UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MARCO ANTONIO ZANATA ALVES

Increasing Energy Efficiency of Processor Caches via Line Usage Predictors

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Prof. Dr. Philippe O. A. Navaux
Advisor

Porto Alegre, 2014

CIP – CATALOGING-IN-PUBLICATION

Alves, Marco Antonio Zanata

Increasing Energy Efficiency of Processor Caches via Line Usage Predictors / Marco Antonio Zanata Alves. – Porto Alegre: PPGC da UFRGS, 2014.

116 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2014. Advisor: Philippe O. A. Navaux.

1. Line Usage Predictors. 2. Sub-block Usage Predictors. 3. Replacement Policy. 4. Early Write-Back. 5. Cache Memories. 6. Energy Efficient. I. Navaux, Philippe O. A.. II. Título.

Dedicated to my parents and my beloved wife.

ACKNOWLEDGEMENTS

To all my family, friends, collaborators and advisors a sincere thank you.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AHT	Access History Table
AIP	Access Interval Predictor
AL	Added Latency for column accesses
BBV	Basic Block Vector
BTB	Branch Target Buffer
CAS	Column Address Strobe
CCD	Column to Column Delay
CMP	Chip Multiprocessor
CMT	Chip Multithreading
CWD	Column Write Delay
DBI	Dynamic Binary Instrumentation
DDR	Double Data Rate
DEWP	Dead Line and Early Write-Back Predictor
DRAM	Dynamic Random Access Memory
DSBP	Dead Sub-Block Predictor
FAW	Four row Activation Window
FCFS	First-Come First-Serve
GAg	Global Adaptive branch prediction using one Global PHT
GAs	Global Adaptive branch prediction using per-Set PHT
GCC	GNU Compiler Collection
GEMS	General Execution-driven Multiprocessor Simulator

ILP	Instruction Level Parallelism
IPC	Instructions per Cycle
ISA	Instruction Set Architecture
KIPS	Kilo Instructions per Second
LFSR	Linear Feedback Shift Register
LLC	Last-Level Cache
LRU	Least Recently Used
LTP	Last-Touch Predictor
LvP	Live-time Predictor
LWP	Last Write Predictor
McPAT	Multi-core Power, Area, and Timing
MOB	Memory Order Buffer
MPKI	Misses per Kilo Instructions
MSHR	Miss-Status Handling Registers
NAS	Numerical Aerodynamic Simulation
NoC	Network-on-Chip
NUCA	Non-Uniform Cache Architecture
NUMA	Non-Uniform Memory Access
OoO	Out-of-Order
OpenMP	Open Multi-Processing
OS	Operating System
PAg	Per-address Adaptive branch prediction using one Global PHT
PAs	Per-address Adaptive branch prediction using per-Set PHT
PC	Program Counter
PCM	Performance Counter Monitor
PHT	Pattern History Table
RAPL	Running Average Power Limit
RAS	Row Address Strobe

RAT Registers Alias Table

RC Row Cycle

RCD RAS to CAS Delay

ROB Reorder Buffer

RP Row Precharge

RRD Row to Row activation Delay

RTP Read To Precharge

SDP Skewed Dead-Block Predictor

SFP Spatial Footprint Predictor

SiNUCA Simulator of Non-Uniform Cache Architectures

SMT Simultaneous Multi-Threading

SPP Spatial Pattern Predictor

SSV Search Set Vector

TLB Translation Look-aside Buffer

TLP Thread Level Parallelism

VWQ Virtual Write Queue

WR Write To Read delay time

WTR Write Recovery time

CONTENTS

Increasing Energy Efficiency of Processor Caches via Line Usage Predictors

ABSTRACT

Energy consumption is becoming more important for processor architectures, where the number of cores inside the chip is increasing and the total power budget is kept at the same level or even reduced. Thus, energy saving techniques such as frequency scaling options and automatic shutdown of sub-systems are being used to maintain the trade-off between power and performance. To deliver high performance, current Chip Multiprocessors (CMPs) integrate large caches in order to reduce the average memory access latency by allocating the applications' working set on-chip. These cache memories have traditionally been designed to exploit temporal locality by using smart replacement policies, and spatial locality by fetching entire cache lines from memory on a cache miss.

However, recent studies have shown that the number of sub-blocks within a line that are actually used is often low, and those sub-blocks that are used are accessed only a few times before becoming dead (that is, never accessed again). Additionally, many of the cache lines remain powered for a long period of time even if the data is not used again, or is invalid. For modified cache lines, the cache memory waits until the line is evicted to perform the write-back to next memory level. These write-backs compete with read requests (processor demand and cache prefetch), increasing the pressure on the memory controller. For these reasons, the energy efficiency and performance of cache memories are not ideal.

This thesis introduces cache line usage predictors to increase the energy efficiency of cache memories. We propose the Dead Sub-Block Predictor (DSBP) and Dead Line and Early Write-Back Predictor (DEWP) mechanisms to enable energy savings without performance degradation. DSBP is used to predict which sub-blocks of a cache line will be actually accessed and how many times they will be used in order to bring into the cache only those sub-blocks that are necessary, and power them off after they are accessed the predicted number of times. DEWP predicts dead lines as soon as they receive the last access, and turns off these lines. Dirty lines are scheduled for write-back after the last write operation occurs, increasing the energy savings potential and also reducing the pressure on the memory controller. Both proposed mechanisms also reduce pollution in cache memories by prioritizing dead lines for eviction in the existing replacement policy.

Although each introduced mechanism is capable of performing separately inside a system, both mechanisms can also be mixed in the same cache hierarchy. This mixed implementation is interesting because the sub-block granularity is more suitable for cache levels closer to the processor, where the cache lines are quickly evicted, while the Last-Level Cache (LLC) tends to use the whole cache line before its eviction.

In order to evaluate our proposed mechanisms, we introduce the Simulator of Non-Uniform Cache Architectures (SiNUCA). This cycle-accurate microarchitecture simulator is validated in terms of performance and energy consumption by comparing it to a real processor. Our performance results were obtained executing single-threaded applications from SPEC-CPU2006 and multi-threaded applications from SPEC-OMP2001 and NAS-NPB benchmark suites. The energy related results were obtained by integrating SiNUCA with the Multi-core Power, Area, and Timing (McPAT) framework and the CACTI power modeling tool.

When applying our mechanisms on all the cache levels, we observe on average a 36% energy reduction for DSBP, 25% energy reduction using DEWP and an average reduction of 37% in the energy consumption applying DSBP on L1 and L2 and DEWP on the LLC. All these reductions caused a negligible performance loss of less than 4% on average.

Keywords: Line Usage Predictors, Sub-block Usage Predictors, Replacement Policy, Early Write-Back, Cache Memories, Energy Efficient.

Aumentando a Eficiência Energética da Memória Cache de Processadores através de Preditores de Uso de Linhas da Cache

RESUMO

O consumo de energia se torna cada vez mais importante para a arquitetura de processadores, onde o número de *cores* dentro de um mesmo *chip* está aumentando mas o total de energia disponível se mantém no mesmo nível ou até mesmo se reduz. Assim, técnicas para economizar energia, tais como opções de escala de frequência e desligamento automático de subsistemas, estão sendo usadas para manter a troca entre energia e desempenho. Para se obter alto desempenho, os atuais *Chip Multiprocessors (CMPs)* integram grandes memórias *cache* a fim de reduzir a latência média para acesso a memória principal, através da alocação do conjunto de dados da aplicação dentro do *chip*. Essas memórias *cache* tem sido projetadas tradicionalmente para explorar a localidade temporal usando políticas de substituição inteligentes e localidade espacial buscando todos os dados da linha da *cache* após uma falta de dados.

Entretanto, estudos recentes mostraram que o número de sub-blocos dentro da linha da memória *cache*, que são realmente usados, costuma ser baixo, sendo que, os sub-blocos que são usados recebem poucos acessos antes de se tornarem mortos (isto é, nunca mais são acessados). Além disso, muitas da linhas da memória *cache* permanecem ligadas por longos períodos de tempo, mesmo que os dados não sejam usados novamente ou são inválidos. Para linhas de *cache* modificadas, a memória *cache* aguarda até que a linha seja expulsa para que esta seja gravada (*write-back*) de volta no próximo nível de memória. Essas escritas competem com as requisições de leitura (demanda do processador e pré-busca da *cache*), aumentando a pressão no controlador de memória. Por essas razões, a eficiência energética e o desempenho das memórias *cache* não são ideais.

Essa tese propõe a aplicação de preditores de uso de linhas da *cache* para aumentar a eficiência energética das memórias *cache*. São propostos os mecanismos *Dead Sub-Block Predictor (DSBP)* e *Dead Line and Early Write-Back Predictor (DEWP)* para permitir economia de energia sem que haja degradação do desempenho. DSBP é usado para prever quais sub-blocos da linha da *cache* serão usados e quantas vezes eles serão acessados de forma a trazer para a *cache* apenas os sub-blocos úteis e desliga-los após eles serem acessados pelo número de vezes previsto. *DEWP* prevê linhas de *cache* mortas assim que elas recebem o último acesso, desligando essas linhas. As linhas sujas são escalonadas para sofrerem *write-back* após a última operação de escrita, aumentando o potencial de salvar energia, reduzindo também a pressão no controlador de memória. Ambos os mecanismos propostos também reduzem a poluição nas memórias *cache*, dando prioridade para a expulsão de linhas mortas, melhorando as atuais políticas de substituição.

Embora cada mecanismo apresentado seja capaz de funcionar separadamente dentro do sistema, ambos os mecanismos podem também ser misturados em uma mesma hierarquia de *cache*. Essa implementação mista é interessante pois a granularidade de sub-bloco é preferível para níveis de *cache* próximos do processador, onde as linhas de memória *ca-*

che são expulsas rapidamente, enquanto o último nível de *cache* tende a usar toda a linha antes da sua expulsão.

Com o intuito de avaliar os mecanismos propostos, é apresentado o *Simulator of Non-Uniform Cache Architectures (SiNUCA)*. Esse simulador de microarquitetura com precisão de ciclos é validado em termos de desempenho e consumo de energia através da comparação com um processador real.

Os resultados de desempenho foram obtidos executando aplicações das cargas de trabalho *single-threaded* do conjunto SPEC-CPU2006 e aplicações *multi-threaded* dos conjuntos SPEC-OMP2001 e NAS-NPB. Os resultados relativos a energia foram obtidos integrando o *SiNUCA* com as ferramentas de modelagem *Multi-core Power, Area, and Timing (McPAT)* e CACTI.

Quando aplicados os mecanismos em todos os níveis de memória *cache*, observou-se em média uma redução de 36% no consumo de energia usando o *DSBP*, 25% usando o *DEWP* e 37% quando usou-se o *DSBP* nos níveis L1 e L2 e o *DEWP* no último nível. Todas essas reduções causaram uma perda desprezível de desempenho de menos de 4% em média.

Palavras-chave: Preditores de Linha da Cache, Memórias Cache, Eficiência Energética.

1 INTRODUCTION

High performance computing uses aggressive techniques to obtain parallelism in multiple levels: at the instruction level using superscalar pipelines with Out-of-Order (OoO) execution (SMITH; SOHI, 1995), larger OoO execution windows, frequency increase, and others (HENNESSY; PATTERSON, 2007) such as better branch predictors and data prefetchers; at the thread and process level using Chip Multithreading (CMT) techniques with Chip Multiprocessor (CMP).

However, Instruction Level Parallelism (ILP) techniques have less room for improvements due to limits imposed by wire-delay problems (clock wall), power consumption limit (power wall) (BORKAR, 1999), and ILP extraction problems (ILP wall) (AGARWAL et al., 2000). Thus, Thread Level Parallelism (TLP) techniques such as CMP with increasing number of cores have become the most likely way for the industry to continue delivering more powerful processors on each new generation.

Most of the multi-core processors are built with cores simpler than traditional single cores, with shorter pipelines and simpler control structures (OLUKOTUN et al., 1996). This leads to the possibility of integrating more cores on the same physical silicon area. This complexity reduction on the cores is also beneficial for the power constraints inside the chip. For these reasons, the focus has changed to also explore the thread and process parallelism (UNGERER; ROBIC; SILC, 2002).

