. As a result, block s_1 is loaded into the cache. Similarly, r_2 also induces a miss and causes the loading of s_2, which becomes the most recently used block in cache. Since the state of the cache is $[s_2, s_1]$ when r_3, r_4, and r_5 are successively reached, they all lead to hits. Then r_6 induces a miss and block s_3 is loaded. As a result, the next two references, r_7 and r_8, lead to hits, closing the first iteration of the first loop. For the subsequent iterations of that loop, r_4 to r_8 induce hits, since they reference blocks s_3 and s_2, which are not replaced. For the same reason, on exit of that loop, r_9 also leads to a hit. Then r_{10} leads to a miss and block s_4 is loaded. As r_{11}, r_{12}, and r_{13} all refer to the same block s_4, they all hit, closing the first iteration of the second loop. For the subsequent iterations, r_{11} to r_{13} induce hits, since they refer to block s_4, which is not replaced. Note that r_6 has induced hits for all but the first iteration of the first loop, because it references the first item of a block. In contrast, r_{11} has always induced hits, because it does not reference the first item. In this scenario, the program results in 4 misses and 89 hits, i.e. a miss rate of approximately 4%. This example reviews the fact that the effectiveness of on-demand fetching strongly relies on the temporal locality induced by loop iterations and the spatial locality induced by references to successive addresses within a block. The WCET for this scenario is 97 cycles.

Figure 4 illustrates the scenario for a fully locked cache. In this example, blocks s_3 and s_4 are locked in the cache. Such locking is indicated in the CFG by the solid rectangles. Since as a result of full locking, no block will ever be replaced, the prediction of memory



(a) CFG

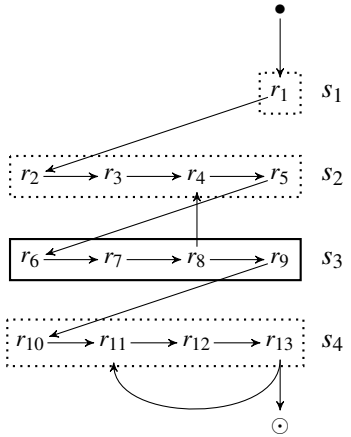
First reference			Subsequent references		
State		Result	State		Result
$[s_3, s_4]$	r_1	*Miss*			
$[s_3, s_4]$	r_2	*Miss*			
$[s_3, s_4]$	r_3	*Miss*			
$[s_3, s_4]$	r_4	*Miss*	$[s_3, s_4]$	r_4	*Miss*
$[s_3, s_4]$	r_5	*Miss*	$[s_3, s_4]$	r_5	*Miss*
$[s_3, s_4]$	r_6	*Hit*	$[s_3, s_4]$	r_6	*Hit*
$[s_3, s_4]$	r_7	*Hit*	$[s_3, s_4]$	r_7	*Hit*
$[s_3, s_4]$	r_8	*Hit*	$[s_3, s_4]$	r_8	*Hit*
$[s_3, s_4]$	r_9	*Hit*			
$[s_3, s_4]$	r_{10}	*Hit*			
$[s_3, s_4]$	r_{11}	*Hit*	$[s_3, s_4]$	r_{11}	*Hit*
$[s_3, s_4]$	r_{12}	*Hit*	$[s_3, s_4]$	r_{12}	*Hit*
$[s_3, s_4]$	r_{13}	*Hit*	$[s_3, s_4]$	r_{13}	*Hit*

(b) Cache states

Figure 4 – Fully locked cache

behavior is trivial. Since there are only two blocks and both are locked, only the references to instructions belonging to blocks s_3 and s_4 will lead to hits; all the others result in misses. In this scenario, the program induces 25 misses and 68 hits, i.e. a miss rate of around 27%. As compared to the previous scenario, despite the higher predictability, the energy consumption and the ACET are largely increased, as a result of a larger miss rate. The WCET for this scenario is 118 cycles.

Figure 5 illustrates the behavior of a cache where only block s_3



(a) CFG

First reference			Subsequent references		
State		Result	State		Result
$[s_3, I]$	r_1	*Miss*			
$[s_3, s_1]$	r_2	*Miss*			
$[s_3, s_2]$	r_3	*Hit*			
$[s_3, s_2]$	r_4	*Hit*	$[s_3, s_2]$	r_4	*Hit*
$[s_3, s_2]$	r_5	*Hit*	$[s_3, s_2]$	r_5	*Hit*
$[s_3, s_2]$	r_6	*Hit*	$[s_3, s_2]$	r_6	*Hit*
$[s_3, s_2]$	r_7	*Hit*	$[s_3, s_2]$	r_7	*Hit*
$[s_3, s_2]$	r_8	*Hit*	$[s_3, s_2]$	r_8	*Hit*
$[s_3, s_2]$	r_9	*Hit*			
$[s_3, s_2]$	r_{10}	*Miss*			
$[s_3, s_4]$	r_{11}	*Hit*	$[s_3, s_4]$	r_{11}	*Hit*
$[s_3, s_4]$	r_{12}	*Hit*	$[s_3, s_4]$	r_{12}	*Hit*
$[s_3, s_4]$	r_{13}	*Hit*	$[s_3, s_4]$	r_{13}	*Hit*

(b) Cache states

Figure 5 – Partially locked cache

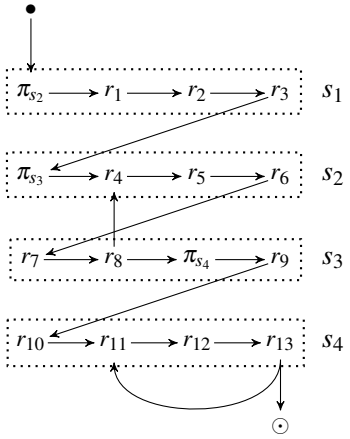
is locked, as denoted by the solid rectangle in the CFG. To indicate such partial locking in cache state, we show the locked block as if permanently stuck at the same position. Note that, since one of two blocks is locked, the 2-way set associative cache essentially degenerates into a direct-mapped cache. Although, there is no need to predict cache behavior for references to instructions belonging to block s_3 (which will always hit), prediction is required for all the other references.

Although a valid block is in cache when reference r_1 is reached,

a miss is induced because the locked block is not s_1. As a result, s_1 is loaded into the single unlocked block. That is why the cache state is $[s_3, s_1]$ when r_2 is reached, leading to a miss. Since s_3 is locked, s_2 replaces s_1 in cache. As a result, r_3, r_4, and r_5 lead to hits. Since s_3 was locked in cache, a hit is induced when r_6 is reached for the first time (as opposed to what happened for the unlocked cache). In this scenario, the program induces 3 misses and 90 hits, i.e. a miss rate of approximately 3%, which is smaller than the one obtained for the unlocked cache and much smaller than the one observed under full cache locking. This illustrates the fact that partial cache locking tends to reduce the energy consumption and the ACET as compared to full locking. The WCET for this scenario is 96 cycles. Indeed, the literature shows that partial cache locking outperforms full cache locking both in terms of ACET and WCET (PUAUT; ARNAUD, 2006) (DING; LIANG; MITRA, 2012).

Figure 6 shows an optimized version of the program used in the previous examples, where prefetch instructions were inserted. In that figure, π_{s_2}, π_{s_3}, and π_{s_4} are references to prefetch instructions that load blocks s_2, s_3, and s_4, respectively. Note that, when a prefetch instruction is inserted, it may displace references that are located before the prefetch instruction to previous blocks. For instance, r_6 was moved from block s_3 (in the previous scenarios) to the block s_2 (in the current scenario). For simplicity, we assume that the latency of the prefetch instructions is zero.

Since π_{s_2} is the first reference to block s_1, it leads to a miss. As a consequence, two blocks are loaded into the cache: one as a result of on-demand fetching (s_1), another due to the first prefetch instruction (s_2). Thus, when r_1 is reached, the cache state is $[s_1, s_2]$, resulting in a hit. For the same reason, r_2 and r_3 also lead to hits. Despite being a reference to the first instruction of a block, π_{s_3} leads to a hit. This illustrates that, although the reference to a prefetch instruction may raise a miss (e.g. π_{s_2}), another may result in a hit (e.g. π_{s_3}) if it lies in a block that was prefetched by a former prefetch instruction. Therefore, the overhead of inserting a prefetch instruction can be eliminated by properly inserting another prefetch instruction before it.



(a) CFG

First reference			Subsequent references		
State		Result	State		Result
$[I,I]$	π_{s_2}	*Miss*			
$[s_1,s_2]$	r_1	*Hit*			
$[s_1,s_2]$	r_2	*Hit*			
$[s_1,s_2]$	r_3	*Hit*			
$[s_1,s_2]$	π_{s_3}	*Hit*			
$[s_2,s_3]$	r_4	*Hit*	$[s_3,s_2]$	r_4	*Hit*
$[s_2,s_3]$	r_5	*Hit*	$[s_2,s_3]$	r_5	*Hit*
$[s_2,s_3]$	r_6	*Hit*	$[s_2,s_3]$	r_6	*Hit*
$[s_2,s_3]$	r_7	*Hit*	$[s_2,s_3]$	r_7	*Hit*
$[s_3,s_2]$	r_8	*Hit*	$[s_3,s_2]$	r_8	*Hit*
$[s_3,s_2]$	π_{s_4}	*Hit*			
$[s_3,s_4]$	r_9	*Hit*			
$[s_3,s_4]$	r_{10}	*Hit*			
$[s_4,s_3]$	r_{11}	*Hit*	$[s_4,s_3]$	r_{11}	*Hit*
$[s_4,s_3]$	r_{12}	*Hit*	$[s_4,s_3]$	r_{12}	*Hit*
$[s_4,s_3]$	r_{13}	*Hit*	$[s_4,s_3]$	r_{13}	*Hit*

(b) Cache states

Figure 6 – Unlocked cache with prefetching

When prefetch instructions lie in prefetched blocks or when they are not the first reference of a block fetched on demand, their contribution to the execution time narrows down to the time of a hit. Since the program was optimized so that a block was loaded in cache before it was referenced, all references except the first one induce hits. In this scenario, the optimized program induces a single miss and 95 hits, i.e.

a miss rate of 1%, which is the smallest value among all four scenarios.

The example illustrates that, as compared to partial cache locking, proper prefetching may lead to smaller ACET, since the processor is stalled less frequently.

The WCET for this scenario is 97 cycles. Since, for simplicity, we assumed that WCET analysis is able to explore every state of the cache, prefetching led to a similar value for WCET as compared to partial cache locking. As the unlocked cache state space is generally too large to explore (WILHELM et al., 2008) and since locking essentially degenerates the cache into a simpler one, partial cache locking typically leads to a much smaller WCET, as will be shown in Section 6.2.

The examples illustrate that, when targeting energy-efficient real-time systems, the key is to reduce not only the WCET, but also the miss rate, for two reasons. First, a reduction in miss rate decreases the number of accesses to main memory. Second, since it reduces the ACET, it also decreases static consumption. Although a reduction in miss rate only decreases dynamic consumption in main memory, it decreases the static consumption in both, main memory and cache.

1.4 CONTRIBUTIONS AND PUBLICATIONS

1.4.1 A fast energy-aware estimation technique

Although average-case assessment is time consuming, best-case and worst-case scenarios can be identified with less computational effort: they can rely on the *Abstract Interpretation* (AI) (COUSOT; COUSOT, 1977) of a program, instead of executing it. In the field of Real-Time Systems, AI is employed to find tight bounds for *worst-case* and *best-case* execution times (WCET and BCET)

This thesis proposes a unified technique to estimate the ACET and average-case energy consumption (ACEC) from the time and energy computed for the worst-case and the best-case execution scenarios (WCET and BCET). Instead of employing it to provide execution time bounds for real-time scheduling, our technique uses Abstract Interpre-

tation at compile time to optimize the average-case energy efficiency of the memory subsystem. The key idea lies in the fact that compilers often drive optimization based on iterative improvement. As a consequence, the accuracy of the cost function is less important than its derivative's, since the variation in cost will be the actual driver for decision making.

1.4.2 A new optimization technique

This thesis proposes a novel technique that inserts prefetch instructions for improving the energy efficiency of instruction caches. In contrast with most real-time optimization techniques (DING; YAN; ZHANG, 2009; DING; LIANG; MITRA, 2012; PLAZAR et al., 2012), 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 (Appendix A). 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.