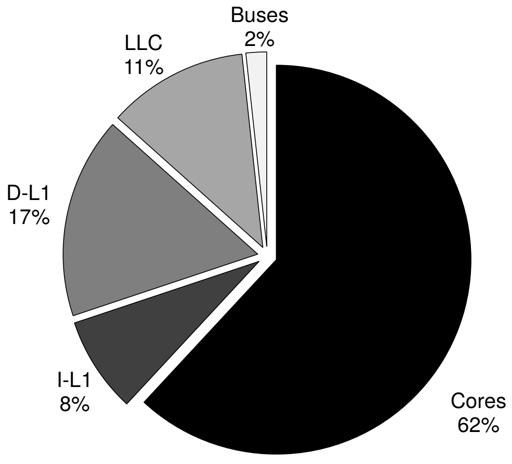
Recently, energy efficiency has become a key design parameter in computer architectures. While the number of transistors on a chip has been increasing rapidly, the total power budget remains at the same level or even decreases. In this context, each technique that leads to reduction of power consumption has a great impact on the total energy consumption of the system. Thus, new processors begin to present frequency scaling options, automatic shutdown of sub-systems and other techniques to ensure that machines will reduce the unnecessary energy consumption, and will consume full power only when demanded.

1.1 The Problem

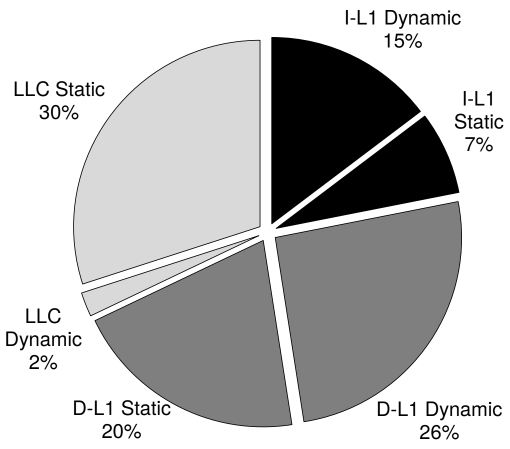
Cache memories, while key to high performance, consume a significant fraction of total chip power (LI et al., 2009), varying from 15% to 23% of the total chip power. As such, designing energy efficient processors starts with efficient design of such power-hungry components.

In order to evaluate the cache energy consumption inside the chip, we executed the SPEC-CPU2006 single threaded benchmark suite and the SPEC-OMP2001 and NAS-NPB parallel benchmark suites modeling two different processors using the *McPAT* frame-

Figure 1.1: Breakdown of Core 2 Duo energy consumption, executing SPEC-CPU2006, SPEC-OMP2001 and NAS-NPB benchmark suites.

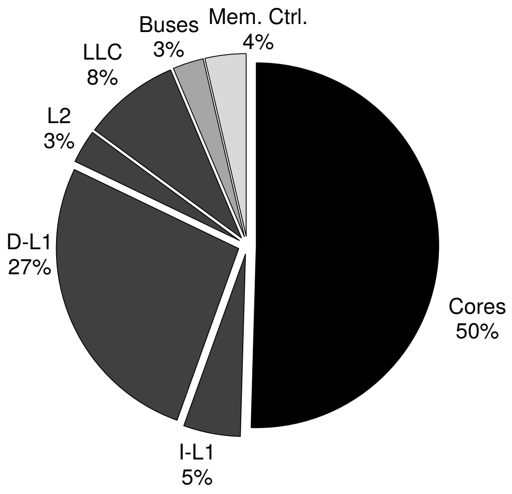


(a) Total chip energy consumption.

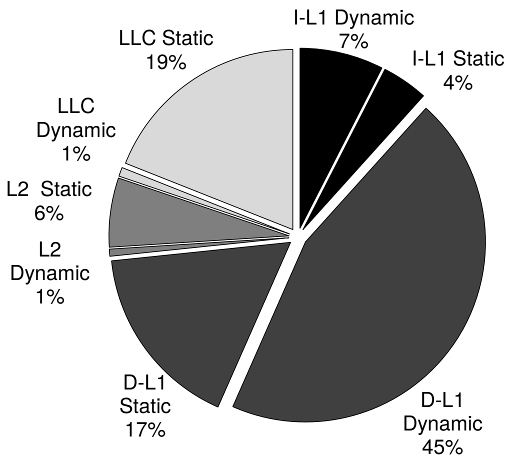


(b) Cache memory energy consumption.

Figure 1.2: Breakdown of Sandy Bridge energy consumption, executing SPEC-CPU2006, SPEC-OMP2001 and NAS-NPB benchmark suites.



(a) Total chip energy consumption.



(b) Cache memory energy consumption.

work (LI et al., 2009). The parallel benchmarks execute with the same number of threads as the number of cores available in the processor.

Figure 1.1 shows the Intel Core 2 Duo (Conroe microarchitecture (BOJAN et al., 2007)) energy consumption. The average power consumed by the chip corresponds to 22 W according to our energy model. It can be observed that 36% of energy consumed inside the chip is consumed by the cache memories, while each core (2 cores in total) is responsible for 31% of the energy consumed by the chip (62% in total).

Figure 1.2 shows the Intel Xeon (Sandy Bridge microarchitecture (YUFFE et al., 2011)) energy consumption. The average power consumed by the chip corresponds to 37 W. It can be observed that 43% of energy consumed inside the chip is consumed by the cache memories, while each core (8 cores in total) is responsible for 6% of the energy consumed by the chip (50% in total).

Due to the reduced power budget, and the increasing portion of energy being consumed by the cache memories, mechanisms to improve the cache efficiency are becoming more important. Moreover, as we can see in Figures 1.2(b),1.3(b) the major sources of energy consumption inside the cache sub-system are related to the L1 dynamic and static energy, and the Last-Level Cache (LLC) static energy, which can be explained by to the high number of L1 operations and to the large area occupied by the *LLC*.

For today's processors with a fixed cache line size, energy inefficiency can occur on two levels: 1) on a cache line level where a line is kept alive much longer than necessary, and 2) on a sub-block level, when parts of a cache line that will never be used are brought into the cache, and also when active sub-blocks become dead after a few accesses but are kept alive until the line is evicted.

Besides the energy consumption problem, keeping cache lines that will not be used anymore (dead lines) inside the cache, increases cache pollution and memory contention. Cache pollution increases when the replacement policy takes wrong decisions by removing alive blocks instead of already dead blocks. Pollution can also increase the number of cache misses, thus generating negative impact on the performance of the system. The impact on memory contention happens when the cache keeps dirty lines which already received the last write. By doing so, these lines will only suffer write-back to the main memory whenever the cache line is evicted. However, considering that memory accesses occur in bursts (WANG; KHAN; JIMÉNEZ, 2012a), the write-back operation can increase the memory pressure in those moments when lots of data are being requested (burst of operations).

1.2 Motivation

In order to evaluate the inefficiency of current cache memories, we measured the average usage of L1 cache lines in the sub-block granularity. Figure 4.2 using SPEC-CPU2006, SPEC-OMP2001 and NAS-NPB shows that on average only 57% of a line is used by the single threaded benchmarks and 81% for multi-threaded benchmarks. Similar results were observed in (KUMAR; WILKERSON, 1998; CHEN et al., 2004; KHARBUTLI; SOLIHIN, 2008). We make the new observation (Figure 4.3) that 83% of sub-blocks become dead after less than four accesses.

Measuring the average time between the last access to a cache line and the line eviction from the cache memory, we can evaluate the inefficiency in the cache line granularity. On average the time between the last access to a cache line and its eviction is 40% of the lifetime of the line (KAXIRAS; HU; MARTONOSI, 2001).

These results show that there are several opportunities for energy savings in traditional cache designs. Cache lines are unused for long periods of time while still consuming energy. Furthermore, the L1 and L2 caches present more inefficiency on the sub-block level, while the LLC uses the whole line more consistently, but the line is still powered on for a long time without accesses.

Based on the Sandy Bridge processor, we show in Section 4.1 that an oracle mechanism that perfectly predicts the dead sub-blocks inside the L1 cache and turns-off those sub-blocks can save on average 51% of the L1 cache energy consumption, that represents savings of 34% for the entire cache sub-system energy consumption.

Another oracle mechanism, presented in Section 5.1, that turns off LLC lines after their last access or when the cache line becomes invalid, generates LLC energy savings of 65% on average, that represents savings of 11% for the entire cache sub-system energy consumption.

Considering the high theoretical energy savings achieved by cache line usage predictors, at the sub-block or cache line granularities, the benefits of such mechanisms inside the cache memory to increase its energy efficiency are clear.

1.3 Hypotheses and Objectives

In order to increase the energy efficiency of cache memories, we formulate the following hypotheses:

- By studying the historic memory access behavior of the application, it is possible to predict the cache line usage.
- Using the predicted cache line usage, dead lines can be identified and turned off in order to save energy and reduce the cache pollution and memory pressure.
- Looking at the sub-block granularity allows to further increase the energy savings.

Based on these hypotheses, the main goal of this thesis is to introduce mechanisms to increase the energy efficiency of the cache memories. This objective will be achieved by the following steps:

- We propose a Dead Sub-Block Predictor (DSBP) to predict at run-time the cache line usage at the sub-block granularity. This mechanism will be used to store only the useful sub-blocks inside the cache line and to turn-off the sub-blocks when they become dead.
- We propose a Dead Line and Early Write-Back Predictor (DEWP) to detect when a cache line receives its last read and last write. This predictor will be used to early write-back dirty lines as soon as they receive their last access and to turn-off the line as it receives its last read.
- To evaluate the new mechanisms, we developed the Simulator of Non-Uniform Cache Architectures (SiNUCA), a new cycle accurate, trace-driven simulator composed of the following main components: processor, cache memories, interconnections and memory system. This simulator is able to simulate multi-core systems, with multi-banked caches and Network-on-Chip (NoC) interconnections. This simulator is validated with a real machine in terms of performance and energy consumption.

DSBP and DEWP overlap in functionalities, however, the sub-block granularity fits better for small caches that are closer to the processor while the line granularity fits better for big LLC memories. The smaller caches have a lower sub-block usage compared to the cache levels closer to the main memory. The reason for this low usage is that the smaller caches tend to evict the cache lines sooner, reducing the chances for full cache line usage.

The dead line predictors should also be used to improve the cache replacement policy in order to reduce cache pollution. For the cases where the cache lines are dirty, the last write predictor will help to reduce the memory pressure in the memory controller.

The general goal of this thesis is to design mechanisms that enable energy savings in the cache memory, maintaining the performance of the system. The overhead of such mechanisms will also be evaluated in order to show their benefits and possible drawbacks.

The simulator used for the evaluations will be mainly developed to verify our mechanisms, by providing a controlled environment capable to simulate the microarchitecture inside the cores, the cache memory sub-system with multi-banked caches, including the *NoC* interconnection and a detailed memory controller.

1.4 Contributions

The main contributions of this thesis are:

- DSBP – Dead Sub-Block Predictor (ALVES et al., 2012)

  Sub-block usage predictor: A mechanism to predict and turn on only the useful sub-blocks of each cache line.

  Dead sub-block predictor: A mechanism to predict when each sub-block inside a cache line becomes dead and turn off these dead sub-blocks.

  Earlier eviction of dead lines: A mechanism to improve the cache replacement algorithm. The sub-block predictor gives feedback to the replacement algorithm after all the sub-blocks become dead, marking dead lines as future victims for eviction.

- DEWP – Dead Line and Early Write-Back Predictor (ALVES et al., 2013)

  Last line read predictor: A mechanism that aims to save energy by turning off dead or invalid cache lines when the cache line receives a write in another cache.

  Last line write predictor: A mechanism to perform early write-back of dirty cache lines to main memory, since these lines will not be modified anymore.

  Earlier eviction of dead lines: A predictor to detect whenever a cache line receives its last access, prioritizing those lines for early eviction.

- SiNUCA – Simulator of Non-Uniform Cache Architectures

  Performance and energy validated simulator: Our simulator is validated in terms of performance and energy compared to real processor. It will be used to implement and evaluate our proposed mechanisms.

Compared to the related work, our mechanisms produce more accurate predictions achieving higher energy savings, with a negligible overhead that keeps the performance on the same level. DSBP is the first mechanism that is capable of predicting and turning off cache lines on the sub-block granularity, enabling dynamic and static cache energy savings. DEWP predicts last writes, saving static energy whenever the cache line becomes dead. Both mechanisms do not require broadcast signals for all the cache lines nor complex internal simulators to predict the cache line usage.

1.5 Document Organization

The remainder of this text is organized as follows. Background and related work are presented in Chapter 2. Chapter 3 gives a detailed overview of the new simulator, together with the workloads used in our evaluations. Chapter 4 presents the Dead Sub-Block Predictor. Chapter 5 introduces the Dead Line and Early Write-Back Predictor. Chapter 6 presents the analysis of a system implementing both predictors. Chapter 7 discuss the main conclusions of this thesis. Appendix A shows additional SiNUCA validation results.

2 ENERGY EFFICIENCY IN CACHE MEMORIES

This chapter introduces the concepts and the sources of inefficiency in cache memories. We also discuss the related work on cache usage predictors. The prior work related to the simulator is discussed in the next chapter.

2.1 Sources of Inefficiency in Cache Memories

Cache memories are energy and performance inefficient in cache line and sub-block granularity. In order to understand the sources of inefficiency some terms must be explained first:

Dead line: A cache line is considered being dead on the period of time between its last access (read or write) and its eviction from the cache memory.

Dead sub-block: In the same way as the dead line, a sub-block is dead after its last access. However, a sub-block may become dead before the line is declared dead. This is because some sub-block can be accessed separately from the others.

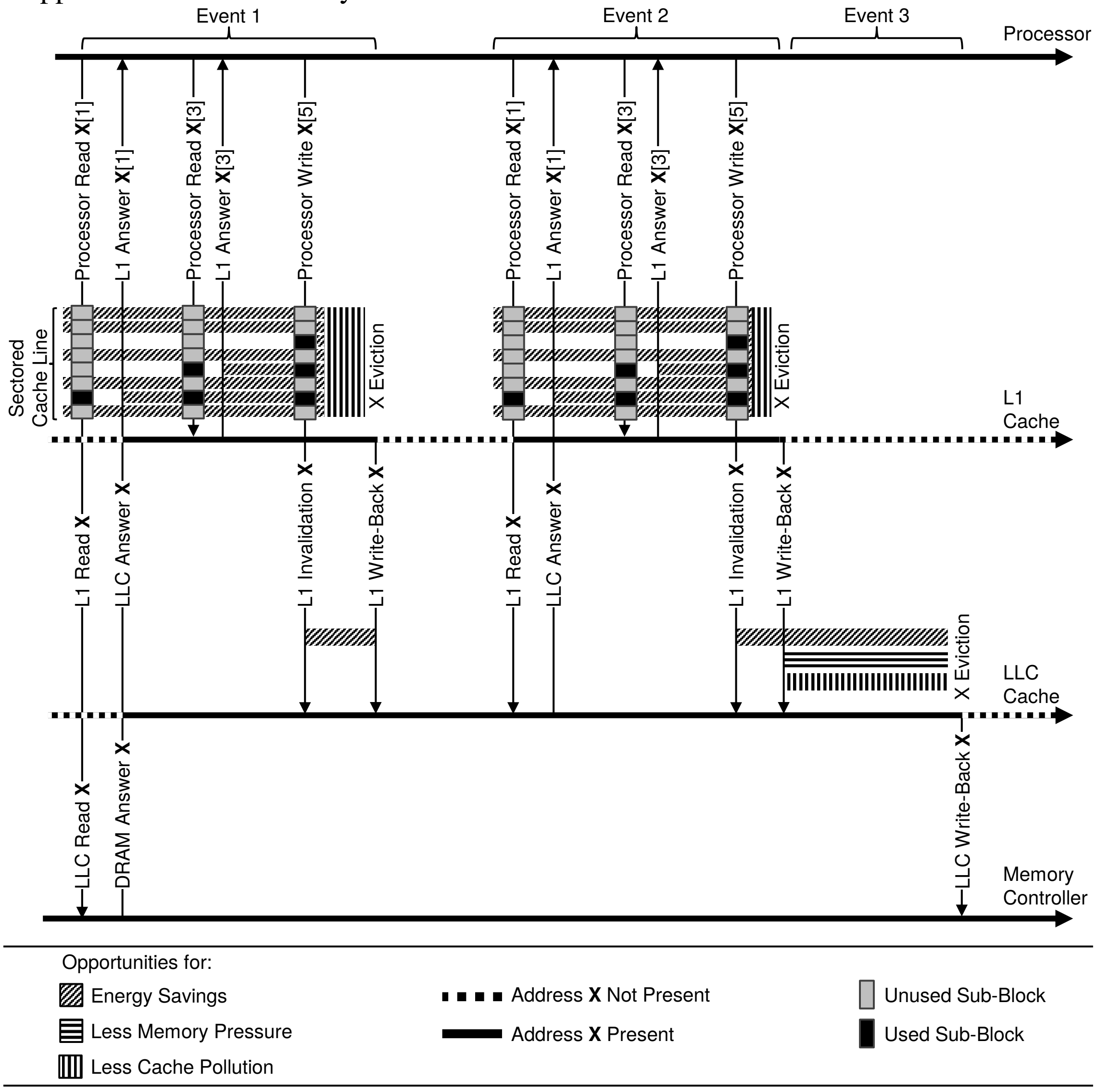
Cache pollution: The fact of having multiple cache lines, that the program will not use in a close future, residing inside the cache, causing other still useful lines to be prematurely evicted from the cache, is considered a cache pollution problem. This cache pollution problem is also called as cache noise by some authors.

Memory pressure: Considering that memory operations tends to occur in bursts of accesses (see Figure 5.2), the high number of requests into the main memory causes a memory pressure, which is even higher considering that dirty cache lines need to be written-back to the main memory before its complete eviction to make room for a new cache line. These write-back operations competing with the read requests into the memory controller is a source of a high memory pressure.

Figure 2.1 presents a scenario with several opportunities to increase the cache memory efficiency. This figure shows a series of memory accesses (reads and writes) from the processor to several addresses of cache lines (*X*, *Y* and *Z*) over the time. There are two cache levels. The L1 cache is a direct-mapped sectored cache, split into 8 sub-blocks of 8 bytes each. The LLC is a 2-way set associative without sub-blocks. For simplicity we assume that these addresses have the same index, which means that they get mapped to the same cache set. The two horizontal lines referent to lines *A* and *B* show two specific cache lines in the L1 cache and LLC. A solid line indicates that the line with address *X* is present and valid, while the dashed line indicates not present.

Several events are shown in the figure:

Figure 2.1: Opportunities for increasing the cache memory efficiency, modeling a direct-mapped L1 cache and 2-way set associative LLC.



Event 1: The processor reads the address *X*, requesting the sub-block 1. As the line is not present in the L1 cache and LLC, it needs to be requested from the memory controller. After one more read and write operations all the sub-blocks inside the cache line become dead due to the read of another line. Due to another cache accesses the address *X* is written-back to the LLC and then evicted from the L1 cache.

On the sub-block level (L1 cache), we can observe two opportunities to save static energy. First, when a never used sub-block is brought to the cache line, it could be turned off. Second, when a sub-block receives its last access and it becomes dead, it could also be turned off. Cache pollution could also be reduced by early evicting the cache line *X* after it receives its last access (all the sub-blocks are dead). Notice that the sub-block number *X[5]* needs to be turned on until the write operation, considering that the processor is capable of writing only one part within the sub-block.

On the cache line granularity (LLC), we have the opportunity to save energy by turning off the cache line during the time the line is invalid.

Event 2: Once again, the processor requests the address *X*, bringing a copy of this line from the LLC. We can observe the same opportunities to save energy and reduce cache pollution.

Event 3: At this point, the cache address *X* is evicted from the L1 cache and written-back to the LLC. After a certain period of time, this line *X* is evicted from the LLC, performing thus a write-back to the main memory of the dirty cache line *X*. Energy savings can be obtained by turning off the line *X* in the LLC when it is invalid and after it receives the write-back. We can also observe opportunities to reduce the memory pressure between the moment when the LLC receives the last write-back of the address *X* and it writes-back this line to the main memory. The opportunity to reduce cache pollution happens in the same time, when the line becomes dead and it could be marked for early eviction to make room for a new line.

During a L1 cache miss, the first operation performed by the cache is to access the whole cache line at the same time as the tag is obtained. The second operation is to compare the tag to the target memory address and forward the requested bytes in the same cycle (HUANG et al., 2001). The parallel access to the data and the tag arrays guarantees a low cache latency.

Besides the static energy on the sub-block level, we could also save dynamic energy during cache reads. If several sub-blocks are turned off, less bytes need to be accessed, which requires less dynamic energy.

The gains in terms of cache energy savings using an oracle line predictor can be easy modeled by computing the time when the line is invalid, as well as between the last access and the line eviction, subtracting the static energy spent on those cycles from the total energy consumption of the cache sub-system. For the oracle sub-block predictor, the static energy savings can be easily extended from the line predictor. The dynamic energy savings can be computed by analyzing the number of alive sub-blocks on every cache line access. Further modeling details are presented in Sections 4.3 and 5.3.

Figure 2.2: Overall cache energy consumption for the oracle line usage and oracle sub-block usage predictors.

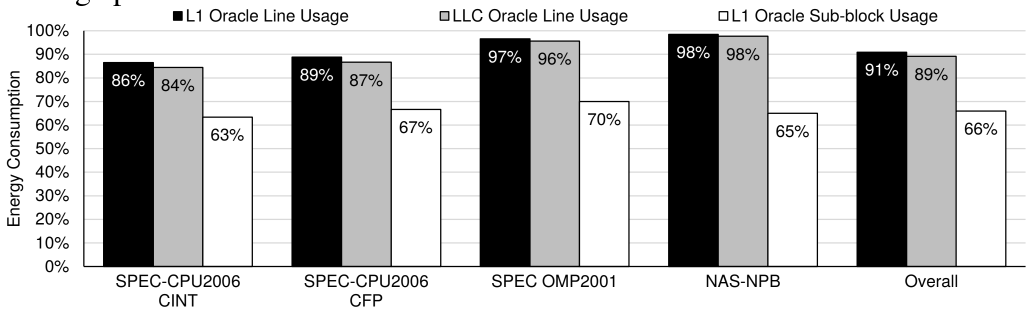


Figure 2.2 presents the overall energy consumption for the entire cache sub-system considering two different oracles (line usage and sub-block usage) applied to different cache levels (L1 and LLC), modeling a Sandy Bridge machine.

We can observe that energy savings are achievable on both the line and sub-block granularities. However, working on the sub-block basis, more opportunities for energy savings are present. The reason is that the applications tend to access the sub-blocks of line unevenly during the lifetime of the line, such that waiting for the whole line to become dead loses energy savings opportunities. The oracle mechanisms can also reduce the energy consumption of never used prefetched cache lines, reducing the impact of these lines on the total static energy consumption.

The possible gains in terms of performance are hard to be modeled, because any small difference in the replacement policy leads the cache to have a completely different line allocation.

To exploit these energy savings opportunities, it is necessary to create an accurate and low overhead mechanism capable of predicting line and sub-block usage. We propose two mechanisms to work in both sub-block and cache line granularity, in order to reduce the energy consumption, while using the performance opportunities to keep the final execution time at the same level.

2.2 Related work

Previous work has introduced line usage predictors, dead line predictors and last write predictors, which are applied to problems such as reducing static energy consumption, prefetching, cache pollution, memory controller contention among others.

This section discusses these mechanisms and compares them to our approaches. We also introduce the mechanism based on related work that will be compared to our predictors.

2.2.1 Line Usage Predictors

KUMAR; WILKERSON (1998) proposed a Spatial Footprint Predictor (SFP) which predicts the neighboring words to be prefetched on a cache miss. The goal is to reduce the L1 miss ratio while keeping the cache pollution at a minimum. Based on run-time history tables, the predictor explores spatial locality by using a cache with small lines but fetching multiple neighboring lines that are likely to be used in the near future.

CHEN et al. (2004) proposed a Spatial Pattern Predictor (SPP) to predict cache line usage patterns. The mechanism uses the Program Counter (PC) and the first data offset requested to correlate historical data about line usage to predict future usage patterns of L1 cache lines. The goal of this technique is to reduce static energy by bringing into the cache just those sub-blocks that were predicted to be accessed. The authors also introduce a prefetching technique to bring only the predicted spatial patterns for contiguous groups of up to 512 bytes.

PUJARA; AGGARWAL (2008) studied three types of mechanisms to predict the useless data in a cache block (cache noise), guided by the usage history of words in a cache line. Their results motivate the use of PC and offset as the index of a history table. In order to couple their mechanism with any prefetcher, the authors proposed an implementation considering that the pattern of the line being prefetched will be the same as the line which triggered the prefetcher.

Line usage predictors like SPP (CHEN et al., 2004) could be easily extended in order to start predicting dead sub-blocks. We extended SPP to predict useful and dead sub-blocks. However, our results show that DSBP performs on average 10% better in terms of energy reduction than the adapted SPP. Moreover, the adapted SPP increased the total cache misses on all the cache levels by 20% on average compared to the baseline without any predictor. This increase on total cache misses caused a 9% increase in the execution time.

The reason adapted SPP achieves poor results is that it uses an algorithm that resets the old pattern and starts a new one every time a new cache line comes into the cache so that new patterns can be learned quickly. However, this allows multiple Pattern History Table (PHT) pointers to the same PHT entry to simultaneously co-exist and therefore incorrect patterns are recorded more frequently.

2.2.2 Counter Based Dead Line Predictor

KHARBUTLI; SOLIHIN (2008) presented two counter-based mechanisms (AIP and LvP). Access Interval Predictor (AIP) counts the number of accesses to the set that has the line during the line's current access interval and identifies it as dead when the event count reaches the threshold. Live-time Predictor (LvP) records the number of accesses to a cache line and predicts the line as dead when the access counter reaches a certain threshold. The results show that LvP delivers higher accuracy with less complexity. The mechanism uses a hash of the PC which caused the cache miss to index into a table that stores the history of the number of accesses from previously evicted lines. The mechanism is used to identify dead lines and to evict them early, and also to bypass never re-accessed cache lines.