1.4.3 Energy consumption evaluation of two real-time techniques

This thesis compares the proposed optimization technique, which relies on prefetching of unlocked instruction caches (WUERGES; OLIVEIRA; SANTOS, 2013), with a state-of-the-art technique suitable for real-time systems, which employs instruction cache locking (DING; LIANG; MITRA, 2012). The main contribution is a direct comparison in terms of their impact on worst-case execution time (WCET), average-case execution time (ACET), and energy consumption. To our knowledge, this is the first time two techniques suitable for real-time

systems are compared both in terms of the energy efficiency and the predictability of the memory subsystem.

1.4.4 Scope of application

The techniques proposed in this thesis suppose the access to the memory hierarchy from the perspective of a single processor, which might be one among multiple cores of a system-on-chip (SoC). As opposed to applications targeting the end-user, which generally rely on multiple threads allocated to distinct cores, real-time applications generally assume *multi-tasking* on a single core (often relying on a more deterministic architecture than those used for end-user applications). To be performed for a task running on a same processor along with other tasks, the proposed optimization technique should be applied in the scope determined by two successive preemption points predefined in the code of the task, as explained in Section 7.2.

1.4.5 Publications

The description of the proposed estimation technique (reported in Chapter 2) was published in the proceedings of the IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2011) (WUERGES; OLIVEIRA; SANTOS, 2011). The description of the proposed optimization technique (reported in Chapter 5) and its experimental evaluation (reported in Chapter 6) were published in the proceedings of the 50th IEEE/ACM Design Automation Conference (DAC 2013) (WUERGES; OLIVEIRA; SANTOS, 2013). The experimental comparison of the proposed technique with partial cache locking (also reported in Chapter 5) is part of an article submitted for publication in the Springer Journal of System Architecture (JSA).

1.5 ORGANIZATION OF THIS THESIS

This thesis is structured as follows. The next chapter describes the main fundamental concepts and techniques supporting the proposed

optimization technique. Particularly, it shows preliminary experimental evidence that the WCET scenario can be used to induce energy optimization through iterative improvement. Chapter 3 analyzes related work in prefetching and cache locking. Chapter 4 presents the notions required to model the target optimization problem. Chapter 5 presents the proposed WCET-aware instruction prefetching technique. Chapter 6 experimentally evaluates the impact of the proposed optimization with respect to on-demand fetching and directly compares the results with a state-of-the-art partial cache locking technique. Chapter 7 summarizes our concluding remarks and perspectives for future work. Appendix A provides the formal WCET guarantees.

2 FAST ENERGY ESTIMATION

This chapter describes a novel technique that makes unconventional use of Abstract Interpretation for fast energy estimation. First, it summarizes the main methods used to track execution time and energy consumption in conventional, embedded, and real-time compilers (Section 2.1). Then it shows how a compiler's workflow can be adapted for fast energy-aware estimations (Section 2.2). Finally, it shows experimental evidence that the derivatives of our estimates are as accurate as those obtained from trace-based approaches, but can be computed at least 1000 times faster, being suitable for driving embedded code optimizations that employ iterative improvement.

2.1 MAIN METHODS FOR ESTIMATION

The identification of worst and best-case execution scenarios requires loop analysis (to determine the maximal and minimal number of iterations), path analysis (to find the most and the least critical execution paths), and abstract interpretation (to set upper and lower bounds on program outcome).

Loop analysis techniques are well-known and can be found in classic compiler textbooks (MUCHNICK, 1997). An efficient and pragmatic way of performing path analysis is to rely on the so-called Implicit Path Enumeration Technique (IPET) (LI; MALIK; WOLFE, 1995). IPET represents paths as constraints of an integer linear programming (ILP) problem where either WCET or BCET can be adopted as cost function.

Abstract Interpretation (AI) (COUSOT; COUSOT, 1977) is a theory that relies on an abstract semantics for static program analysis. AI efficiently computes properties of a program without actually executing it. The abstract semantics precludes, for instance, the iterative execution of loop bodies, which largely contributes to speeding up program analysis. The AI theory can be applied to determine upper and lower bounds for cache behavior. An accurate and efficient AI

semantics, known as must-may analysis (THEILING; FERDINAND; WILHELM, 2000), was proposed for reasoning about cache behavior.

For fast decision making, conventional compilers often rely on simple metrics (like instruction count) (MUCHNICK, 1997) to track performance, but are energy unaware. Many compiling techniques tailored to optimizing embedded software (VERMA; MARWEDEL, 2007; UDAYAKUMARAN; DOMINGUEZ; BARUA, 2006; CHEN et al., 2006) rely on trace-based estimation of ACET and ACEC. Albeit accurate, such estimation is rather inefficient: it needs program profiling to select a trace that approximates the average case and requires actual program execution to induce the selected trace's memory access pattern, which is the very key to estimation.

A couple of embedded compiling techniques (FALK, 2009; FALK; KLEINSORGE, 2009) proposed WCET as a suitable metric to guide code optimization for applications under real-time constraints, but they did not investigate the correlation between WCET and ACET nor addressed energy optimization.

The lack of fast ACET and ACEC estimation techniques is likely to hamper optimizing compilers in face of growing energy efficiency requirements. This motivated us to investigate time and energy correlations between the average-case and (best) worst-case execution scenarios and led us to extend AI and IPET to cope with energy, as shown in the next section.

2.2 THE PROPOSED ENERGY-AWARE WORKFLOW

Figure 7 shows how we extended a conventional workflow to build an energy-aware compiler. From the specified hardware parameters (a configuration c and a target technology t), a physical memory model supplies the time and the energy spent by each memory access. The compiler front-end translates the source code of a program p into an intermediate representation (IR) of the program, which is largely independent from language and target instruction set. The compiler back-end translates the IR into executable machine code.

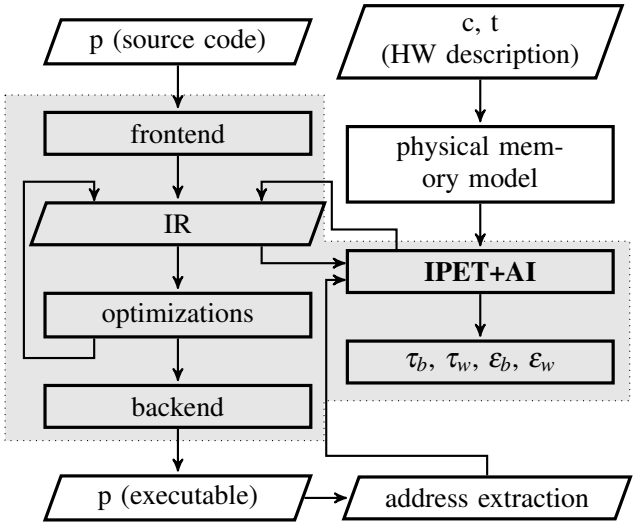


Figure 7 – Energy-aware workflow based on AI and IPET

Our time and energy analyzer, which runs in-between optimization passes at IR level, is build upon IPET (LI; MALIK; WOLFE, 1995) and AI (THEILING; FERDINAND; WILHELM, 2000). These techniques (originally proposed for execution time analysis), were extended to handle energy consumption.

At IR level, an access to memory is represented by a symbolic reference. However, memory behavior depends on effective addresses (which are defined at linking time). That is why an extractor gets the address corresponding to a memory reference from the executable file's symbol table.

Since it is performed at IR level, our estimation technique is independent from the target instruction set, except for address extraction. However, as our extractor was designed to handle fixed instruction-length architectures, it is compatible with most energy-efficient processors (e.g. ARM, MIPS, PowerPC).

Notice that, as a result of the analyzer's dependence on effective addresses, a program must be compiled twice to benefit from time and energy analysis. For the first run, the analyzer is turned off, but it is

activated for the second.

We implemented the IPET+AI module as a regular pass inside GCC (STALLMAN, 2010). To solve the ILP problem resulting from IPET, a popular ILP solver (BERKELAAR; EIKLAND; NOTEBAERT, 2004) was invoked. Our physical memory model relied on CACTI v6.5 (MURALIMANOHAR; BALASUBRAMONIAN; JOUPPI, 2007). All the other modules were reused from GCC's infrastructure (version 4.6.1).

Before presenting the experiments performed with the described workflow, the next section formulates the mathematical background for proper interpretation of experimental results.

2.3 EXPERIMENTAL EVIDENCE OF SUITABILITY FOR ITERATIVE IMPROVEMENT

2.3.1 A formulation to evaluate accuracy

We want to analyze the properties of a set P of programs running on a set C of distinct memory configurations which admit different implementations depending on a set T of target technologies. Let (p,c,t) represent a program $p \in P$ running on a memory configuration $c \in C$ that is implemented with technology $t \in T$. Let a, b, and w denote average, best, and worst-case execution time scenarios, respectively. Let $\tau_i(p,c,t)$ denote the *time estimate* under a given execution time scenario i, with $i \in \{a,b,w\}$, and let $\varepsilon_i(p,c,t)$ denote the *estimate for the energy consumed* when the program runs under execution time scenario i. From now on, we will informally refer to time and energy estimates for average, best, and worst-case scenarios. However, the reader should be aware that a scenario will always be defined by its *execution time*, even when we measure its energy consumption.

2.3.2 Normalization

To evaluate the accuracy of employing worst and best-case estimates as substitutes for average-case estimates, we track the ratio

between their values, as follows. Let $\nu_i(p,c,t)$ be the execution time when running a program p in scenario i normalized to the average-case execution time, i.e. $\nu_i(p,c,t) = \frac{\tau_i(p,c,t)}{\tau_a(p,c,t)}$. Similarly, let $\eta_i(p,c,t)$ be the energy consumption when running a program p in scenario i normalized to the average-case energy consumption, i.e. $\eta_i(p,c,t) = \frac{\varepsilon_i(p,c,t)}{\varepsilon_a(p,c,t)}$.

To obtain time and energy summaries for each program p running in execution scenario i, we take the geometric mean of $n = |C \times T|$ values, as follows:

$\nu_i(p) = \sqrt[n]{\prod_{\forall c,t} \nu_i(p,c,t)}$ and $\eta_i(p) = \sqrt[n]{\prod_{\forall c,t} \eta_i(p,c,t)}$.

As a result, we can say that the WCET is ν_w times higher than the ACET and that the BCET is ν_b times smaller than the ACET. Similar interpretations hold for energy consumption.

2.3.3 Correlation

Let us define the *set of time and energy estimates* in an execution scenario $i \in \{a,b,w\}$, for a given program p:

$T_i(p) = \{\forall (c,t) \in C \times T : \tau_i(p,c,t)\}$,

$E_i(p) = \{\forall (c,t) \in C \times T : \varepsilon_i(p,c,t)\}$.

We can also define sets of time and energy estimates for a given memory configuration c, as follows:

$T_i(c) = \{\forall (p,t) \in P \times T : \tau_i(p,c,t)\}$,

$E_i(c) = \{\forall (p,t) \in P \times T : \varepsilon_i(p,c,t)\}$.

To track the correlation between two sets of data, say X and Y, we adopted Pearson's coefficient $\rho(X,Y)$, i.e. the ratio between their covariance and the product of their standard deviations. We define two correlations per estimate, as follows.

The *upper* and the *lower* time and energy correlations of a given program p are, respectively:

$\tau^+(p) = \rho(T_a(p), T_w(p))$, $\tau^-(p) = \rho(T_a(p), T_b(p))$,

$\varepsilon^+(p) = \rho(E_a(p), E_w(p))$, $\varepsilon^-(p) = \rho(E_a(p), E_b(p))$.

It is straightforward to define similar correlations for a given configuration c, i.e. $\tau^+(c)$, $\tau^-(c)$, $\varepsilon^+(c)$, and $\varepsilon^-(c)$.

For a given configuration, since we monitor the magnitude *varia-*

tion when correlating sets of estimates (X and Y) from distinct program and technology choices, their full correlation ($\rho(X,Y) = 1$) means that their estimates have exactly the same derivatives with respect to program choice (for a given target technology); if uncorrelated ($\rho(X,Y) = 0$), one estimate is not a good substitute for the other. The same holds for a given program w.r.t. configuration change. That is why, we monitor correlations $\tau^+(p)$, $\tau^-(p)$, $\tau^+(c)$, $\tau^-(c)$, $\varepsilon^+(p)$, $\varepsilon^-(p)$, $\varepsilon^+(c)$, and $\varepsilon^-(c)$ in the next section. As average values are obtained by highly accurate trace-based estimation, the correlations whose values are close to one will indicate that the correlating worst-case or best-case estimate has the same derivative accuracy as the average-case estimate. In short, the former can be used as a (faster) substitute for the latter.