2.2.3 Trace Based Dead Line Predictors

LAI; FIDE; FALSAFI (2001) introduced the Last-Touch Predictor (LTP) which uses an execution trace to predict the last touch to a cache line. The key intuition behind LTP is that memory evictions are triggered by program instructions and that program behavior tends to repeat. The mechanism generates a signature based on a trace of instructions that access a cache line. By matching current signatures with previously stored signatures that lead to dead cache lines, the mechanism can predict when a given line becomes dead. The goal of this work is to allow the lines to self-invalidate when their last access is detected.

KHAN et al. (2010) proposed a Skewed Dead-Block Predictor (SDP) to predict dead lines and use these lines as a virtual victim cache. This skewed predictor is similar to the LTP mechanism, but uses two global tables. Each one of the tables is indexed by a different hash function to reduce the impact of conflicts between them. They also propose a more complex mechanism (KHAN et al., 2010), which uses three tables to implement the skewed predictor. This mechanism also introduces a sampling cache structure which uses only some of the cache lines to build the prediction table's information.

2.2.4 Time Based Dead Line Predictors

KAXIRAS; HU; MARTONOSI (2001) presented a cache decay mechanism which uses theories from competitive algorithms to create a time-based strategy. They exploit long dead periods by turning off cache lines during such periods. This approach aims to reduce static power dissipated by the cache. Each line contains a counter with a number of cycles since the last access and each line is turned off after a certain interval. A hierarchical counter mechanism is adopted to reduce the bits required per cache line. This hierarchical counter is formed by a global counter which broadcasts an increment signal to the smaller counters every time it overflows. The authors also explore an adaptive mechanism to automatically choose the best decay interval on a cache line granularity.

ABELLA et al. (2005) introduced the Inter-Access Time per Access Count (IATAC) mechanism to predict and turn off dead lines with the objective of reducing L2 cache static energy. This mechanism predicts a cache line to be dead when it detects that the line has not received any access for a period greater than the average time between different accesses. The mechanism keeps track of the average time between accesses in a global table with a separate entry for each access count (i.e., there is a different average time stored for the difference between the 1st and 2nd access and the difference between the 2nd and 3rd access and so on).

2.2.5 Last Write Predictors

LEE; TYSON; FARRENS (2000) proposed the Eager Write-back mechanism which performs early write-back of dirty lines from the L1 cache whenever the line achieves the Least Recently Used (LRU) position. The objective of this work is to speculatively "clean" the dirty cache lines prior to their eviction, in order to avoid the performance degradation during clustering bus traffic in a write-back approach.

STUECHELI et al. (2010) presented the Virtual Write Queue (VWQ) technique, which exposes to the memory controller the dirty blocks near to the LRU position in the LLC. The mechanism issues scheduled write-backs to improve the write-back efficiency when the memory rank is idle, considering that a bank can be formed by more than one rank. This technique uses a Search Set Vector (SSV) which gathers information from all the sets, in order to keep track of the dirty lines and how close they are from the LRU position. As a cache line becomes close to the LRU position, that line is marked as critical blocks to write-back.

WANG; KHAN; JIMÉNEZ (2012a) proposed a Last Write Predictor (LWP) to predict whenever the cache line receives its last write. The prediction mechanism uses three tables with a skewed organization similar to SDP mechanism to detect the last-written blocks and store pointers of these blocks into a last-write buffer. The objective of this mechanism is to make the last-write blocks available for the main memory scheduling before the line gets evicted. This predictor issues a write-back request to the memory when no read request targets the same rank.

WANG; KHAN; JIMÉNEZ (2012b) presents a two level rank idle predictor to schedule the write-back to the memory only when long phases of idle rank cycles are predicted. Together with the rank idle predictor, this paper uses the SSV in order to keep track of dirty lines and send the blocks close to the LRU position to write-back.

2.2.6 Overall Comparison

Table 2.1: State-of-the-art last access predictors.

Mechanism Description	**Saves Energy**	**Predicts Dead Lines**	**Sub-block Level**
SPP: Line usage predictor. (CHEN et al., 2004).	Yes	No	Yes
LvP: Counter based dead line predictor. (KHARBUTLI; SOLIHIN, 2008).	No	Yes	No
LTP: Trace based dead line predictor. (LAI; FIDE; FALSAFI, 2001).	No	Yes	No
SDP: Trace based dead line predictor. (KHAN et al., 2010).	No	Yes	No
IATAC: Time based dead line predictor. (ABELLA et al., 2005).	Yes	Yes	No
DSBP: Dead Sub-Block Predictor. (ALVES et al., 2012).	Yes	Yes	Yes

Table 2.1 presents a comparison between DSBP and its related work. DSBP performs better than previous work by treating the energy consumption problem at all the levels

of the cache hierarchy, and not only predict which sub-blocks should be brought into the cache, but also when active sub-blocks become dead. Additionally, DSBP reduces the number of accesses to the pattern history table by updating it only when a new pattern is detected.

Table 2.2: State-of-the-art last write predictors.

Mechanism **Description**	**Saves** **Energy**	**Predicts** **Dead Lines**	**Early** **Write-Back**
Eager Write-Back. (LEE; TYSON; FARRENS, 2000).	No	No	Yes
VWQ: Virtual Write Queue. (STUECHELI et al., 2010).	No	No	Yes
SSV: Search Set Vector. (LAI; FIDE; FALSAFI, 2001).	No	No	Yes
LWP: Last Write Predictor. (WANG; KHAN; JIMÉNEZ, 2012a).	No	No	Yes
Two Level Rank Idle Predictor. (WANG; KHAN; JIMÉNEZ, 2012b).	Yes	No	Yes
DEWP: Dead Line and Early Write-Back Predictor. (ALVES et al., 2013).	Yes	Yes	Yes

Table 2.2 shows a comparison between DEWP and its related work. Regarding the DEWP mechanism, none of the previous approaches have taken into account that dirty lines remain turned on for long periods of time, wasting energy while these lines could be evicted early. Thereby, energy can be saved and memory contention reduced. This thesis introduces a mechanism that performs the prediction of last read, last write and last access on a cache line basis, exploring the energy savings achievable by turning off invalid and dead lines, and performing early write-backs.

To evaluate DSBP and DEWP, we implemented a mechanism similar to those present in LTP (LAI; FALSAFI, 2000; LAI; FIDE; FALSAFI, 2001), SDP (KHAN et al., 2010) and LWP (WANG; KHAN; JIMÉNEZ, 2012a). These previous predictors represent the state-of-the-art and they were proposed to be used for bypassing dead lines or prioritizing them for eviction or perform early write-back after the cache line receives its last write. In our evaluations, we implemented the basis of these previous mechanisms, the *skewed predictor*, to power off the cache lines predicted to be dead and prioritizing them for eviction, also performing early write-back of dirty cache lines when predicted to be dead. This implementation is called SKEWED in the thesis.

3 SIMULATOR OF NON-UNIFORM CACHE ARCHITECTURE (SINUCA)

Some existing open source academia simulators have been verified for single threaded applications (DESIKAN; BURGER; KECKLER, 2001). However, with the latest advances in CMP, Simultaneous Multi-Threading (SMT), branch prediction, memory disambiguation prediction (DOWECK, 2006), Non-Uniform Cache Architecture (NUCA) (KIM; BURGER; KECKLER, 2003, 2002), NoC (BJERREGAARD; MAHADEVAN, 2006; BERTOZZI et al., 2005) and other mechanisms, academia does not continue the process of validating its microarchitecture simulators. For parallel architectures, mechanisms such as cache coherence, locks and synchronization between threads, interconnection networks and memory controllers affect application performance. The correct use of these shared resources can be very different depending on their implementation.

Another problem that academia researchers suffer when validating their simulators is the lack of a suite of microbenchmarks that stresses different hardware components independently to correlate their implementation with real processors. Each developer usually creates his own microbenchmarks and seldom shares them. The reason why the community accepts this is because there is no better option currently available. Moreover, the operation of specific components in the processor is not published for intellectual property reasons. For this reason, the processor must be seen as a black box, whose behavior can only be measured on a high level. As a possible way to observe the behavior at a finer granularity, microbenchmarks should be used. An emerging issue is the energy consumption modeling of the processor components, which is another feature missing from current simulators.

For these reasons, writing an accurate and validated simulator for modern microarchitectures is a difficult task, and no publicly available simulator is validated for modern parallel architectures.

With these problems in mind, we developed SiNUCA, a performance and energy validated, trace-driven, cycle accurate simulator for the x86 architecture. The simulator is capable of running single and multi-threaded applications and multiple workloads with a high level of detail of all the pipeline components. Additionally, we implemented a large set of microbenchmarks to correlate the simulator results (performance and other statistics) with two existing x86 platforms.

SiNUCA has the following main features:

High Accuracy: SiNUCA implements architectural components with a high level of detail, not only in the execution pipeline, but also in the memory and interconnection components. We also accurately model parallel architectures such as multi-core and multi-processor systems. We used publicly available information for the implementation of the simulator. Where this information was not available, we used microbenchmarks in order to observe the behavior of a real machine.

SiNUCA was validated with single-threaded and multi-threaded applications using microbenchmarks and larger workloads. The simulation statistics are compared to real machines. The microbenchmarks had an average difference of 7% while the SPEC-CPU2006 achieved a difference of 12% comparing the Instructions per Cycle (IPC) when simulating machines with the x86 Sandy Bridge architecture.

Energy Model: Energy consumption is becoming more important for current and future processor architectures. However, most current simulators do not model energy consumption, only performance. To evaluate the energy consumption, we integrated the Multi-core Power, Area, and Timing (McPAT) (LI et al., 2009, 2013) tool, which uses component statistics generated by SiNUCA. These results were validated using energy hardware counters in the real machine. Our results show a difference of 18% comparing the average energy consumption when simulating the microbenchmarks.

Support for Emerging Techniques: SiNUCA is able to model several state-of-the-art technologies, such as NUCA, Non-Uniform Memory Access (NUMA), NoC and Double Data Rate (DDR) 3 memory controllers. Besides the traditional techniques such as cache prefetchers, branch predictors and others, the support for new technologies is important for accurate simulation of new systems.

Flexibility: Another important feature to support computer architecture research is the ease of implementing or extending features. This is provided by SiNUCA with a modular architecture, written in C++, which provides a direct access to the operational details of all the components. Other simulators are limited by metalanguages that do not expose all the functionalities of the microarchitecture, making it more difficult to model new mechanisms and modify the existing ones.

The rest of this chapter is organized as follows: We begin with an overview of the state-of-the-art in computer architecture simulation. SiNUCA is presented in Section 3.2. The microbenchmarks are introduced in Section 3.3. The evaluation methodology and results are presented in Section 3.4. We summarize SiNUCA in Section 3.5.

3.1 Related Work

In this section, we analyze related computer architecture simulators and compare them to our proposed simulator.

The work of (DESIKAN; BURGER; KECKLER, 2001) validates *sim-alpha*, a simulator based on the *sim-out-order* version of SimpleScalar (AUSTIN; LARSON; ERNST, 2002). In this work, the authors aim to simulate the Alpha 21264 processor, using all available documentation. They use microbenchmarks in order to scrutinize every aspect of the architecture, being able to achieve an average error of 2% for a set of microbenchmarks, and an error of 18% for the SPEC-CPU2000 applications. The authors identify the memory model as the main source of errors. They show that often used simulators, such as SimpleScalar, might contain modeling errors and become less reliable to test certain new features.

We use a similar validation process for SiNUCA, making separate evaluations for specific components inside the processor by using microbenchmarks. We extend the control and memory microbenchmarks and include parallel applications.

Virtutech SimICS (MAGNUSSON et al., 2002) is a full-system, functional simulator which measures execution time based on the number of instructions executed multiplied by a fixed IPC, and the number of stall cycles caused by the latency of all components.

WEAVER; MCKEE (2008) compared the SESC simulator (RENAU et al., 2005) to the Dynamic Binary Instrumentation (DBI) using QEMU (BELLARD, 2005). The authors show that in general, cycle-accurate simulators generate inaccurate results over a long execution time, due to lack of correctness in architectural details. They are able to obtain similar results in an order of magnitude shorter time with DBI. The paper also lists the flaws in cycle-accurate simulators. They cite speed, obscurity, source code forks, generalization, validation, documentation and lack of operating system influence as the major factors when considering the use of a simulator. Regarding these issues, SiNUCA solves several issues with code modularity, use of traced instructions and validation.

Gem5 (BINKERT et al., 2011) is a combination of two simulators: M5 (BINKERT et al., 2006) and General Execution-driven Multiprocessor Simulator (GEMS) (MARTY et al., 2005). Within Gem5, M5 simulates the cores, whereas GEMS simulates the memory hierarchy. The validation of the Gem5 simulator, modeling a simple embedded system (BUTKO et al., 2012), shows that errors can vary from 0.5% to 16% for the applications from the SPLASH-2 and ALPBench suites. However, for small synthetic benchmarks with tiny input sizes, the error varies from 3.7% to 35.9%. The authors conclude that the Dynamic Random Access Memory (DRAM) model is inaccurate.

PTLsim (YOURST, 2007) is a cycle accurate, full-system x86-64 microprocessor simulator and virtual machine. It is integrated with the Xen hypervisor in order to provide full-system capabilities.

Multi2Sim (UBAL et al., 2007) is an event-driven simulator based on the premise of simulating multi-threaded systems. It was extended to simulate heterogeneous CPU-GPU systems in (UBAL et al., 2012).

COTSon is a simulator framework jointly developed by HP Labs and AMD (ARGOLLO et al., 2009) to provide fast and accurate evaluation of current and future computing systems, covering the full software stack and complete hardware models. As SimICS, COTSon also abstracts processor microarchitecture details, prohibiting development of novelty at this level.

MARSSx86 (PATEL et al., 2011) is a cycle-accurate simulator based on PTLSim. Although it simulates all architectural details, it does not ensure accuracy, as can be seen in their comparison to the SPEC-CPU2006 workload, getting errors of up to 60% with an average error of 21%.

Table 3.1 summarizes the main characteristics for computer architecture simulators. Full-system simulation enables processor designers to evaluate OS improvements or its impact on the final performance. However, the OS simulation can introduce noise during the evaluations, requiring several simulation repetitions (higher simulation time) in order to obtain reliable results with a reduced Operating System (OS) influence.

Microarchitectural and multi-core simulation are required to evaluate most of the state-of-the-art component proposals.

We consider SimICS, GEMS, Gem5 and COTSon as not easy to extend because these simulators have private source code, need metalanguages to modify the architecture or require modifications of multiple simulation levels in order to perform microarchitectural changes.

Table 3.1: Comparison of state-of-the-art simulators.

Simulator Name	**Full-System Simulation**	**Micro-Architecture Simulation**	**Multi-Core Simulation**	**Extension Flexibility**	**NoC Modeling**	**NUCA Support**	**Memory Controller Modeling**
SimAlpha	No	Yes	No	High	No	No	No
SimICS	Yes	No	Yes	Low	No	No	No
SESC	No	Yes	Yes	High	Detailed	Yes	Extension
GEMS	No	Yes	Yes	Low	Simple	Yes	No
M5	Yes	Yes	Yes	High	Simple	Yes	No
Gem5	Yes	Yes	Yes	Low	Simple	Yes	No
PTLsim	Yes	Yes	No	High	No	No	Extension
Multi2Sim	No	Yes	Yes	High	Detailed	Yes	No
COTSon	Yes	No	Yes	Low	Detailed	Yes	No
MARSSx86	Yes	Yes	Yes	High	Detailed	No	No
SiNUCA	No	Yes	Yes	High	Detailed	Yes	Detailed

Regarding NoC modeling, different detail levels can be observed among the simulators. SimAlpha, SimICS and PTLsim do not natively support it. GEMS, M5 and Gem5 model only the interconnection latency without modeling traffic contention.

Considering NUCA, we classified the simulators as having support if they model at least multi-banked caches, also known as static NUCA (KIM; BURGER; KECKLER, 2003).

The memory controller is becoming more important in modern processors, because of its integration inside the processor chip. If a simulator only simulates a fixed latency for every DRAM request, it is classified as not capable of modeling a memory controller. Although SESC and PTLsim do not support memory controller modeling natively, extensions were proposed in order to overcome this deficiency.

We can observe that SiNUCA offers all the features required to evaluate cache memory mechanisms, simulating in detail the cache memories and memory controllers, while offering a highly detailed model of the OoO processor microarchitecture.

3.2 SiNUCA

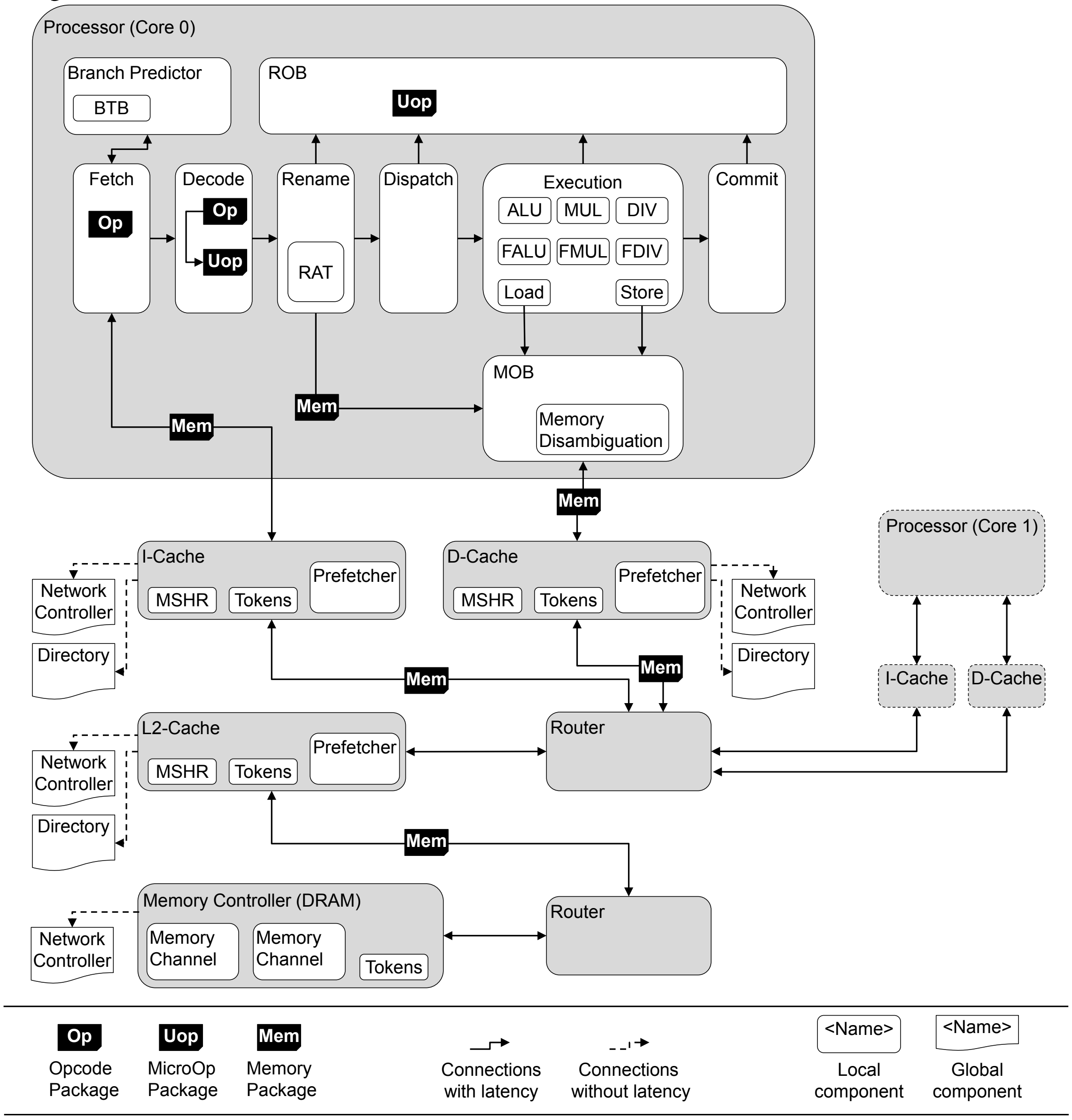
We developed SiNUCA, a trace-driven simulator, which executes traces generated on a real machine with a real workload without the influence from the OS or other processes. The traces are simulated in a cycle-accurate way, where each component is modeled to execute its operations on a clock cycle basis. SiNUCA currently focuses on the x86_32 and x86_64 architectures.

SiNUCA was developed in C++ to make use of object-oriented principles and generate modular components. Such characteristics are preferable for simulators in order to ease the implementation of new features or components.

The ideal simulator should be as flexible as possible. Considering this, multiple parameters are available to be set through the SiNUCA configuration file. The configuration file is split into modules, such that each module contains the internal parameters to be used by the simulator. The simulator uses a component called SiNUCA configurator. It uses the *libconfig* (LINDNER, 2013) library internally to read the configuration file and to instantiate all the components inside SiNUCA. Libconfig is a fully reentrant parser and it supports include directives. In this way, SiNUCA components can be defined separately and included multiple times inside the main configuration file, facilitating the description of multi-core and many-core systems.

3.2.1 System Model

Figure 3.1: SiNUCA architecture with its main components and interconnections, modeling an Intel Core 2 Duo architecture.



The main components of SiNUCA are illustrated in Figure 3.1. The description of the components is presented below:

Memory Package: Every memory operation inside the simulator is encapsulated within this component.

Opcode Package: The instructions inside the simulator trace and front-end are encapsulated within this component.

MicroOp Package: After decoding the Opcode Package, the micro-operations are encapsulated within this component.

Token: Every communication between two memory components, such as cache memories and memory controllers, needs to request a token from the final package destination before the communication starts. Tokens avoid deadlocks during package transfers that require more than one hop to reach their final destination.

Processor: This component is responsible for executing the Opcodes. It consists of the fetch, decode, rename, dispatch, execute and commit stages. Currently, only an OoO processor is modeled, implementing a Reorder Buffer (ROB) to handle the OoO execution. Further details about the processor's internal components will be presented in Section 3.2.2.

Cache Memory: This component is responsible for modeling instruction and data caches, both single and multi-banked models (static NUCA), implementing a Miss-Status Handling Registers (MSHR) internally per bank. This component keeps only the tag array of the cache, reducing the memory usage and the trace size.

Miss-Status Handling Register (MSHR): This buffer keeps information about the cache misses which arrive at each cache bank. The requests wait to be serviced and are then sent back to the requester in the case of a cache hit, or replicated to the next memory level in the case of a cache miss.

Prefetcher: The memory packages serviced by the cache memory are sent to the prefetcher, so that the memory addresses required by the application in the future can be predicted and fetched. This component is responsible for implementing several prefetch policies and to provide the cache memory with the generated requests as soon the cache has space in its MSHR. Currently, a *stride prefetcher* (BAER; CHEN, 1991) and a *stream prefetcher* (JOUPPI, 1990) are available to be simulated.

Memory Controller: This component implements the memory controller, formed by multiple channels, each channel with multiple banks. All memory requests that miss in all the cache levels will be sent to this component, which will schedule the requests to the memory banks. The memory controllers can support NUMA modeling as well.

Memory Channel: Inside the memory controller, multiple memory channels can be instantiated. Each channel may be connected to one or more banks, each bank with its own read and write buffers.

Router: This component was developed to model the contention and delays imposed by the interconnections. It implements a general NoC router that automatically delivers packages using the routing information inside each package.

Network Controller: This controller is used to generate a communication graph with the routes available in the modeled architecture. All packages that need to be transmitted have a pointer to the routing information contained in the routing table. The routing information describes which path to take for each intermediary component (routing input/output ports).

Directory: This component models the MOESI cache coherence protocol, which is accessed directly by the caches for every memory operation. It is responsible for creating locks on cache lines in order to control concurrent operations, to change the cache line status (for example, after writes or when propagating invalidations), to generate write-backs and to inform the caches about hits or misses. During read requests, the directory creates a read lock to that memory address, thus only read accesses can be provided to that address in any cache in the system. During write operations, the directory locks the cache line for the write operation. In this way, no other operation can be performed by other caches on the same address until the lock is released.

The network controller and the directory are virtual components, which model lookup tables and do not generate latency on accesses to them. All other components are real components and the connection between them always generates latency.

3.2.2 Processor Description

Figure 3.1 presents the details of the processor architecture. The processor consists of 6 main stages, each stage can be configured to take multiple cycles to complete. Although the processor could be simulated with less stages, we chose to implement these main stages separately in order to increase the flexibility and ease simulator extensions. The implementation details for the main components and stages inside the processor are given below:

Fetch Stage: This stage is responsible for fetching new opcodes from the trace reader. Internally, it handles the next two opcodes from the trace in order to send them to the branch predictor. In case of a branch instruction, the branch predictor can inform if it was correctly predicted or not, and update the prediction mechanism. In case of a non-branch instruction or a correctly predicted branch, it sends the next instruction to the fetch buffer, so the instruction address can be requested from the instruction cache.