2.3.4 Experimental results

Programs were selected from the Mibench benchmark (GUTHAUS, 2001), were all compiled at GCC's O3 optimization level, and were targeted to ARM's Cortex instruction set. For simplicity, we limited the experiments to the estimation of execution time and energy consumption induced by instruction caches due to their higher consumption as compared to data caches (DALLY et al., 2008). We employed 21 distinct cache configurations with same line size (32 bytes). A configuration c is a pair $c = (n,s)$, where n is the associativity and s is the cache size expressed in KB. Configurations were selected such that $n \in \{1,2,4\}$ and $s \in \{2,4,8,16,32,64,128\}$. We targeted two CMOS technologies: 65nm and 40nm.

We reused GCC's automatic loop analysis. Since the accuracy of the estimates depends on the compiler's ability of determining loop bounds, we selected programs whose analysis succeeded for at least 40% of their loops (whenever it failed, we assigned loop bounds manually). For each selected program, Table 1 shows the total number of loops (NL) and the *loop yield* (LY), i.e. the fraction of all loops whose bounds were automatically determined.

To obtain highly accurate ACET and ACEC estimates, we first

Table 1 – Program characterization

Program	p	NL	LY	Size [kB]
basicmath_large	1	32	0.5	822
basicmath_small	2	32	0.5	822
bitcnts	3	2	0.5	608
qsort_large	4	4	0.5	615
qsort_small	5	4	0.5	596
rijndael	6	15	0.4	637
sha	7	26	0.84	600

generated a trace for each program p from typical stimuli supplied within the benchmark suite and we run each program for the generated trace for every memory specification (c,t). The monitored values were employed as time and energy estimates ($\tau_a(p,c,t)$ and $\varepsilon_a(p,c,t)$).

2.3.5 Magnitude accuracy

Figure 8 depicts the energy estimates for worst and best-case scenarios normalized to the average case (η_w and η_b). A figure almost exactly resembling Figure 8 can be drawn for normalized time estimates (ν_w and ν_b), but is omitted for simplicity. On the one hand, the ratios between best and average cases (ν_b and η_b) exhibit a huge variance from program to program (2 orders of magnitude in most cases), which is largely due to the fact that loops are assumed to execute only once in the best-case execution scenario. This makes τ_b and ε_b unsuitable for estimating ACET and ACEC. On the other hand, the ratios between worst and average cases (ν_w and η_w) show a much smaller variance. Therefore, we might consider τ_w and ε_w as estimates for ACET and ACEC, provided that an inaccuracy of 1 order of magnitude is accept-

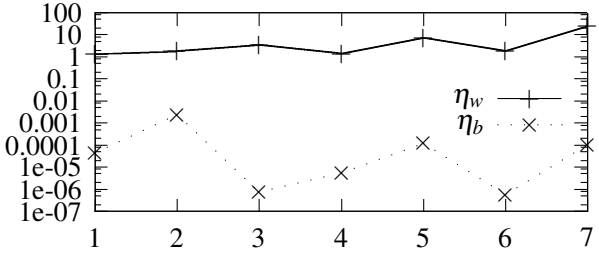


Figure 8 – Normalized energy consumption for each program

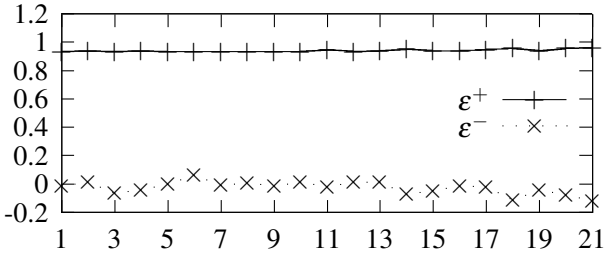


Figure 9 – Correlation between energy estimates for each configuration

able. Although not tolerable at all for hardware allocation, it may be acceptable for compilers, since many optimization heuristics are based on iterative improvement.

2.3.6 Derivative's accuracy

Figure 9 depicts energy correlations for each configuration. Note that the energy consumed by a given program in the worst-case scenario is tightly correlated with ACEC. The energy consumed at the best-case scenario is uncorrelated with ACEC. A figure almost exactly resembling Figure 9 can be drawn for time correlations (τ^-, τ^+), but is omitted for simplicity. Therefore, we can expect worst-case estimates to be proper substitutes for average-case estimates regardless of program and technology, but best-case estimates must be ruled out, since they are inaccurate in magnitude and derivative.

2.3.7 Estimation efficiency

To compare our estimation to the trace-based estimation employed by some energy-aware embedded compilers (VERMA; MARWEDEL, 2007; UDAYAKUMARAN; DOMINGUEZ; BARUA, 2006; CHEN et al., 2006), Figure 10 shows the average speed-up for each program. Our estimation is at least 1000 times faster than those based on profiling. To compare its efficiency to a conventional compiler's (STALLMAN, 2010), we measured the relative contribution of our estimation to compile time, as shown Figure 11. Our estimation contributes with less than 50% of a program's compile time. Note that, although this overhead may seem large for conventional compilers, it is acceptable for embedded compilers, especially those involving real-time analysis.

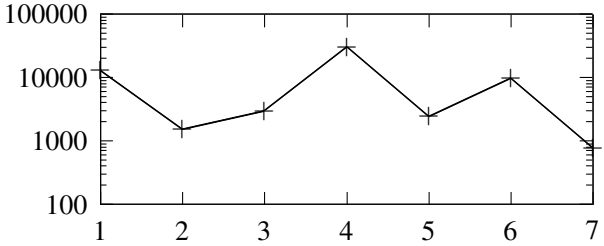


Figure 10 – Estimation speed-up for each program

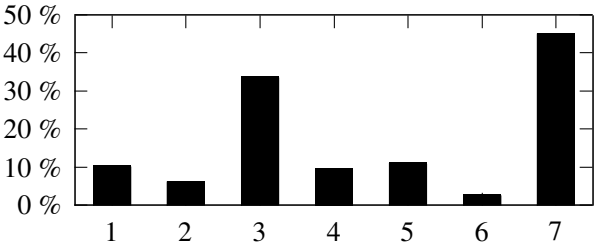


Figure 11 – Estimation overhead on compilation time for each program

3 RELATED WORK

This chapter reviews the main techniques used in instruction cache optimization that are suitable to real-time applications (and hopefully to energy efficiency too). We do not review optimizations based on improving the capture of temporal locality (e.g. procedure sorting (MCFARLING, 1989; HWU; CHANG, 1989)) or on improving spatial locality (e.g. basic block placement, procedure splitting, intraprocedural code positioning (MUCHNICK, 1997)). Since they are based on fixed heuristics targeting average execution time, they are hardly suitable to real-time applications.

3.1 PREFETCHING

Sequential prefetching (SMITH, 1978) 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 always lie in a line contiguous to the one containing the branch instruction. This led to more sophisticated techniques. *Target prefetching* (SMITH; HSU, 1992) 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* (PIERCE; MUDGE, 1996) 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 mecha-

nisms 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 (LUK; MOWRY, 2001). 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 (GUPTA; CHI, 1990), *cooperative prefetching* (LUK; MOWRY, 2001) was proposed. Hardware control is limited to sequential prefetching while non-sequential flows are handled by software prefetching.

3.1.1 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 (AGARWAL et al., 2004). A recent work (TANG et al., 2011) 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.

3.1.2 Prefetching under real-time constraints

There are two conflicting views on how to handle caches under real-time constraints. Those who prescribe cache locking (DING; LIANG; MITRA, 2012; PLAZAR et al., 2012) (to trade-off performance for predictability) argue that cache-aware WCET analysis (FER-

DINAND et al., 1999; THEILING; FERDINAND; WILHELM, 2000) often neglects the interference between tasks (PUAUT, 2002). They prescribe a combination of instruction prefetching and cache locking (PUAUT, 2006) (APARICIO et al., 2010). 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 (FERDINAND et al., 1999; THEILING; FERDINAND; WILHELM, 2000) of cache behavior (during WCET analysis) argue that cache locking may unnecessarily give up performance (APARICIO et al., 2010). Under such assumption, the original cache abstract semantics proposed in (FERDINAND et al., 1999) was extended in (YAN; ZHANG, 2007) to incorporate the effect of *next-N-line prefetching*. Based on such extension, a later work (DING; YAN; ZHANG, 2009) 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.

In contrast with most real-time optimization techniques, which target the minimization of the WCET as a single objective, we propose a novel code optimization technique (relying on software prefetching) for reconciling real-time guarantees and energy efficiency. It reduces the number of misses in unlocked caches and, provenly, does not increase the WCET. The technique, which is described in Chapter 5, relies on preliminary WCET analysis to identify the most profitable prefetches and to determine their prefetching points in the execution flow. In Chapter 6, the impact of the optimization is evaluated as compared to on-demand fetching and cache locking.

3.2 CACHE LOCKING

Full cache locking (FCL) (FALK; PLAZAR; THEILING, 2007), was the first technique to use abstract interpretation to determine which memory blocks should be locked to improve WCET. It builds a list of functions to be successively locked into cache until all cache

blocks are exhausted. The list is sorted according to two criteria. First, functions belonging to the worst-case execution path (WCEP) are sorted by their contribution to the WCET and inserted in the list. Then the remaining functions are inserted, sorted according to their potential impact on the WCEP. Due to limitations of the implementation infrastructure, WCET analysis had to be performed three times in (FALK; PLAZAR; THEILING, 2007) to determine the WCEP. However, the authors claim that it could be performed in a single pass.

An extension of FCL (LIU; LI; XUE, 2009) proposed the use of an Integer Linear Programming (ILP) formulation to obtain an optimal solution to the problem of deciding which functions should be locked. FCL was further improved by refining its granularity (PLAZAR et al., 2012) so that basic blocks are locked instead of whole functions.

Partial cache locking (PCL) (DING; LIANG; MITRA, 2012) also relies on ILP to determine which blocks must be locked. PCL was shown to improve the schedulability of multitasking real-time systems (DING; LIANG; MITRA, 2013).

Like the previous techniques, PCL performs the locking once, in the beginning of the task. To further reduce WCET, dynamic cache locking (DCL) (DING; LIANG; MITRA, 2014) changes which memory blocks are locked into the cache on entry to some loops.

4 MODELING AND PROBLEM FORMULATION

This chapter formalizes the main notions required for modeling cache behaviour, conditional execution, and determination of WCET, which support the technique proposed in the next chapter. Based on such notions, the target optimization problem is defined at the end of this chapter.

4.1 CACHE BEHAVIOR

For self-containment, we review the main concepts from (FERDINAND et al., 1999; THEILING; FERDINAND; WILHELM, 2000). The main storage and the cache are divided in *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* $L = \{l_1, \cdots, l_n\}$ and the main storage by a set of *blocks* $S = \{s_1, \cdots, s_m\} \cup \{I\}$, 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* $\mathscr{U} : C_c \times S \rightarrow C_c$ *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* $\hat{c} : L \rightarrow 2^S$. $\hat{C}$ *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* $\hat{\mathscr{U}} : \hat{C} \times S \rightarrow \hat{C}$ handles abstract states. The abstract update functions used in this work are described in (FERDINAND et al., 1999).

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 (FERDINAND et al., 1999). Although not formally described here (for simplicity), such *must-may* analysis was already introduced informally in the examples of Chapter 1.

Although we rely on such classical functions for preliminary WCET analysis, we propose novel update and join functions to drive code optimization in Section 5.5.

4.2 CONDITIONAL EXECUTION

We assume a conventional representation as starting point:

Definition 3 *Given a program, its control flow graph is a directed graph* $CFG = (B, F)$ *where* $bb_i \in B$ *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) (LI; MALIK, 1995) casts the properties of execution paths into an integer linear programming (ILP) formulation, providing efficient static analysis (THEILING; FERDINAND; WILHELM, 2000) and accurate WCET bounds (FERDINAND et al., 1999). 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_1 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 $n_{bb_1} = n_{bb_2} + n_{bb_3}$. 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 scenario.