In the case of a mispredicted branch, the fetch stage will be stalled until the branch is solved. The fetch stage is also responsible for controlling the synchronization of multiple threads according to the Open Multi-Processing (OpenMP) primitives inside the trace. For instance, during an OpenMP barrier, the fetch stage of each processor will control when all processors reach the same barrier, so the simulation can continue. During this stall period, no new instruction is fetched from the trace for the processors waiting in the barrier.

Decode Stage: This stage takes the opcodes ready in the fetch buffer, and decodes them into multiple microOps. The microOps are inserted into the decode buffer. For every branch decoded, an in-flight branch counter is incremented, and then decremented after the branch is retired. This stage stalls when the maximum number of branches inside the pipeline is reached.

Rename Stage: This stage obtains available microOps from the decode buffer and inserts them into the ROB. If the microOp contains a register operation, the Registers Alias Table (RAT) solves false dependencies, and stores information about real dependencies. If the microOp contains a memory operation, the related memory operation will be inserted into the Memory Order Buffer (MOB) read or write buffer. In case of a full ROB or MOB, this stage will stall.

Dispatch Stage: MicroOps are sent to the reservation stations, even if they still have missing dependencies. This stage stalls if all the reservation stations are full.

Execution Stage: Once the microOp is ready for execution, with all the dependencies solved, the microOp is executed. In this stage, the constraints on the number of functional units (such as integer and floating-point ALU) and cycles between microOps are respected. In the end of this stage, data forwarding is performed, and the real dependent microOps are marked as resolved. For memory operations, the memory package inside the MOB relative to this operation will be marked as ready.

Commit Stage: This stage is responsible for retiring the opcodes in-order from the ROB.

Branch Predictor: The predictor receives the current and next instruction in the trace. It is responsible for implementing any branch predictor mechanism with its Branch Target

Buffer (BTB) and other structures. It returns to the processor if the branch was predicted correctly or not, together with the information about the stage in which the branch will be solved. Thus, the fetch stage will stall until the mispredicted branch reaches the informed stage. During each prediction, the prediction correctness information is also updated inside the predictor structures. Currently, the Per-address Adaptive branch prediction using per-Set PHT (PAs), Global Adaptive branch prediction using per-Set PHT (GAs), Per-address Adaptive branch prediction using one Global PHT (PAg) and Global Adaptive branch prediction using one Global PHT (GAg) (YEH; PATT, 1991, 1992) branch predictor mechanisms are available inside the simulator.

Reorder Buffer (ROB): This circular buffer stores the microOps after the decode stage. It also handles information about the dependencies and the current stage of the microOps.

Register Alias Table (RAT): This table stores the information of the last microOps to write every register. Every time a register is going to be re-written by a microOp or the microOp is executed, this table is updated. Every time a register is being read, this table is consulted in order to gather information about the true dependencies between microOps.

Memory Order Buffer (MOB): Whenever a memory operation is executed, that is, its address is generated, the memory package inside the load or store buffer located inside the MOB is marked as ready. The package waits in one of these buffers until the memory services the request. After the memory operation is serviced, the data dependencies are solved and the memory operation can be committed.

Memory Disambiguation: This component contains the table that stores the last memory write addresses and their sizes. This table works in a similar manner as the RAT, but in this case, it can solve false dependencies between memory operations depending on the disambiguation policy.

3.2.3 Simulator Traces

The SiNUCA input traces are split into three different files:

Static trace: This file contains the trace of the assembly code divided in basic blocks. This file is completely loaded into the main memory at the beginning of the simulation.

Dynamic trace: This file contains the calls to the basic blocks generated during the program execution. For each basic block call, the instructions are obtained from the static trace.

Memory trace: This file contains information about the memory operations generated during the program execution. For each memory operation, a new trace line is read so the memory address and memory size can be retrieved.

3.2.3.1 Trace Format

In order to show a simple example from the SiNUCA traces, Table 3.2 presents a short synthetic code, consisting of a loop that executes two iterations, adding the loop index to a variable.

Table 3.3 shows the static, dynamic and memory traces for SiNUCA, generated from the synthetic code presented in Table 3.2.

The static trace is divided into basic blocks, which are presented after @. The instructions inside the basic block consist of the following fields: *assembler instruction*, *Opcode*

Table 3.2: Source code example written in C and Assembly.

C source code	Assembly code (AT&T syntax)		
		movl	`$0x0,-0x8(%rbp)`
for (i = 0; i < 2; i++) {	**loop:**	**mov**	`-0x8(%rbp),%eax`
sum += i;		**add**	`%eax,-0x4(%rbp)`
}		**addl**	`$0x1,-0x8(%rbp)`
		cmpl	`$0x1,-0x8(%rbp)`
		jbe	**loop**

Table 3.3: SiNUCA traces for a simple source code.

Static Trace	Memory Trace	Dynamic Trace
#**main**	R 4 0x1701448 1	1
@1	#	2
MOV 1 0x95727 4 1 14 1 34 14 0 1 0 0 0 0 0	R 4 0x1701448 2	2
#**main**	R 4 0x1701452 2	
@2	W 4 0x1701452 2	
MOV 8 0x95717 3 1 14 1 65 14 0 1 0 0 0 0 0	R 4 0x1701448 2	
ADD 1 0x95720 3 2 14 65 1 34 14 0 1 0 1 0 0 0	W 4 0x1701448 2	
ADD 1 0x95723 4 1 14 1 34 14 0 1 0 1 0 0 0	R 4 0x1701448 2	
CMP 1 0x95727 4 1 14 1 34 14 0 1 0 0 0 0 0	#	
JBE 7 0x95731 2 2 35 34 1 35 0 0 0 0 0 1 0 0	R 4 0x1701448 2	
	R 4 0x1701452 2	
	W 4 0x1701452 2	
	R 4 0x1701448 2	
	W 4 0x1701448 2	
	R 4 0x1701448 2	

type, *instruction address*, *instruction size*, *number of read registers*, *read registers*, *number of write registers*, *write registers*, *base register*, *index register*, *is-read* flag, *is-read2* flag, *is-write* flag, *is-branch* flag, *is-predicated* flag, *is-prefetch* flag.

The *assembler instruction* is only saved in order to ease the trace debug, but it is not used inside the simulator. The registers inside SiNUCA are represented by the register number given by Pin, instead of the register name. This transformation facilitates the dependency analysis.

The *base register* and *index register* fields are needed to differentiate the registers used to access the memory from the other registers used inside the instruction. This is necessary to keep the correct dependencies inside each microOp.

The flags *is-read*, *is-read2*, *is-write* and *is-branch* are used in order to decode the opcode into multiple microOps. The *is-predicated* and *is-prefetch* flags are not currently used, but they are kept so the simulator can be easily extended.

The memory trace is formed by the following fields: *R/W indicator*, *memory operation size*, *memory address*, *basic block number*. The *R/W indicator* and the *basic block number* are only used to guarantee that the memory trace is correct, that is, these fields are matched with the static instruction which fetched the memory operation.

The dynamic trace only contains the *basic block number*. In order to read the trace, the processor reads this file fetching the basic block number, then it fetches the instructions inside the static trace for that specific basic block. The basic blocks present in the dynamic trace correspond only to the actual execution flow, that is, no wrong path (such as a mispredicted branch) is traced. If the instruction in the basic block performs a memory operation, the memory trace is fetched in order to get the memory address and the size of the memory access.

3.2.3.2 Trace Generation

Considering that some applications can take a very long time to run on a real machine, it is impractical to execute them completely in a cycle accurate simulator such as SiNUCA. Our trace generator uses the PinPoints (PATIL et al., 2004) tool to find and trace only the representative portions of the applications.

PinPoints uses the SimPoint methodology (SHERWOOD et al., 2002) to profile and identify representative portions (slices) of an application, in order to trace only the most significant slice of single-threaded applications.

SimPoints uses basic block distribution analysis (SHERWOOD; PERELMAN; CALDER, 2001) in order to determine cyclic behavior of an application. This analysis correlates clusters of basic blocks called Basic Block Vectors (BBVs) to the entire execution, in order to identify the initialization phase and phases with periodic behavior inside the program. The final goal of this analysis is to find the preferred simulation points in the application in order to achieve a representative sample of its execution. According to the authors, using the basic block distribution analysis, it is possible to find small representative slices of the program, which result in an IPC error of 6% or less.

Considering that PinPoints does not work well for parallel applications, we used it only to generate traces from the SPEC-CPU2006 applications. For the parallel applications, we instrumented the trace generator to trace only given parallel regions. For the microbenchmarks, the full execution traces were used.

For multi-threaded applications, the waiting time in a OpenMP spin-lock during the trace generation can represent a large portion of the execution time. This occurs because threads need to wait for others to write their traces. This spin-lock waiting time can be higher if the number of threads being traced is bigger than the number of cores of the machine that is generating the traces.

In order to avoid the spin-lock stalls, the basic blocks that contain OpenMP locks are removed from the traces. However, in order to correctly simulate the barriers and the waiting time lost during synchronization, the traces still contain the OpenMP primitives that are used to synchronize the threads internally during the simulation.

During the simulation of multi-threaded applications, the simulated processors treat the OpenMP primitives as they appear in the dynamic trace. This means that during program barriers, each processor that reaches the barrier stops fetching instructions and waits until all the processors reach the same barrier. When all processors reach the barrier, the lock is released and the execution continues. The OpenMP atomic and critical primitives are preserved inside the dynamic trace, and the processors also guarantee that only one processor will enter a critical section at a time.

3.2.4 Energy Modeling

In order to obtain estimations regarding the energy and power consumption, we instrumented the simulator to generate all the statistics required by McPAT, version 1.0 (LI et al., 2009, 2013).

The McPAT framework is a validated and widely used tool which supports processor configurations ranging from 90 nm to 22 nm and beyond, modeling timing, area and power for the device types forecast in the ITRS roadmap (Semiconductor Industry Association, 2007). It supports splitting energy results into dynamic and static energy, with the static energy formed by subthreshold and gate leakage.

The power, area and timing models are organized into three levels: The *Architectural level*, where a multi-core model is decomposed into major components such as cores, interconnections, caches and memory controllers; The *Circuit level*, which maps the architectural blocks to hierarchical wires, arrays, logic and clocking networks; The *Technology level*, which uses data from the ITRS roadmap to calculate the physical parameters of devices and wires.

3.3 Microbenchmarks

As mentioned before, many architectural details of modern processors are not public. For this reason, a set of microbenchmarks was developed in order to evaluate and estimate the behavior of processor components. We developed a suite of single and multi-threaded microbenchmarks to isolate and validate specific aspects of SiNUCA. These benchmarks will be presented in this section.

3.3.1 Single-Threaded Microbenchmarks

The single-threaded microbenchmarks were inspired by the SimAlpha validation (DESIKAN; BURGER; KECKLER, 2001). Five categories of benchmarks were defined. These are *Control*, *Dependency*, *Execution*, *Memory* and *Multi-thread.* These categories stress different stages of the pipeline and evaluate different components separately.

3.3.1.1 Control Benchmarks

Five *Control* benchmarks were designed to stress different situations commonly found in programs: *Complex*, *Conditional*, *Random*, *Small BBL*, and *Switch*.

Control Complex mixes *if-else* and *switch* constructs in order to create a hard to predict branch behavior.

Control Conditional implements a simple if-then-else construct in a loop that is repeatedly executed, and alternates between taking and not taking the conditional branch.

Control Random implements an if-then-else construct in a loop that is repeatedly executed. For every iteration, a Linear Feedback Shift Register (LFSR) function decides between taking and not taking the conditional branch. The LFSR function is a well known fast pseudo-random number generator, which was implemented with 16 bits in order to get a period length of 65535 iterations.

Control Small BBL evaluates the number of in-flight branches at the same time inside the processor by executing a loop with only the control variable incrementing inside the loop.

Control Switch tests indirect jumps with a *switch* construct formed by 10 case statements within a loop. Each case statement is taken $n/10$ times on consecutive loop iterations before moving to the next case statement, where n is the total number of repetitions of the loop.

3.3.1.2 Dependency Benchmarks

Six *Dependency Chain* benchmarks were used to stress the forwarding of dependencies between instructions. These evaluate dependency chains every 1, 2, 3, 4, 5 and 6 instructions. In this way, every instruction needs to wait for the execution of the previous one with a real dependency, evaluating thus the data forwarding time.

3.3.1.3 Execution Benchmarks

Six *Execution* benchmarks were defined to stress the integer and floating point functional units: *integer-add*, *integer-multiply*, *integer-division*, *floating-point-add*, *floating-point-multiply*, *floating-point-division*.

All the *Execution* benchmarks execute 32 independent operations inside a loop, with a reduced amount of memory operations, control hazards and data dependences, allowing a close-to-ideal throughput.

3.3.1.4 Memory Benchmarks

Three *Memory* benchmarks were created: *load-dependent*, *load-independent*, *store-independent*.

Load Dependent executes a loop which walks a linked list, waiting for each load to complete before starting the next. Different linked list sizes (16 KB, 32 KB, 64 KB, 128 KB, 256 KB, 512 KB, 1024 KB, 2048 KB, 4096 KB, 8192 KB, 16384 KB and 32768 KB) are used in the evaluation.

Load Independent repeatedly executes 32 parallel independent loads from an array and adds its values in a scalar variable. Different array sizes were used in the evaluation. Twelve array sizes were evaluated (16 KB, 32 KB, 64 KB, 128 KB, 256 KB, 512 KB, 1024 KB, 2048 KB, 4096 KB, 8192 KB, 16384 KB and 32768 KB) in order to stress different cache levels.

Store Independent repeatedly performs 32 parallel independent stores of a fixed scalar iterating over all the positions of an array, evaluating multiple array sizes as well. The same twelve array sizes as for the load independent were evaluated.

3.3.2 Multi-Threaded Microbenchmarks

Four *multi-threaded* microbenchmarks were designed to reproduce different scenarios from real multi-threaded applications. The applications consist basically of two loops, i and j, where the loop j executes a single iteration per thread. Every thread accumulates the multiplication of i per j.

Barrier All Together: all threads have the same amount of work, waiting for a barrier after every new iteration in the loop i.

Barrier Master Slow: in this case, only the thread master iterates over an inner loop, forcing all other threads to wait at the barrier to synchronize.

Critical: for this application, the accumulator used by all the threads is a shared variable, thus, all the increments requires the thread to enter in a critical section, creating great contention.

False Sharing: to stress the coherence protocol, an array is shared by different threads to store the sum being calculated. However, since the multiple array elements (8 bytes each) will share the same cache line, a false sharing effect occurs. In order to evaluate the influence of different amounts of threads suffering from false sharing, different pads of 0, 8, 24 and 56 bytes were evaluated. These pad sizes were chosen to keep the same amount of threads falsely sharing the cache lines. Notice that this would not be possible with a pad of 16 bytes, as there would be one variable every 24 bytes, leading to some cache lines having 3 elements and others having 4 elements.

3.4 Evaluation

To evaluate and validate SiNUCA, we performed extensive experiments with a large set of sequential and parallel benchmarks. This section presents the methodology of the experiments, validation results, a discussion of the differences between the simulator and the real machine, and the simulator performance.

3.4.1 Methodology

3.4.1.1 Workloads

For the microbenchmarks, we equalized their execution time to 0.5 seconds on the real machine. This ensures that the amount of work performed is significant while keeping the simulation time reasonable for the full execution.

Besides the microbenchmarks, we used 43 benchmarks from 3 different benchmark suites: all 29 applications from the SPEC-CPU2006 (HENNING, 2006) suite, 7 OpenMP multi-threaded applications from SPEC-OMP2001 (SAITO et al., 2006), and 7 OpenMP multi-threaded applications from the Numerical Aerodynamic Simulation (NAS) Parallel Benchmarks (NPB) version 3.3.1 (BAILEY et al., 1991) suite. All benchmarks were compiled for x86-64 using the GNU Compiler Collection (GCC) version 4.6.3 with the `-O3` performance optimization flag.

The SPEC-CPU2006 benchmark suites (integer and floating point) were executed using the *reference* input set, executing a representative slice of 200 million instructions selected by PinPoints.

The SPEC-OMP2001 benchmarks were executed using the *reference* input set as well, while the NAS-NBP suite was run using input set size *A*. Each benchmark executes up to one time step from its parallel region.

Table 3.4: Average number of instructions (millions) per thread.

	Name	1 Thread	2 Threads	4 Threads	8 Threads
SPEC-OMP2001	applu.M	809 M	404 M	200 M	97 M
	apsi.M	1437 M	717 M	359 M	177 M
	fma3d.M	329 M	165 M	82 M	41 M
	galgel.M	417 M	209 M	105 M	53 M
	mgrid.M	1136 M	568 M	284 M	142 M
	swim.M	1383 M	690 M	345 M	170 M
	wupwise.M	1629 M	815 M	407 M	202 M
	Average	1020 M	510 M	255 M	126 M
NAS-NPB	BT.A	1851 M	926 M	463 M	231 M
	CG.A	355 M	178 M	89 M	44 M
	FT.A	3077 M	1539 M	769 M	385 M
	IS.A	247 M	123 M	62 M	31 M
	LU.A	805 M	419 M	221 M	124 M
	MG.A	2081 M	1040 M	520 M	260 M
	SP.A	467 M	234 M	117 M	58 M
	Average	1269 M	637 M	320 M	162 M

Table 3.4 shows the average number of instructions simulated per thread, for the SPEC-OMP2001 and NAS-NBP suites. Notice that we are always tracing only one time step of each application, such that the total number of executed instructions across the different numbers of threads are similar.

Table 3.5: Memory footprint of the benchmarks.

SPEC-CPU2006 INT		**SPEC-CPU2006 FP**		**SPEC-OMP2001**		**NAS-NPB**	
Name	Footprint	Name	Footprint	Name	Footprint	Name	Footprint
astar	36.80 MB	bwaves	96.82 MB	applu	1387.36 MB	BT	42.74 MB
bzip2	4.25 MB	cactusADM	55.50 MB	apsi	206.14 MB	CG	21.80 MB
gcc	21.13 MB	calculix	16.06 MB	fma3d	614.94 MB	FT	321.52 MB
gobmk	3.10 MB	dealII	2.48 MB	galgel	4.70 MB	IS	66.01 MB
h264ref	2.04 MB	gamess	0.33 MB	mgrid	429.68 MB	LU	40.59 MB
hmmer	3.98 MB	GemsFDTD	149.71 MB	swim	1433.71 MB	MG	429.58 MB
libquantum	32.00 MB	gromacs	3.49 MB	wupwise	768.00 MB	SP	45.19 MB
mcf	382.64 MB	lbm	198.37 MB				
omnetpp	26.78 MB	leslie3d	74.41 MB				
perlbench	1.99 MB	milc	150.65 MB				
sjeng	4.29 MB	namd	1.95 MB				
xalancbmk	30.28 MB	povray	0.17 MB				
		soplex	213.28 MB				
		sphinx3	5.88 MB				
		tonto	0.88 MB				
		wrf	47.09 MB				
		zeusmp	47.54 MB				
Average	45.77 MB	**Average**	62.62 MB	**Average**	692.08 MB	**Average**	138.20 MB

The applications from SPEC-CPU2006, SPEC-OMP2001 and NAS-NPB are presented in Table 3.5. The memory footprint of each application is shown along its name. For the parallel applications, the memory footprint is split among the threads, where each thread footprint is almost the same.

Two applications from the multi-threaded benchmarks, *art* from SPEC-OMP2001 and *EP.A* from NAS-NPB, are formed by a small initialization parallel region and between 3 and 5 huge parallel regions. Considering that simulating less than one algorithm step would be meaningless, these applications were removed from our evaluations.

Trace-driven simulators usually have a limitation simulating applications with a producer-consumer behavior. This happens because the trace often does not have information about the dependency between the producer and consumer. Due to this reason, the applications that contain OpenMP locks (*ammp*, *equake*, *gafort* from SPEC-OMP2001 and *UA.A* from NAS-NPB) were removed from our evaluation.

3.4.1.2 Real Machine and Simulation Parameters

Table 3.6 presents the parameters used in order to model the Intel Core 2 Duo (Conroe microarchitecture - model E6300)(BOJAN et al., 2007)) and Intel Xeon (Sandy Bridge microarchitecture - model Xeon E5-2650)(YUFFE et al., 2011) architectures inside SiNUCA. The table shows parameters of the execution cores, branch predictor, cache memories and prefetchers, as well as the memory controller.

3.4.1.3 Evaluation Methodology

In order to reduce the influence from the operating system, the experiments in the real machine were repeated 100 times. For each execution, we set a static affinity between

Table 3.6: Parameters to model the Core 2 Duo and Sandy Bridge processors.

Parameter	Core 2 Duo Configuration	Sandy Bridge Configuration
OoO Execution Cores	2 cores @ 1.8 GHz, 65 nm; in-order front-end and commit; 12 stages (2-fetch, 2-decode, 2-rename, 2-dispatch, 2-commit stages); 16 B fetch block size, fetch up to 6 instructions Decode and commit up to 4 instructions; Rename/dispatch/execute up to 4 micro instructions; 18-entry fetch buffer, 24-entry decode buffer; 3-alu, 1-mul. and 1-div. int. units (1-4-26 cycle); 1-alu, 1-mul. and 1-div. fp. units (3-5-5 cycle); 1-load and 1-store functional units (1-1 cycle); MOB entries: 24-read and 16-write; 96-entry ROB;	8 cores @ 2.0 GHz, 32 nm; in-order front-end and commit; 16 stages (3-fetch, 3-decode, 3-rename, 2-dispatch, 3-commit stages); 16 B fetch block size, fetch up to 6 instructions Decode and commit up to 5 instructions; Rename/dispatch/execute up to 5 micro instructions; 18-entry fetch buffer, 28-entry decode buffer; 3-alu, 1-mul. and 1-div. int. units (1-3-32 cycle); 1-alu, 1-mul. and 1-div. fp. units (3-5-10 cycle); 1-load and 1-store functional units (1-1 cycle); MOB entries: 64-read and 36-write; 168-entry ROB;
Branch Predictor	1 branch per fetch; 8 parallel in-flight branches; 4 K-entry 4-way set-associative, LRU policy BTB; Two-Level PAs 2-bits predictor; 16 K-entry PBHT; 128 lines and 1024 sets SPHT;	1 branch per fetch; 8 parallel in-flight branches; 4 K-entry 4-way set-associative, LRU policy BTB; Two-Level GAs 2-bits predictor; 16 K-entry PBHT; 256 lines and 2048 sets SPHT;
L1 Data Cache	32 KB, 8-way, 64 B line size; LRU policy; 1-cycle; MSHR: 8-request, 10-write-back, 1-prefetch; Stride prefetch: 1-degree, 16-strides table;	32 KB, 8-way, 64 B line size; LRU policy; 2-cycle; MSHR: 8-request, 10-write-back, 1-prefetch; Stride prefetch: 1-degree, 16-strides table;
L1 Inst. Cache	32 KB, 8-way, 64 B line size; LRU policy; 1-cycle; MSHR: 8-request, 1-prefetch; Stride prefetch: 1-degree, 16-strides table;	32 KB, 8-way, 64 B line size; LRU policy; 2-cycle; MSHR: 8-request, 1-prefetch; Stride prefetch: 1-degree, 16-strides table;
L2 Cache	Shared 2 MB, 8-way, 64 B line size; LRU policy; Inclusive Cache; MOESI coherence protocol; 4-cycle; MSHR: 2-request, 6-write-back, 2-prefetch; Stream prefetch: 2-degree, 16-distance, 64-streams;	Private 256 KB, 8-way, 64 B line size; LRU policy; MSHR: 4-request, 6-write-back, 2-prefetch; 4-cycle; Stream prefetch: 2-degree, 32-distance, 256-streams;
L3 Cache	None	Shared 16 MB (8-banks), 2 MB per bank; 16-way, 64 B line size; LRU policy; 6-cycle; Inclusive LLC; MOESI coherence protocol; MSHR: 8-request, 12-write-back; Bi-directional ring interconnection;
DRAM Controller and Bus	Off-chip DRAM controller, 8 DRAM banks/channel; 2-channels, 4 burst length; 8 KB row buffer per bank, Open-row first; DDR2 667 MHz; 2.8 core-to-bus frequency ratio; 4-CAS, 4-RP, 4-RCD and 30-RAS DRAM cycles;	On-chip DRAM controller, 8 DRAM banks/channel; 4-channels, 8 burst length; 8 KB row buffer per bank, Open-row first; DDR3 1333 MHz; 1.5 core-to-bus frequency ratio; 9-CAS, 9-RP, 9-RCD and 28-RAS DRAM cycles;

the threads and the processor cores, in order to reduce the overhead of migration. The OS priority for the benchmark applications was set to 10 using $nice - 10$, reducing the influence of the background tasks.

To obtain the performance counters (cycles, instructions, branch mispredictions and cache misses) we used the *perf* tool (Linux Kernel Developers, 2013). To measure the energy consumption, the Intel Performance Counter Monitor (PCM) tool (INTEL, 2012) was used to measure the Running Average Power Limit (RAPL) counters (INTEL, 2013). Core 2 Duo architecture does not provide energy related hardware counter, thus, only performance wide evaluations are performed for this architecture.