4.3 DETERMINATION OF THE WCET SCENARIO

Given a program p and a referenced memory item r, let $t_w^p(r)$ denote the time spent, in the WCET scenario, when accessing that item. Given a basic block bb, let $t_w^p(bb) = \sum_r t_w^p(r)$ 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 is $t_w^p(bb) \times n_{bb}$. The objective function for the ILP problem is:

$$maximize : \sum_{bb \in B} t_w^p(bb) \times n_{bb}, \tag{4.1}$$

whose solution leads to the *number of executions* of each basic block bb in the *WCET scenario*, written n_{bb}^w. Note that $n_{bb}^w = 0$ for every bb not belonging to the WCET path. The overall contribution of an item r to the WCET is:

$$\tau_w^p(r) = t_w^p(r) \times n_{B(r)}^w, \tag{4.2}$$

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:

$$\tau_w^p = \sum_{bb \in B} t_w^p(bb) \times n_{bb}^w \tag{4.3}$$

4.4 PROBLEM FORMULATION

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

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

Problem 1 *Given a program p, find a prefetch-equivalent program p' such that $\tau_w^{p'} \leq \tau_w^p$ and it minimizes the energy consumption, for a given prefetch latency Λ, a given cache configuration, and a given process technology.*

The use of a cost function that fully captures energy consumption may unnecessarily increase runtime. Since the results in Chapter 2 recommend the use of iterative improvement to exploit fast energy estimation, the technique described in the next chapter solves an *instance* of Problem 1 where the miss rate is used as a cost function, because it is proportional to the dynamic consumption in main memory and proportional to the static consumption in the whole memory subsystem.

5 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. 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.

5.1 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 proposed in (FERDINAND et al., 1999), leading to an implicit loop representation where back edges are broken:

Definition 6 *Given a program, its abstract control flow graph is a polar, directed acyclic graph* $ACFG = (R, E)$ *where each vertex* $r_i \in R$ *is a reference to a memory item and each edge* $(r_i, r_j) \in E$ *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 (*$\bullet$*) and the sink (*$\odot$*).*

To denote that r_i reaches r_j through a path in the ACFG, we write $r_i \rightsquigarrow r_j$. 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 a directed acyclic graph* $ACFG^* = (R, E^*)$ *such that there exists an edge* $(r_j, r_i) \in E^*$ *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 $ACFG^*$ as $PRED^*(r) = \{r' \in R \mid (r', r) \in E^*\}$ and $SUCC^*(r) = \{r' \in R \mid (r, r') \in E^*\}$, respectively. A given predecessor of r is denoted as $pred^*(r)$.

Definition 8 *Given an item r_i, we write $\mathscr{S}(r_i)$ to denote the memory block where r_i is stored. Conversely, given a memory block s, we write $\mathscr{R}(s)$ 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 4.1.

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

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

Property 1 *When $\mathscr{B}(r_i, r_{i+1}) - \mathscr{B}(r_{i-1}, r_i) = \emptyset$, the access to item r_i resulted in a hit.*

Property 2 *When $\mathscr{B}(r_i, r_{i+1}) - \mathscr{B}(r_{i-1}, r_i) = \{s\}$, the access to item r_i resulted in a miss and r_i is stored in memory block s.*

Property 3 *When $\mathscr{B}(r_{i-1}, r_i) - \mathscr{B}(r_i, r_{i+1}) = \{s'\}$, the access to item r_i replaced the memory block s'.*

5.2 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 12, our technique is applied to a simple straight-line program. From the *ACFG* of the original program (12a), it shows the

$ACFG^*$s representing the intermediate optimization steps for each visited vertex (12b) until the $ACFG$ of the optimized program is obtained (12c). 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 order (12a, 12c) or the replaced blocks in reverse order (12b). The blocks of a cache line are denoted as $[MRU, LRU]$, to indicate the most and the least recently used blocks.

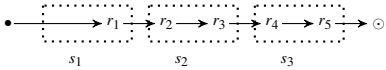
Figure 12a shows the states at each edge of the ACFG. By applying Properties 1 and 2 to every successive pair of edges, we obtain whether the outcome was a hit or a miss.

Figure 12b presents our reverse analysis step-by-step from sink to source. Initially, the edge $(\odot, r_5)$ 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_5, r_4, r_3, and r_2 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 π_{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, π_{s_3}) and (π_{s_3}, r_1). When π_{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 π_{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 12c, 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.

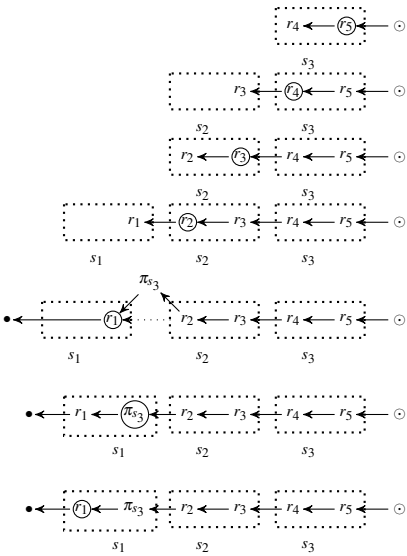
A second example handles conditional constructs with the help of join functions. When a reference r is reached from distinct paths, the state at its leaving edge depends on the taken path. To derive a single output state from multiple input states, *join vertices* are added to the $ACFG$ ($ACFG^*$) and their behaviors are modeled by *join functions*, as

a) Original program



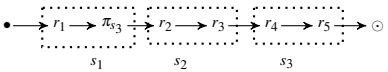
edge	state	outcome
$(\bullet, r_1)$	$[I, I]$	
(r_1, r_2)	$[s_1, I]$	miss
(r_2, r_3)	$[s_2, s_1]$	miss
(r_3, r_4)	$[s_2, s_1]$	hit
(r_4, r_5)	$[s_3, s_2]$	miss
$(r_5, \odot)$	$[s_3, s_2]$	hit

b) Optimization steps



edge	state	replaced
$(\odot, r_5)$	$[I, I]$	
(r_5, r_4)	$[s_3, I]$	none
(r_4, r_3)	$[s_3, I]$	none
(r_3, r_2)	$[s_2, s_3]$	none
(r_2, r_1)	$[s_2, s_3]$	none
$(r_1, \bullet)$	$[s_1, s_2]$	s_3
(r_2, π_{s_3})	$[s_2, s_3]$	
(π_{s_3}, r_1)	$[s_1, s_2]$	s_3
$(r_1, \bullet)$	$[s_1, s_2]$	none

c) Optimized program



edge	state	outcome
$(\bullet, r_1)$	$[I, I]$	
(r_1, π_{s_3})	$[s_1, I]$	miss
(π_{s_3}, r_2)	$[s_3, s_1]$	hit
(r_2, r_3)	$[s_2, s_3]$	miss
(r_3, r_4)	$[s_2, s_3]$	hit
(r_4, r_5)	$[s_3, s_2]$	hit
$(r_5, \odot)$	$[s_3, s_2]$	hit

Figure 12 – The technique applied to a straight-line program

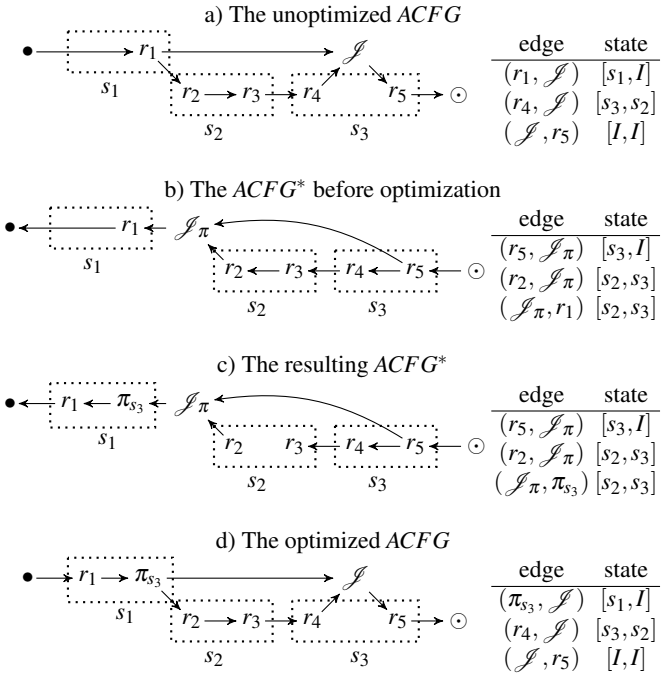


Figure 13 – How conditional control flows are handled

illustrated in Figure 13. Figure 13a shows that a vertex $\mathscr{J}$ performing a conventional join function determines the state at its leaving edge as the intersection of the states of its entering edges (FERDINAND et al., 1999). Figure 13b shows that a vertex $\mathscr{J}_\pi$ 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 13c and 13d directly show the resulting *ACFG** and *ACFG*.

5.3 HOW LOOPS ARE HANDLED

This supplemental example illustrates that, to handle loops, our technique relies on the VIVU transformation (FERDINAND et al., 1999) (which is often employed by conventional WCET analysis) to derive an acyclic *ACFG* from a cyclic *CFG*. Figure 14a shows a *CFG*

prior to the VIVU transformation, where a back edge closes a loop. Figure 14b 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 control flow. In that figure, r_2^f denotes the reference to an item r_2 in the *first* loop iteration and r_2^o denotes the reference to the same item in *other* loop iterations. From the *ACFG* in Figure 14b, our technique obtains the $ACFG^*$ in Figure 14c, according to the mechanisms already illustrated in the previous examples. Figure 14d shows the resulting CFG for the optimized program.

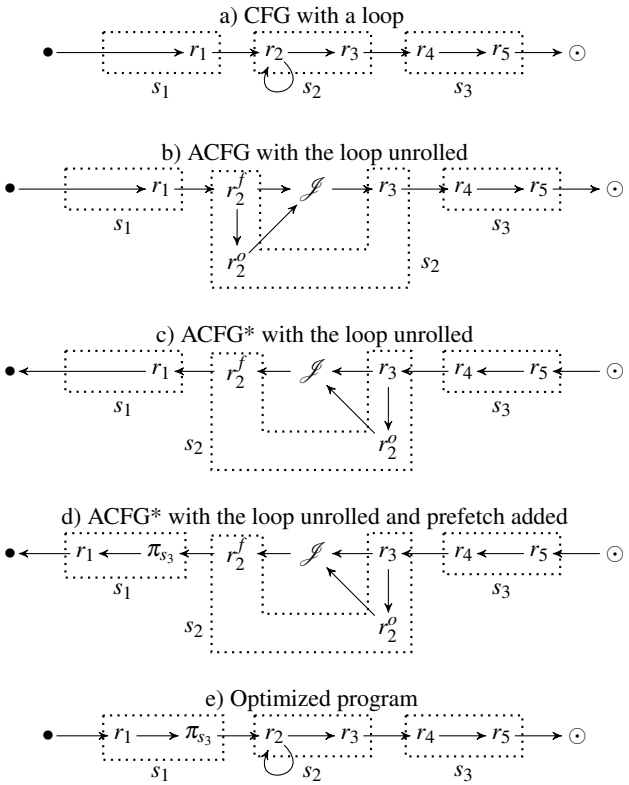


Figure 14 – How the technique handles loops

5.4 THE JOINT IMPROVEMENT CRITERION

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

$$\tau_w^p(r_i, r_j) = \sum_{r \in \{x | x \in R \wedge r_i \rightsquigarrow x \rightsquigarrow r_j\}} t_w^p(r) \times n_{B(r)}^w \tag{5.1}$$

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_i, r_{i+1}) denote some program point such that $r_i \rightsquigarrow r_j$. To check if the insertion of $\pi_{s'}$ at program point (r_i, 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:

$$t_w^{p_{n-1}}(r_{i+1}, r_{j-1}) = \sum_{r \in \{x | x \in R \wedge r_{i+1} \rightsquigarrow x \rightsquigarrow r_{j-1}\}} t_w^{p_{n-1}}(r) \tag{5.2}$$

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

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}:

$$mcost(r_j) = \tau_w^{p_{n-1}}(r_j) \tag{5.3}$$

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 $\pi_{s'}$ inserted at point (r_i, r_{i+1}) in program p_{n-1}, leading to a new program p_n:

$$pcost(r_i) = \tau_w^{p_n}(\pi_{s'}) + \tau_w^{p_n}(r_j) \tag{5.4}$$

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

$$rcost(r_i) = \sum_{r \in \{x | x \in R \wedge x \rightsquigarrow r_i\}} \tau_w^{p_n}(r) - \sum_{r \in \{x | x \in R \wedge x \rightsquigarrow r_i\}} \tau_w^{p_{n-1}}(r) \tag{5.5}$$

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 $r_i \rightsquigarrow r_j$ 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:

$$profit(r_i, r_j) = \begin{cases} 0 & \text{if } r_j \text{ is a prefetch} \\ 0 & \Lambda > t_w^{p_{n-1}}(r_{i+1}, r_{j-1}) \\ 0 & rcost(r_i) > 0 \\ mcost(r_j) - pcost(r_i) & \text{otherwise} \end{cases} \tag{5.6}$$

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 suppress that miss (in a program p_n).

5.5 THE NOVEL OPTIMIZATION ALGORITHM

As a precondition, our algorithm assumes that traditional WCET analysis (to determine $t_w^p(r)$ for each $r \in R$ and n_{bb}^w for every $bb \in B$) and classical VIVU analysis (to transform a cyclic *CFG* into an acyclic *ACFG*) (FERDINAND et al., 1999) were performed beforehand.

The proposed optimization algorithm relies on the novel update function informally described in Figure 12 and the novel join function illustrated in Figure 13. Their formal descriptions are available in Algorithms 1 and 2, which are explained next.

We propose the *prefetching update function* $\hat{\mathscr{U}}_\pi : \hat{C} \times R \to \hat{C}$ 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* $\hat{\mathscr{J}}_\pi : \hat{C} \times \hat{C} \to \hat{C}$ 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 $\hat{\mathscr{U}}_\pi(\hat{c}, r_i)$

```
1  s = 𝒮(r_i)
2  if ∃s' ∈ S | ℬ(ĉ) − ℬ(Û(ĉ,s)) = {s'} ≠ {I}:
3       r_j = ℛ(s')
4       if profit(r_i, r_j) > 0:
5            R  := R ∪ {π_s'}
6            E* := E* ∪ {(pred*(r_i), π_s'), (π_s', r_i)}
7            E* := E* − {(pred*(r_i), r_i)}
8            relocate_upwards(r_i)
9            return Û_π(ĉ, π_s')
10 return Û(ĉ,s)
```

Our technique, which is formally described in Algorithm 3, runs a *non-conventional* static analysis in *reverse execution order* to find the profitable prefetches that do not increase the WCET.

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

Algorithm 2 The proposed join function $\hat{\mathscr{J}}_\pi(\hat{c}_1, \hat{c}_2)$

```
1  let (r_x, J) ∈ E* | ĉ(r_x, J) = ĉ_1
2  let (r_y, J) ∈ E* | ĉ(r_y, J) = ĉ_2
3  if mcost(r_x) < mcost(r_y)
4       return ĉ(r_y, J)
5  else
6       return ĉ(r_x, J)
```

Algorithm 3 The proposed prefetching optimization

```
1  build ACFG* = (R, E*) from program p
2  ≺_T = {(u,v) ∈ R × R | (u,v) ∈ E* ∨ (v,u) ∉ E*}
3  let succ_{≺_T}(r) be the successor of r in ≺_T
4  c(⊙, succ*(⊙)) := ĉ_I
5  r := ⊙
6  while (succ_{≺_T}(r) != •)
7       {r_z} := SUCC*(r)
8       if r is a join vertex:
9            {r_x, r_y} := PRED*(r)
10           ĉ(r_z, r) := Ĵ_π(ĉ(r_x, r), ĉ(r_y, r))
11      else
12           {r_x} := PRED*(r)
13           ĉ(r_z, r) := Û_π(ĉ(r_x, r), r)
14      r := succ_{≺_T}(r)
15 build ACFG = (R, E) for program p'
```

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).

Let us now evaluate the worst-case complexity of the proposed algorithm. 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 5.3 and 5.4 takes $O(1)$. Although the second summation of Equation 5.5 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 $CFG = (B,F)$, we bound the set R by virtually unrolling each loop at most once when applying the VIVU transformation (FERDINAND et al., 1999).

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 Appendix A).

6 EXPERIMENTAL EVALUATION

This chapter directly compares the technique proposed in the previous chapter with partial cache locking (DING; LIANG; MITRA, 2012) when using the fast estimation technique proposed in Chapter 2.

6.1 EXPERIMENTAL SETUP

We ran both techniques on all 37 programs of the Mälardalen WCET benchmark (GUSTAFSSON et al., 2010), which are shown in Table 2, along with the sizes of the respective binaries.

Table 2 – The adopted benchmark suite

Size (kB)	Program	Size (kB)	Program
595	duff	600	edn
599	sqrt	594	insertsort
592	crc	597	fdct
598	compress	593	select
597	cnt	595	ludcmp
600	qurt	608	statemate
591	bs	601	fir
592	expint	597	qsort-exam
597	bsort100	602	adpcm
600	ns	591	fibcall
596	ndes	594	jfdctint
595	whet	591	recursion
591	lcdnum	598	cover
599	lms	623	nsichneu
591	fac	618	st
595	ud	601	fft1
596	minver	592	matmult
594	janne_complex	595	prime
603	des		

Each program was run under 36 cache configurations and two technologies (45nm and 32nm), leading to 2664 use cases. The cache configurations employed in our experiments are denoted as $k = (a, b, c)$ in Table 3, where a is the associativity, b is the block size (in bytes), and c is the cache capacity (in bytes).

Table 3 – Cache configurations

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

We assume that each program fully owns the instruction cache. This choice captures our understanding that real-time schedulers should incorporate some mechanism to minimize the interference between tasks over the cache. For instance, proper procedure placement on the address space may be used to reduce conflicts or locking may be used to prevent that the blocks brought to cache by a preempted task could be overwritten by the preempting task. Note that, in the latter scenario, our technique would be applied to the unlocked blocks of the cache, which would represent the *effective* cache capacity seen by a given task (in this case, cache locking would not be used to optimize the task but only to avoid cache interference between tasks).

We selected cache capacities so that the average miss rate lies in a large span from 1% to 10% before any of the techniques under

comparison is applied. The resulting cache capacities (in the interval between 256B and 8kB) may seem small in face of real-life cache sizes. However, they just reflect the fact that the chosen benchmark suite consists of small programs whose working set would be fully contained in cache if capacities larger that 8kB were employed. This apparent small caches are the price to pay for the benefit of using a well-known WCET benchmark that was not developed to mimic real-life applications requiring larger memories. A 128MB DRAM was employed as level-two memory.

Since the adopted benchmark was originally designed for WCET evaluation, each program comes with a single set of stimuli, which is embedded in the source code. As we want to compare the techniques also in terms of ACET and energy consumption, it was necessary to modify the source code of each program to accept stimuli from an input file. This allowed us to generate distinct input files with different stimuli. In general, 100 sets of stimuli were randomly generated for each program, except for a few programs whose specificities precluded such an approach. Among the exceptions, there are a couple of programs whose average case behavior depends on the entropy of the input files. For instance, the programs `compress` and `adpcm` tend to reach the worst-case behavior when the stimuli exhibit high entropy. In such cases, we arbitrarily selected 95 sets of binary files containing non-random data. The remaining 5 sets were obtained by generating input files with random content. The major constraint on the generation of multiple sets of stimuli was imposed by the program `bs`. Since it performs binary search within a predefined array with 15 entries, we employed only 20 stimuli (in 15 of them, the elements under search were stored in the array; in 5 of them, they were not).

Both the prefetching technique described in Chapter 5 and PCL (DING; LIANG; MITRA, 2012) were integrated into the GNU compiler (version 4.9.1). We used the '-O2' optimization level and targeted ARMv7. We implemented our own WCET analyzer based on (FERDINAND et al., 1999; THEILING; FERDINAND; WILHELM, 2000) and integrated its components into the tool prototypes of the tech-

niques under comparison. WCET analysis is performed in two steps. The first step occurs within the GNU compiler and determines loop iteration bounds. The second is a post-compiling step that performs the implicit path enumeration technique (LI; MALIK, 1995) and must-may analysis (THEILING; FERDINAND; WILHELM, 2000).

For ACET and energy estimation, we relied on a traditional trace-based approach. For trace generation, we employed an instruction-set simulator available within the GEM5 (BINKERT et al., 2011) simulation environment.

We employed the CACTI 6.5 power/energy model (WILTON; JOUPPI, 1996) 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 (resulting from the insertion of prefetch instructions) on the processor's energy consumption and execution time. However, as will be shown, the measured increase in the number of executed instructions is negligible. Therefore, its impact is likely to be marginal.

6.2 EXPERIMENTAL RESULTS

Figure 15 shows average improvements for the prefetching technique as a function of cache size. The overall average improvement was 10.8% for both ACET and energy consumption. Indeed, energy savings were obtained for all use cases without increasing the memory's contribution to the ACET. To achieve such energy efficiency, the maximal increase in the number of executed instructions was 1.32%. The non-increasing ACET has two consequences: the memory's static energy and the average number of cycles per instruction are not increased (as far as time anomalies are considered second-order effects). As the amount of inserted instructions is negligible, the optimization of the *memory* subsystem may only marginally increase the static consumption of the *rest* of the system.

Figure 15 also plots the average improvement in the WCET. It should be noted that, to preserve real-time guarantees, the prefetching

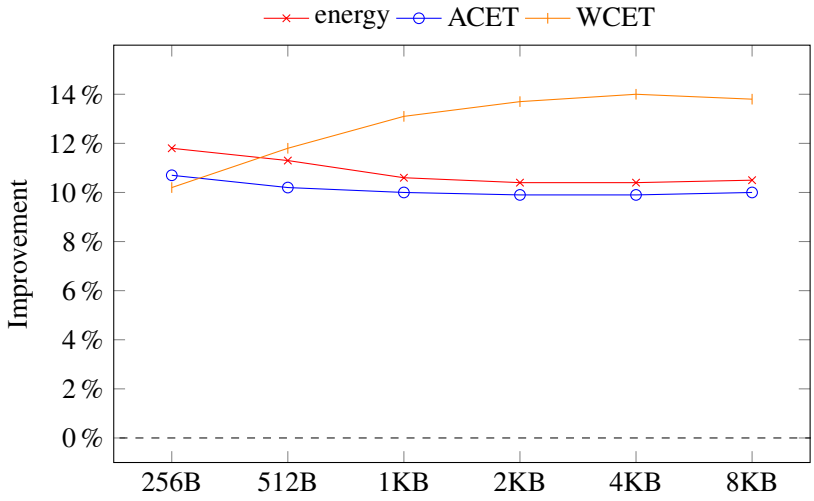


Figure 15 – Impact of prefetching

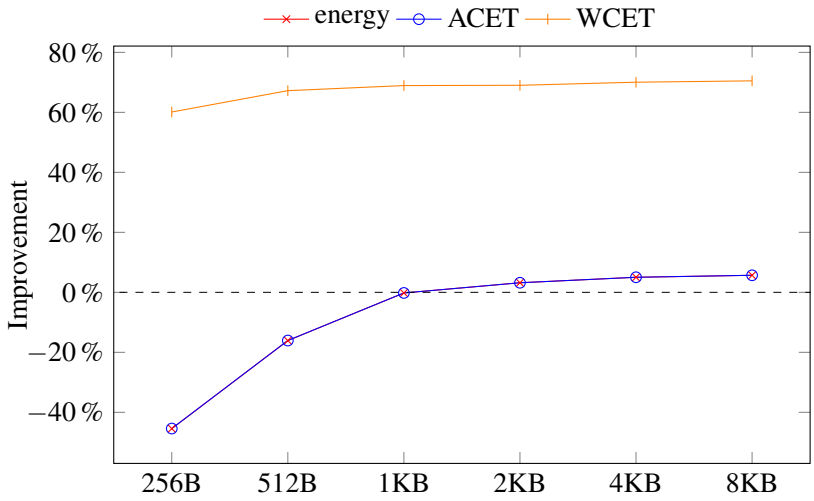


Figure 16 – Impact of cache locking

technique described in Chapter 5 employs the WCET as a *constraint* and therefore does not try to optimize it. However, an average improvement of 14.6% was observed. This shows that, by simply constraining the prefetches that would impair the WCET, it is possible to reduce it

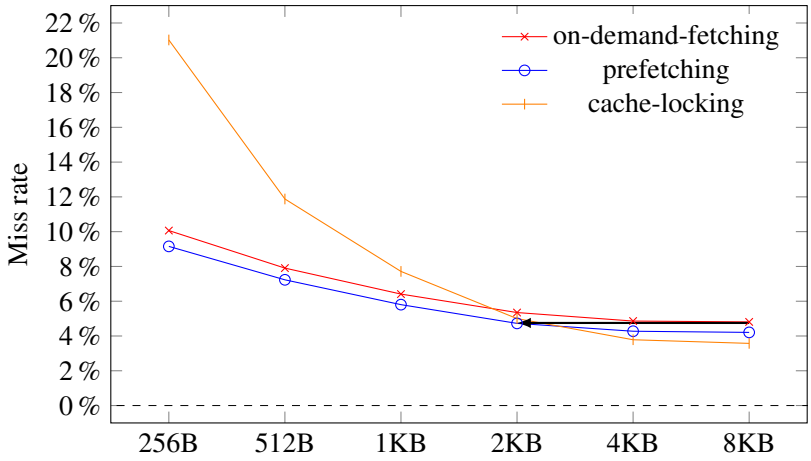


Figure 17 – Impact on miss rate

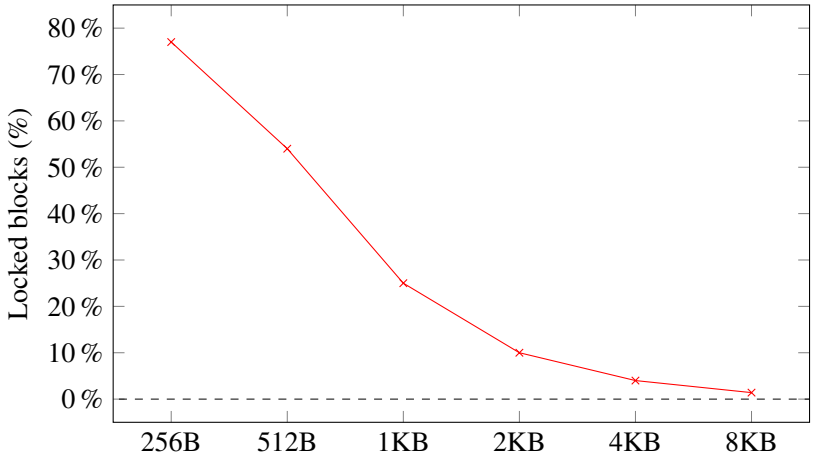


Figure 18 – Percentage of locked blocks

while also reducing energy consumption.

Figure 16 shows the average improvements obtained with cache locking. The overall improvement for the WCET was 67.6% but at expense of a worsening of 7.9% in ACET and energy consumption.

Observe that the improvement on ACET is very close to the improvement on energy, because the static energy is proportional to

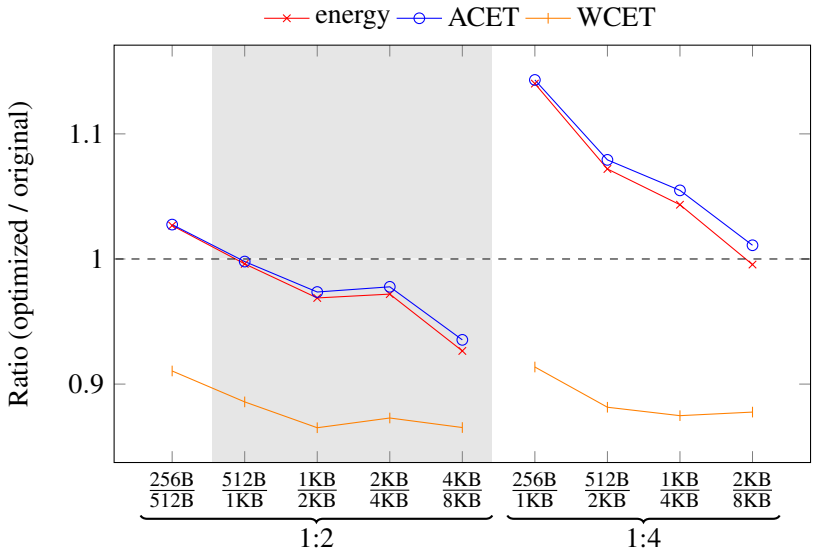


Figure 19 – Prefetching: higher energy efficiency with smaller caches

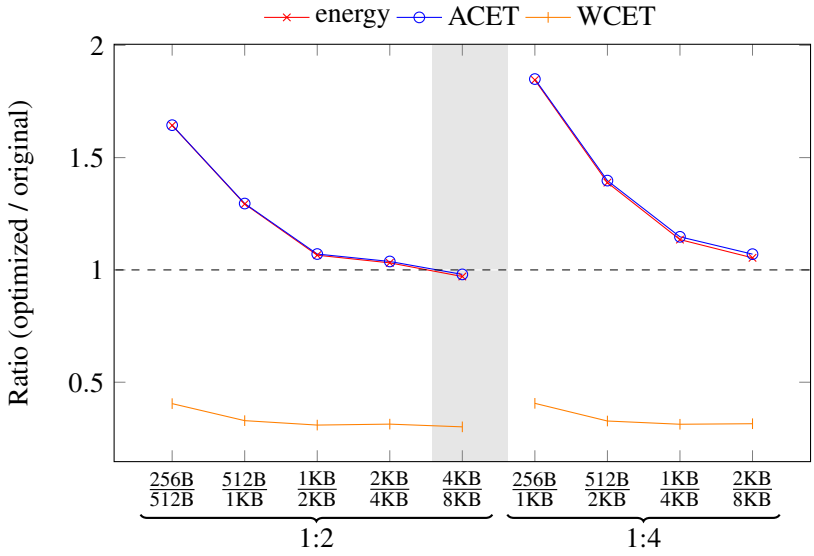


Figure 20 – Cache locking: smaller energy efficiency with smaller caches

the ACET and the dynamic energy, like the ACET, is a function of the miss rate. That is why the respective curves seem to coincide in Figures 15 and 16, depending on the scale.

Note that the improvement on WCET with cache locking is four times larger than the one obtained by prefetching. This is not surprising for two main reasons. First, minimizing the WCET is a single objective for the former and a constraint for the latter. Second, as illustrated in the examples of Section 1.3, locking a block in cache degenerates the associativity of a set, which allows for tighter estimations of the WCET, but tends to increase ACET and energy consumption.

Notice also that, for cache locking, the improvement on WCET is essentially the same regardless of cache size (except for the smallest cache). This seems to be a consequence of its single objective formulation. On the other hand, although the prefetching technique employs iterative improvement for energy and ACET, it does *not* try to optimize the WCET. However, since the potential for block replacement is reduced as cache size increases, the predictability tends to be higher for larger caches, leading to larger improvements on the WCET.

Let us analyze the energy improvements of each technique for different cache sizes. Note that prefetching is more energy efficient for all cache sizes. Notice that cache locking is especially energy inefficient for small caches. It can only save energy when cache sizes are large enough to store most of a program's working set. To reach a maximum improvement of 5.6%, cache locking required 8KB caches. For many of the benchmark programs, such capacity is large enough to keep almost their whole working sets. For exactly the same reason, prefetching leads to smaller improvements for large caches, since there is less margin for miss rate optimization. In spite of that, it exhibits twice the energy improvement for 8KB caches as compared to cache locking.

Figure 17 shows the impact on miss rate. The overall average miss rate reduction using prefetching is 11%. As the arrow shows, the programs optimized with prefetching require less cache capacity than the unoptimized ones to sustain the same miss rate. On the other hand, the overall average miss rate was increased by 25%, with respect to on-

demand fetching, when using cache locking. This is another evidence of why the improvement in WCET comes at the expense of higher ACET and energy consumption. Note that, as compared to on-demand fetching, cache locking was only able to reduce the miss rate for the two largest cache sizes. To expose the reason for that behavior, Figure 18 plots the percentage of cache blocks that were locked. Note that 77% of the cache blocks were locked for the smallest cache but only 1.4% for the largest one (to achieve an almost constant improvement in WCET as depicted in Figure 16). This is a clear evidence that it is *not* the locking mechanism that produces the miss rate reduction for large sizes, but the fact that most of the working set is stored in cache.

As prefetching 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 19 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 area at the lefthand side, the programs optimized with prefetching sustained ACETs less or equal to the unoptimized ones even using only half of the original cache size. Although this can not be sustained for a quarter of the original cache size, note that a small improvement in energy was still observable at the expense of around 1% increase in ACET (for the configuration at 2KB/8KB). Although proven WCET guarantees are provided by the prefetching technique (WUERGES; OLIVEIRA; SANTOS, 2013) when the original and the optimized program run on the same cache configuration, such theoretical guarantees cannot be kept when comparing their behaviors on configurations with arbitrarily selected sizes. However, as Figure 19 indicates, the WCET did not grow for any use case when cache sizes were reduced. This is an evidence that the prefetching technique, by enabling the use of smaller caches, can lead to energy reductions up to 8% while sustaining the same or superior performance and preserving real-time guarantees.

A similar experiment was performed for cache locking, as shown in Figure 20. Note that, as opposed to prefetching, there is a single

configuration for which a slight improvement in energy efficiency was observed.

Therefore, the shaded areas in Figures 19 and 20 indicate that, as opposed to cache locking, prefetching may still be suitable for energy-efficient real-time systems even when cache sizes are over-constrained by design requirements.

For the program that took the longest (`adpcm`), the prototype tool for prefetching took on average 0.43 seconds to run each WCET analysis and 8 minutes to perform the proposed optimization on a workstation based on an Intel `i7-2600K` processor, with 8GB RAM, running at 3.4 GHz. For the same program, cache locking took 21 minutes. When averaged in the whole set of programs, performing cache locking took 2.4 times longer than performing the prefetching optimization.

Figure 21 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%.

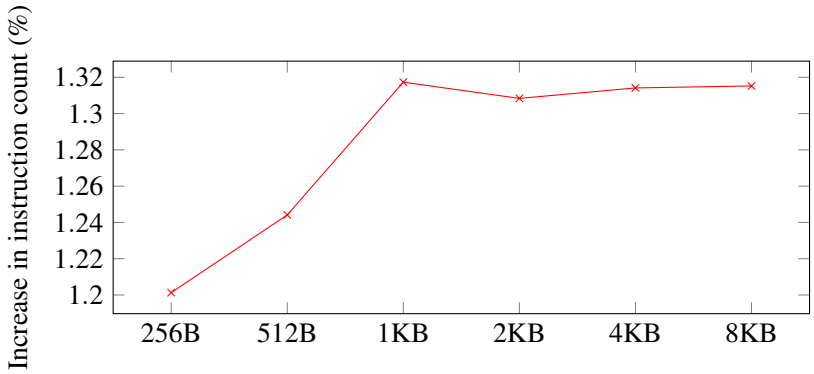


Figure 21 – The negligible overhead of the technique

7 CONCLUSION

7.1 CONCLUDING REMARKS

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.

Since every executed instruction is eventually accessed in cache and its energy per access is fixed for a given configuration, neither prefetching nor cache locking can decrease the dynamic energy spent in cache. Indeed, prefetching does increase cache dynamic consumption proportionally to the number of prefetch instructions that are executed. However, as the proposed technique was designed to insert prefetches only when the WCET is provenly not increased, very few are actually inserted (representing less than 1.3% of the instruction count). Besides, not all of them are executed. As a result, the increase in cache dynamic consumption becomes negligible. Therefore, the issue of saving energy in the memory system narrows down to reducing dynamic consumption in the main memory and static consumption in both cache and main memory. In both cases, the key is a reduction in miss rate, since it decreases both the number of accesses to main memory and the ACET.

Unfortunately, as compared to software prefetching, cache locking has inherently less opportunities to prevent misses when referencing unlocked blocks (for a given cache capacity). Cache locking cannot avoid compulsory misses (except, of course, for the locked blocks). Since locking decreases the effective capacity of a set and reduces its placement alternatives, both capacity and conflict misses may be increased. On the contrary, prefetching may reduce all of them (compulsory, capacity, and conflict misses), because its handling of misses is not limited by construction but merely constrained by the WCET. In cache locking, however, the handling of misses is limited by the choice of a single objective: to reduce the WCET as much as possible.

The experiments provided quantitative evidence that, although cache locking is far superior for real-time systems where energy efficiency is not a major concern, software prefetching of unlocked caches becomes a suitable alternative when energy efficiency requirements cannot be neglected in real-time systems. The results have also shown that software prefetching may sustain both average performance and WCET even when small caches are required (e.g. when the application domain asks for higher clock rates but low energy consumption).

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* 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 (RODRÍGUEZ; JACOB, 2006) 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.

7.2 LIMITATIONS AND EXTENSIONS

- **Two-level memory hierarchy**:

  Since our technique optimizes the code of a real-time task that will be handled by an RTOS, we assume a memory hierarchy without virtual memory (no transaction look-aside buffer). Since current processors used in real-time applications have a single cache level, we evaluated our technique for a two-level memory hierarchy. However, the proposed optimization technique is able to handle extra cache levels (if required in the future), since abstract cache semantics does not change by adding more levels of cache. In such scenario, only the modeling of the memory hierarchy must be extended, not the optimization algorithm.

- **Partial energy modeling of the system**:

Since we targeted memory optimization, we built (from scracth) an infrastructure able to provide safe bounds for the contribution of the memory subsystem to the WCET and to provide accurate estimates for the contribution of the memory subsystem to energy consumption. Although the current infrastructure provides a functional model of the rest of the system (ISA simulator), it does not provide the fine-grain models (processor microarchitecture and interconnect fabric) required for accurate energy estimation. However, we provide a few guarantees that such current limitation of the implementation infrastructure does not jeopardize the evaluation of the proposed optimization technique. Since our technique improved the ACET in all evaluated use cases, if the number of executed instructions were not increased as a result of the proposed optimization, this would mean that the average number of cycles per instruction would *not* be increased by the memory subsystem, being an evidence that no extra static energy would be drained in the rest of the system due to extra runtime. Since the number of inserted prefetch instructions is not zero but negligible and assuming that time anomalies can be ruled out, we can claim that not only their contribution to the static consumption in the rest of the system is negligible, but also their contribution to the dynamic energy consumption in the processor and in the interconnect.

- **No modeling of cache coherence**:

  Since in a multi-tasking environment, the use of shared variables among real-time tasks is unlikely or at least limited, we did not model the impact of cache coherence protocols in abstract cache semantics in our prototype tool. Although this may not substantially change our conclusions about its effectiveness, if the proposed technique is to be used in an environment where tasks do share variables, abstract semantics have to be extended to incorporate the effects of coherence in order to ensure safe WCET bounds.

- **No modeling of impact of preemption in WCET analysis**:

  The proposed optimization technique was applied to the whole scope of the code of a given real-time task, assuming implicitly that the optimized task is not preempted by another. If a preemptive scheduler is used, however, the impact of preemption would not be captured. To overcome this limitation in a pragmatic and safe way, the impact of preemption should be constrained by the insertion of *preemption points* in the program (BERTOGNA et al., 2010). The key idea is to break the program into non-preemptible segments and perform the scheduling algorithm on the task set together with WCET analysis. Since the accuracy of WCET analysis heavily relies on the initial cache state, the algorithm proposed in (BERTOGNA et al., 2010) runs in two steps. First, the unscheduled task of highest priority is scheduled. Then preemption points are placed in all other unscheduled tasks (based on the WCET analysis of the non-preemptible segments of the scheduled tasks of higher priority). The algorithm repeats such two steps until all tasks are scheduled. In such scenario, the scope of optimization of the proposed technique is a *non-preemptible segment of a task* and not the whole program. This extension is recommended as future work.

- **No modeling of hardware prefetching**:

  The prototype tool implementing the proposed code optimization currently assumes that the processor does not perform hardware prefetching at all. Although every pipelined processor naturally performs instruction prefetch, *sequential* prefetching does not affect predictability and the required extension in the abstract semantics is straightforward. Unfortunately, many embedded processors rely on less deterministic architectures that employ hardware prefetching based on *branch* prediction. For instance, the ARM Cortex-R and Cortex-M processors come with a prefetching unit (PFU), which relies on a branch predictor to prefetch instructions either from fall-through or target addresses. In such

scenario, the application of the proposed technique would require disabling either the FPU or, at least, the dynamic branch prediction mechanism.

When it is neither possible to disable hardware prefetching nor branch prediction, the effects of hardware prefetching must be incorporated into the WCET analysis (BURGUIERE; ROCHANGE, 2005) before the proposed technique can be safely applied. In such scenario, the main challenge would be the accurate modeling of the dynamic components of the hardware so as to make sure that every possible state of the cache can be captured by the abstract cache states. Therefore, although extensions in the WCET analysis may be needed, no changes are required in the proposed prefetching algorithm.

7.3 PERSPECTIVES

- **Generalization for data caches**:

  Since this work has shown that instruction prefetching is able to effectively reduce energy consumption under real-time constraints and since data caches are the second major energy consumer among the storage components of an embedded processor (DALLY et al., 2008), this thesis paves the way to the generalization of the proposed optimization algorithm for handling unlocked *data* caches. However, there are a few challenges to be faced. Although the software prefetching of instructions benefits from the fact that the addresses of most instructions are known at compile time (and can therefore be represented within an immediate field of a prefetch instruction), the main difficulty with software data prefetching comes from the fact that the addresses of most variables are only available at runtime and they can change for different iterations of a same loop or distinct invocations of a given function. Besides, the calculation of addresses for data prefetching requires extra register usage. The competition between

registers required to support data prefetching and those needed for the original computation, has the potential to disrupt register allocation (SMITH, 1978). Finally, not all addresses can be calculated. The address of an array element can be calculated if the base, the index, and the stride are known. This allows data prefetching for subsequent iteration of a loop scanning an array. However, more complex data structures (such as trees and linked lists) are a concern (LUK; MOWRY, 1996) since the use of pointers lead to distinct aliases to the same address.

- **Evaluation of impact for real-life applications**:

  Although this thesis have evaluated the proposed technique for a large set of use cases, its impact heavily depends on the memory parameters of a given architecture, which are dictated by the target application. Therefore, case studies with application-specific real-life requirements are let as future work.

APPENDIX A – FORMAL GUARANTEES

This appendix presents the formal proofs that the proposed technique does not increase the WCET as far as all memory operations are kept in program order. Such proofs were elaborated by the author's supervisor, Prof. Luiz C. V. dos Santos, as a theoretical contribution to the cooperative work described in (WUERGES; OLIVEIRA; SANTOS, 2013). They are presented here for completeness.

To improve readability, this section adopts $\sum_{r \rightsquigarrow r_i} \tau_w^{p_n}(r)$ as a shorthand notation for $\sum_{r \in \{x | x \in R \wedge x \rightsquigarrow r_i\}} \tau_w^{p_n}(r)$.

Lemma 1 *Given the ACFG representing a program p_{n-1} and a path $(r_{i+1}, \cdots, r_j)$, if Algorithm 1 inserts, at program point (r_i, r_{i+1}), a prefetch $\pi_{s'}$ for a block $s' = \mathscr{S}(r_j)$, thereby generating a program p_n, the overall contribution to the WCET of all memory items referenced on the new path $(\pi_{s'}, r_{i+1}, \cdots, r_j)$ is smaller than on path $(r_{i+1}, \cdots, r_j)$, i.e. $\tau_w^{p_n}(\pi_{s'}, r_j) < \tau_w^{p_{n-1}}(r_{i+1}, r_j)$.*

Proof 1 Line 4 of Algorithm 1 guarantees, via Equation 5.6, that a prefetch $\pi_{s'}$ is inserted only if $pcost(r_i) < mcost(r_j)$, which from Equations 5.3 and 5.4 leads to $\tau_w^{p_n}(\pi_{s'}) + \tau_w^{p_n}(r_j) < \tau_w^{p_{n-1}}(r_j)$ (I). Since $\pi_{s'}$ is inserted immediately before r_{i+1} and every vertex r such that $r_{i+1} \rightsquigarrow r \rightsquigarrow r_{j-1}$ is untouched by Algorithm 1, we can write $\tau_w^{p_n}(r_{i+1}, r_{j-1}) = \tau_w^{p_{n-1}}(r_{i+1}, r_{j-1})$ (II). Thus, from (I) and (II) we conclude that $\tau_w^{p_n}(\pi_{s'}) + \tau_w^{p_n}(r_{i+1}, r_{j-1}) + \tau_w^{p_n}(r_j) < \tau_w^{p_{n-1}}(r_{i+1}, r_{j-1}) + \tau_w^{p_{n-1}}(r_j)$. Therefore, from Equations 4.2 and 5.1, we can write $\tau_w^{p_n}(\pi_{s'}, r_j) < \tau_w^{p_{n-1}}(r_{i+1}, r_j)$.

Lemma 2 *Given the ACFG representing a program p_{n-1} and a path $(r_{i+1}, \cdots, r_j)$, if Algorithm 1 inserts, at program point (r_i, r_{i+1}), a prefetch $\pi_{s'}$ for a block $s' = \mathscr{S}(r_j)$, thereby generating a program p_n, the overall contribution to the WCET of all memory items reaching r_i is not increased, i.e. $\sum_{r \rightsquigarrow r_i} \tau_w^{p_n}(r) \leq \sum_{r \rightsquigarrow r_i} \tau_w^{p_{n-1}}(r)$.*

Proof 2 Line 4 of Algorithm 1 guarantees, via Equation 5.6, that a prefetch $\pi_{s'}$ is inserted only if $rcost \leq 0$, which from Equation 5.5 leads to $\sum_{r \rightsquigarrow r_i} \tau_w^{p_n}(r) - \sum_{r \rightsquigarrow r_i} \tau_w^{p_{n-1}}(r) \leq 0$, i.e. $\sum_{r \rightsquigarrow r_i} \tau_w^{p_n}(r) \leq \sum_{r \rightsquigarrow r_i} \tau_w^{p_{n-1}}(r)$.

Theorem 1 *Given a program p, Algorithm 3 produces a program p' such that $p' \equiv p$ and $\tau_w^{p'} \leq \tau_w^p$ if all memory operations are kept in program order at execution time.*

Proof 3 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 $\pi_{s'}$ for a block $s' = \mathscr{S}(r_j)$ will be inserted at point r_i, r_{i+1} of program p_{n-1}, thereby generating a program p_n with n prefetches. From Equations 4.2 and 4.3, we can write:

$$\tau_w^{p_{n-1}} = \sum_{bb \in B} t_w^{p_{n-1}}(bb) \times n_{bb}^w = \sum_{r \in R} \tau_w^{p_{n-1}}(r),$$
$$\tau_w^{p_n} = \sum_{bb \in B} t_w^{p_n}(bb) \times n_{bb}^w = \sum_{r \in R} \tau_w^{p_n}(r),$$

which can be rewritten, with the help of Equation 5.1, as follows:

$$\tau_w^{p_{n-1}} = \sum_{r \rightsquigarrow r_i} \tau_w^{p_{n-1}}(r) + \tau_w^{p_{n-1}}(r_{i+1}, r_j) + \sum_{r_{j+1} \rightsquigarrow r} \tau_w^{p_{n-1}}(r)$$

$$\tau_w^{p_n} = \sum_{r \rightsquigarrow r_i} \tau_w^{p_n}(r) + \tau_w^{p_n}(\pi_{s'}, r_j) + \sum_{r_{j+1} \rightsquigarrow r} \tau_w^{p_n}(r)$$

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:

$$\sum_{r \rightsquigarrow r_i} \tau_w^{p_n}(r) + \sum_{r_{j+1} \rightsquigarrow r} \tau_w^{p_n}(r) \leq \sum_{r \rightsquigarrow r_i} \tau_w^{p_{n-1}}(r) + \sum_{r_{j+1} \rightsquigarrow r} \tau_w^{p_{n-1}}(r)$$

Therefore, we conclude that:

$$\tau_w^{p_n} - \tau_w^{p_n}(\pi_{s'}, r_j) \leq \tau_w^{p_{n-1}} - \tau_w^{p_{n-1}}(r_{i+1}, r_j) \Leftrightarrow \tau_w^{p_n} \leq \tau_w^{p_{n-1}} - K,$$

where $K = \tau_w^{p_{n-1}}(r_{i+1}, r_j) - \tau_w^{p_n}(\pi_{s'}, r_j)$.

Since p_n and p_{n-1} are indistinguishable except for $\pi_{s'}$ and we know from Lemma 1 that $K > 0$, we conclude that $p_n \equiv p_{n-1}$ and $\tau_w^{p_n} \leq \tau_w^{p_{n-1}}$ hold for any integer $n > 1$, i.e. $p' \equiv p$ and $\tau_w^{p'} \leq \tau_w^p$ hold for any program

$p' \neq p$ 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 $\tau_w^{p'} = \tau_w^p$. Thus, $p' \equiv p$ and $\tau_w^{p'} \leq \tau_w^p$ hold for any program p' produced by Algorithm 3 from p.

BIBLIOGRAPHY

AGARWAL, D. et al. Transferring performance gain from software prefetching to energy reduction. In: IEEE. *Proc. of Int. Symp. on Circuits and Systems*. [S.l.], 2004. v. 2, p. 241–244.

APARICIO, L. C. et al. Combining prefetch with instruction cache locking in multitasking real-time systems. In: IEEE. *Proc. of IEEE Int. Conf. Embedded and Real-Time Computing Systems and Applications*. [S.l.], 2010. p. 319–328.

BERKELAAR, M.; EIKLAND, K.; NOTEBAERT, P. *lp_solve 5.5, Open Source (Mixed-Integer) Linear Programming System*. 2004. Software.

BERTOGNA, M. et al. Preemption points placement for sporadic task sets. In: *Proc. of Euromicro Conference on Real-Time Systems*. [S.l.: s.n.], 2010. p. 251–260. ISSN 1068-3070.

BINKERT, N. et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 39, n. 2, p. 1–7, ago. 2011. ISSN 0163-5964. Disponível em: <http://doi.acm.org/10.1145/2024716.2024718>.

BURGUIERE, C.; ROCHANGE, C. A contribution to branch prediction modeling in WCET analysis. In: *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*. Washington, DC, USA: IEEE Computer Society, 2005. (DATE '05), p. 612–617. ISBN 0-7695-2288-2. Disponível em: <http://dx.doi.org/10.1109/DATE.2005.7>.

CHEN, G. et al. Dynamic Scratch-pad Memory Management for Irregular Array Access Patterns. *Proc. of Conf. on Design, Automation and Test in Europe*, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, p. 931–936, 2006.

COUSOT, P.; COUSOT, R. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *Proc. of ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, p. 238–252, 1977.

DALLY, W. J. et al. Efficient embedded computing. *IEEE Computer Magazine*, v. 41, n. 7, p. 27–32, July 2008. ISSN 0018-9162.

DING, H.; LIANG, Y.; MITRA, T. WCET-centric partial instruction cache locking. In: *Proc. of IEEE/ACM Design Automation Conference*. New York, NY, USA: ACM, 2012. (DAC '12), p. 412–420. ISBN 978-1-4503-1199-1. Disponível em: <http://doi.acm.org/10.1145/2228360.2228434>.

DING, H.; LIANG, Y.; MITRA, T. Integrated instruction cache analysis and locking in multitasking real-time systems. In: *Proceedings of the 50th Annual Design Automation Conference*. New York, NY, USA: ACM, 2013. (DAC '13), p. 147:1–147:10. ISBN 978-1-4503-2071-9. Disponível em: <http://doi.acm.org/10.1145/2463209.2488916>.

DING, H.; LIANG, Y.; MITRA, T. WCET-centric dynamic instruction cache locking. In: *Proceedings of the Conference on Design, Automation & Test in Europe*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2014. (DATE '14), p. 27:1–27:6. ISBN 978-3-9815370-2-4. Disponível em: <http://dl.acm.org/citation.cfm?id=2616606.2616639>.

DING, Y.; YAN, J.; ZHANG, W. Optimizing instruction prefetching to improve worst-case performance for real-time applications. *Journal of Computing Science and Engineering*, v. 3, n. 1, p. 59–71, 2009.

FALK, H. WCET-aware register allocation based on graph coloring. In: *Proc. of IEEE/ACM Design Automation Conference*. [S.l.: s.n.], 2009. p. 726–731.

FALK, H.; KLEINSORGE, J. C. Optimal static WCET-aware scratchpad allocation of program code. In: *Proc. of IEEE/ACM Design Automation Conference*. [S.l.: s.n.], 2009. p. 732–737.

FALK, H.; PLAZAR, S.; THEILING, H. Compile-time decided instruction cache locking using worst-case execution paths. In: *Proc. of IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. New York, NY, USA: ACM, 2007. (CODES+ISSS '07), p. 143–148. ISBN 978-1-59593-824-4. Disponível em: <http://doi.acm.org/10.1145/1289816.1289853>.

FERDINAND, C. et al. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, v. 35, n. 2-3, p. 163–189, 1999.

GUPTA, R.; CHI, C.-H. Improving instruction cache behavior by reducing cache pollution. In: IEEE COMPUTER SOCIETY PRESS. *Proc. of ACM/IEEE Conference on Supercomputing*. [S.l.], 1990. p. 82–91.

GUSTAFSSON, J. et al. The mälardalen WCET benchmarks–past, present and future. In: *Proc. of Int. Workshop on Worst-Case Execution Time Analysis*. [S.l.: s.n.], 2010. p. 137–147.

GUTHAUS, M. e. a. MiBench: A free, commercially representative embedded benchmark suite. *Proc. of International Workshop on Workload*, v. 131, p. 3–14, 2001.

HWU, W. W.; CHANG, P. P. Achieving high instruction cache performance with an optimizing compiler. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 17, n. 3, p. 242–251, abr. 1989. ISSN 0163-5964. Disponível em: <http://doi.acm.org/10.1145/74926.74953>.

LI, Y.-T. S.; MALIK, S. Performance analysis of embedded software using implicit path enumeration. In: *Proc. of IEEE/ACM Design Automation Conference*. [S.l.: s.n.], 1995. p. 456–461.

LI, Y.-T. S.; MALIK, S.; WOLFE, A. Efficient microarchitecture modeling and path analysis for real-time software. In: *Proc. of IEEE Real-Time Systems Symposium*. [S.l.: s.n.], 1995. p. 298–307.

LIU, T.; LI, M.; XUE, C. J. Minimizing WCET for real-time embedded systems via static instruction cache locking. In: *Proceedings of the 2009 15th IEEE Symposium on Real-Time and Embedded Technology and Applications*. Washington, DC, USA: IEEE Computer Society, 2009. (RTAS '09), p. 35–44. ISBN 978-0-7695-3636-1. Disponível em: <http://dx.doi.org/10.1109/RTAS.2009.11>.

LUK, C.-K.; MOWRY, T. C. Compiler-based prefetching for recursive data structures. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 30, n. 5, p. 222–233, set. 1996. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/248208.237190>.

LUK, C.-K.; MOWRY, T. C. Architectural and compiler support for effective instruction prefetching: a cooperative approach. *ACM Transactions on Computer Systems*, v. 19, n. 1, p. 71–109, 2001. ISSN 0734-2071.

MCFARLING, S. Program optimization for instruction caches. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 17, n. 2, p. 183–191, abr. 1989. ISSN 0163-5964. Disponível em: <http://doi.acm.org/10.1145/68182.68200>.

MUCHNICK, S. S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997. ISBN 9781558603202. Disponível em: <http://books.google.com.br/books?id=Pq7pHwG1_OkC>.

MURALIMANOHAR, N.; BALASUBRAMONIAN, R.; JOUPPI, N. CACTI 6.0: A Tool to Understand Large Caches. *HP Research Report*, Citeseer, 2007.

PIERCE, J.; MUDGE, T. Wrong-path instruction prefetching. In: PUBLISHED BY THE IEEE COMPUTER SOCIETY. *Proc. of ACM/IEEE Int. Symposium on Microarchitecture*. [S.l.], 1996. p. 165–175.

PLAZAR, S. et al. WCET-aware static locking of instruction caches. In: *Proc. of ACM Int. Symposium on Code Generation and Optimization*. New York, NY, USA: ACM, 2012. (CGO '12), p. 44–52. ISBN 978-1-4503-1206-6. Disponível em: <http://doi.acm.org/10.1145/2259016.2259023>.

PUAUT, I. Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems. In: *Proc. of Int. Workshop on Worst-Case Execution Time Analysis*. [S.l.]: Citeseer, 2002.

PUAUT, I. WCET-centric software-controlled instruction caches for hard real-time systems. In: IEEE. *Proceedings of the 18th Euromicro Conference on Real-Time Systems*. Washington, DC, USA: IEEE Computer Society, 2006. (ECRTS '06), p. 217–226. ISBN 0-7695-2619-5. Disponível em: <http://dx.doi.org/10.1109/ECRTS.2006.32>.

PUAUT, I.; ARNAUD, A. Dynamic instruction cache locking in hard real-time systems. *Proc. of the 14th Int. Conference on Real-Time and Network Systems*, 2006.

RODRÍGUEZ, S.; JACOB, B. L. Energy/power breakdown of pipelined nanometer caches. In: ACM. *Proc. of ACM Int. Symp. on Low Power Electronics and Design*. [S.l.], 2006. p. 25–30.

SMITH, A. J. Sequential program prefetching in memory hierarchies. *IEEE Computer Magazine*, IEEE, v. 11, n. 12, p. 7–21, 1978. ISSN 0018-9162.

SMITH, J.; HSU, W. Prefetching in supercomputer instruction caches. In: IEEE COMPUTER SOCIETY PRESS. *Proc. of ACM/IEEE Conference on Supercomputing*. [S.l.], 1992. p. 588–597.

STALLMAN, R. Gnu compiler collection internals. *Free Software Foundation*, 2010. Disponível em: <http://gcc.gnu.org/onlinedocs/gccint/>.

TANG, J. et al. Prefetching in embedded mobile systems can be energy-efficient. *IEEE Computer Architecture Letters*, IEEE, v. 10, n. 1, p. 8–11, 2011. ISSN 1556-6056.

THEILING, H.; FERDINAND, C.; WILHELM, R. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, Springer, v. 18, n. 2/3, p. 157–179, 2000. ISSN 0922-6443.

UDAYAKUMARAN, S.; DOMINGUEZ, A.; BARUA, R. Dynamic Allocation for Scratch-pad Memory Using Compile-time Decisions. *ACM Transactions on Embedded Comp. Systems*, New York, NY, USA, v. 5, n. 2, p. 472–511, 2006. ISSN 1539-9087.

VERMA, M.; MARWEDEL, P. *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*. [S.l.]: Springer Verlag, 2007. ISBN 978-1-4020-5896-7.

WILHELM, R. et al. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, ACM, New York, NY, USA, v. 7, n. 3, p. 1–53, maio 2008. ISSN 1539-9087. Disponível em: <http://doi.acm.org/10.1145/1347375.1347389>.

WILTON, S.; JOUPPI, N. CACTI: an enhanced cache access and cycle time model. *Solid-State Circuits, IEEE Journal of*, v. 31, n. 5, p. 677–688, May 1996. ISSN 0018-9200.

WUERGES, E.; OLIVEIRA, R.; SANTOS, L. Fast estimation of memory consumption for energy-efficient compilers. In: *IEEE International Conference on Electronics, Circuits and Systems*. [S.l.: s.n.], 2011. p. 719–722.

WUERGES, E.; OLIVEIRA, R. S. de; SANTOS, L. C. V. dos. Reconciling real-time guarantees and energy efficiency through unlocked-cache prefetching. In: *Proceedings of the 50th Annual Design Automation Conference*. New York, NY, USA: ACM, 2013. (DAC '13), p. 146:1–146:9. ISBN 978-1-4503-2071-9. Disponível em: <http://doi.acm.org/10.1145/2463209.2488915>.

YAN, J.; ZHANG, W. WCET analysis of instruction caches with prefetching. In: ACM. *Proc. of ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems*. [S.l.], 2007. p. 175–184.