Our evaluation shows information regarding the difference between the real machine and the simulator. This difference represents the absolute accuracy of SiNUCA, it determines if the individual components and total performance and power are correctly modeled. However, the standard deviations of the differences are also presented in order to evaluate the relative accuracy of SiNUCA. A low standard deviation indicates that simulation and real machine are behaving similarly.

In order to summarize the results, the arithmetic mean was used for raw numbers while the geometric mean was used for normalized results. Geometric mean is considered the only correct mean when averaging normalized results (FLEMING; WALLACE, 1986), that is results that are presented as ratios to reference values, this is because using the arithmetic or harmonic mean could change the conclusions depending on what number is used as a reference for the normalization (JAIN, 1991).

3.4.2 Results

3.4.2.1 Single-Threaded Microbenchmarks

Table 3.7: Single-threaded microbenchmarks results for Core 2 Duo.

		Characteristic										IPC			Branch Miss		L1 DATA MPKI	
		Branch Predictor	Branch Miss Penalty	In-flight Branches	Register File	Functional Units	L1 Cache	L2 Cache	LLC	DRAM	Prefetcher	Real	Sim	Diff	Real	Sim	Real	Sim
Control	Complex	•										1.42	2.58	82%	2.89%	0.00%	0.01	0.00
	Conditional	•										1.67	2.73	63%	0.00%	0.00%	0.00	0.00
	Random	•	•									0.93	0.95	2%	28.56%	28.57%	0.01	0.00
	Small_bbl	•		•								3.01	3.00	0%	0.00%	0.00%	0.01	0.00
	Switch	•										1.43	1.83	28%	0.00%	0.00%	0.01	0.00
Dependency	Chain-1				•							1.07	1.09	2%	0.01%	0.00%	0.01	0.00
	Chain-2				•							1.55	2.19	41%	0.01%	0.00%	0.01	0.00
	Chain-3				•							2.17	3.00	38%	0.01%	0.00%	0.01	0.00
	Chain-4				•							2.74	3.00	9%	0.01%	0.00%	0.00	0.00
	Chain-5				•							2.87	3.00	4%	0.00%	0.00%	0.01	0.00
	Chain-6				•							2.93	3.00	2%	0.01%	0.00%	0.00	0.00
Execution	FP-add					•						1.10	1.09	0%	0.01%	0.01%	0.02	0.00
	FP-div					•						0.22	0.22	0%	0.05%	0.03%	0.02	0.01
	FP-mul					•						1.10	1.09	1%	0.01%	0.01%	0.02	0.00
	INT-add					•						2.90	3.00	3%	0.00%	0.00%	0.00	0.00
	INT-div					•						0.03	0.04	51%	0.35%	0.27%	0.37	0.10
	INT-mul					•						0.55	0.55	1%	0.02%	0.01%	0.03	0.01
Mem. Load Dependent	00016kb						•					0.46	0.41	12%	0.05%	0.36%	0.35	0.13
	00032kb						•					0.26	0.36	38%	0.06%	5.76%	36.27	15.53
	00064kb							•				0.08	0.10	23%	0.17%	3.25%	940.87	880.20
	00128kb							•				0.08	0.09	14%	0.16%	1.88%	857.39	881.92
	00256kb							•				0.08	0.09	14%	0.76%	1.17%	841.51	882.58
	00512kb								•			0.08	0.09	13%	0.44%	0.81%	831.49	882.51
	01024kb								•			0.08	0.09	12%	0.33%	0.62%	840.23	881.67
	02048kb									•		0.06	0.02	61%	0.28%	0.52%	813.89	879.67
	04096kb									•		0.05	0.02	52%	0.16%	0.46%	757.55	875.55
	08192kb									•		0.06	0.02	62%	0.25%	0.42%	594.17	867.41
	16384kb									•		0.07	0.02	65%	0.43%	0.37%	519.98	851.80
	32768kb									•		0.06	0.02	59%	0.51%	0.30%	654.83	823.14
Mem. Load Independent	00016kb						•				•	1.15	1.14	1%	0.11%	0.18%	0.05	0.06
	00032kb						•				•	1.14	1.09	4%	0.11%	5.66%	0.16	0.06
	00064kb							•			•	0.25	0.28	13%	0.16%	3.10%	867.25	881.82
	00128kb							•			•	0.25	0.27	7%	0.17%	1.70%	864.00	884.07
	00256kb							•			•	0.25	0.24	5%	0.85%	0.97%	863.54	885.12
	00512kb								•		•	0.25	0.24	6%	0.49%	0.60%	861.11	885.51
	01024kb								•		•	0.22	0.22	1%	0.27%	0.41%	854.58	885.42
	02048kb									•	•	0.08	0.02	72%	0.23%	0.32%	840.56	884.83
	04096kb									•	•	0.06	0.02	59%	0.24%	0.27%	784.72	883.42
	08192kb									•	•	0.06	0.02	63%	0.39%	0.24%	719.36	880.53
	16384kb									•	•	0.06	0.02	62%	0.64%	0.21%	718.11	874.75
	32768kb									•	•	0.07	0.02	65%	0.46%	0.19%	667.61	863.47
Mem. Store Independent	00016kb						•				•	1.16	1.14	1%	0.09%	0.18%	0.02	0.06
	00032kb						•				•	1.14	1.09	5%	0.11%	5.66%	0.13	0.06
	00064kb							•			•	0.12	0.14	14%	0.19%	3.10%	885.18	880.35
	00128kb							•			•	0.12	0.14	14%	0.20%	1.71%	887.39	883.40
	00256kb							•			•	0.12	0.14	13%	0.90%	0.98%	889.46	884.92
	00512kb								•		•	0.12	0.14	13%	0.55%	0.60%	884.44	885.69
	01024kb								•		•	0.10	0.18	69%	0.33%	0.42%	852.44	886.07
	02048kb									•	•	0.04	0.01	79%	0.38%	0.32%	741.83	886.26
	04096kb									•	•	0.02	0.01	67%	0.39%	0.27%	637.85	886.36
	08192kb									•	•	0.03	0.01	69%	0.41%	0.25%	592.45	886.41
	16384kb									•	•	0.03	0.01	71%	0.50%	0.24%	558.08	886.43
	32768kb									•	•	0.03	0.01	74%	0.62%	0.23%	529.15	886.44
GeoMean												**0.24**	**0.19**	**12%**	**0.11%**	**0.14%**	**8.42**	**5.25**
StDev														**29%**				

Table 3.7 and Table 3.8 present the comparison in terms of IPC, branch miss ratio and L1 data Misses per Kilo Instructions (MPKI) for the single-threaded microbenchmarks running on the real Core 2 Duo and Sandy Bridge machine and with SiNUCA (sim). For Core 2 Duo, the geometric mean of the absolute IPC difference for all the microbenchmarks is 12%, with a standard deviation of 29%. For Sandy Bridge, the geometric mean of the absolute IPC difference for all the microbenchmarks is 6%, with a standard deviation of 27%.

Table 3.8: Single-threaded microbenchmarks results for Sandy Bridge.

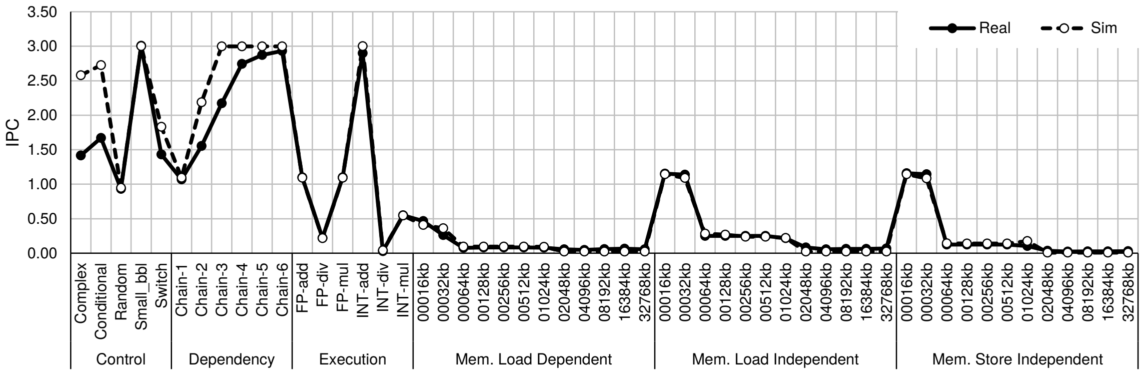
		Characteristic										IPC			Branch Miss		L1 Data MPKI		Energy (Joules)	
		Branch Predictor	Branch Miss Penalty	In-flight Branches	Register File	Functional Units	L1 Cache	L2 Cache	LLC	DRAM	Prefetcher	Real	Sim	Diff	Real	Sim	Real	Sim	Real	Sim
Control	Complex	●										1.81	2.61	44%	0.00%	0.00%	0.00	0.00	13.34	17.15
	Conditional	●										3.00	3.00	0%	0.00%	0.00%	0.00	0.00	10.82	9.16
	Random	●	●									0.96	0.97	1%	28.54%	28.57%	0.01	0.00	13.11	13.75
	Small_bbl	●		●								3.00	3.00	0%	0.00%	0.00%	0.00	0.00	4.17	17.77
	Switch	●										2.75	1.83	33%	0.00%	0.00%	0.00	0.00	4.45	10.92
Dependency	Chain-1				●							1.09	1.09	0%	0.00%	0.00%	0.00	0.00	4.78	21.16
	Chain-2				●							2.14	2.19	2%	0.00%	0.00%	0.00	0.00	1.77	11.37
	Chain-3				●							3.00	3.00	0%	0.00%	0.00%	0.00	0.00	3.94	8.23
	Chain-4				●							3.09	3.00	3%	0.00%	0.00%	0.00	0.00	3.48	8.72
	Chain-5				●							3.09	3.00	3%	0.00%	0.00%	0.00	0.00	5.56	8.23
	Chain-6				●							3.09	3.00	3%	0.00%	0.00%	0.00	0.00	8.74	8.23
Execution	FP-add					●						1.08	1.09	1%	0.01%	0.01%	0.01	0.00	10.82	10.93
	FP-div					●						0.11	0.11	0%	0.04%	0.03%	0.03	0.02	26.52	24.76
	FP-mul					●						1.08	1.09	1%	0.01%	0.01%	0.01	0.00	10.85	11.28
	INT-add					●						3.09	3.00	3%	0.00%	0.00%	0.00	0.00	9.38	10.90
	INT-div					●						0.03	0.03	19%	0.24%	0.28%	0.26	0.19	10.74	7.85
	INT-mul					●						1.08	1.09	1%	0.01%	0.01%	0.01	0.01	5.38	5.31
Mem. Load Dependent	00016kb						●					0.32	0.35	10%	2.64%	0.36%	0.28	0.22	1.03	0.91
	00032kb						●					0.28	0.29	4%	0.18%	5.76%	15.89	15.62	1.15	1.06
	00064kb							●				0.10	0.09	5%	0.21%	3.25%	874.63	880.29	3.45	3.29
	00128kb							●				0.09	0.09	5%	1.83%	1.89%	875.07	882.01	3.55	3.37
	00256kb							●				0.08	0.09	8%	0.91%	1.18%	873.41	882.66	4.19	3.44
	00512kb								●			0.07	0.06	17%	0.57%	0.81%	868.48	882.60	4.81	5.12
	01024kb								●			0.07	0.06	17%	0.39%	0.63%	859.70	881.76	4.84	5.14
	02048kb								●			0.07	0.06	19%	0.32%	0.53%	842.57	879.76	4.85	5.14
	04096kb								●			0.07	0.06	20%	0.28%	0.47%	810.86	875.64	4.90	5.15
	08192kb								●			0.08	0.06	25%	0.26%	0.42%	752.75	867.50	4.93	5.16
	16384kb								●			0.08	0.06	21%	0.24%	0.37%	658.64	851.88	5.55	5.21
	32768kb									●		0.06	0.02	65%	0.23%	0.31%	528.70	823.22	7.91	14.98
Mem. Load Independent	00016kb						●				●	1.14	1.15	0%	0.08%	0.18%	0.12	0.10	0.61	0.71
	00032kb						●				●	1.13	1.13	0%	0.09%	5.66%	0.19	0.11	0.61	0.71
	00064kb							●			●	0.56	0.50	10%	0.10%	3.10%	880.57	881.87	1.25	1.49
	00128kb							●			●	0.54	0.50	8%	0.10%	1.71%	882.16	884.11	1.27	1.49
	00256kb							●			●	0.35	0.45	26%	0.85%	0.98%	881.68	885.16	1.97	1.67
	00512kb								●		●	0.19	0.18	6%	0.48%	0.61%	879.48	885.55	3.69	4.00
	01024kb								●		●	0.19	0.18	7%	0.30%	0.42%	875.49	885.47	3.73	4.06
	02048kb								●		●	0.20	0.18	8%	0.22%	0.32%	867.18	884.87	3.74	4.05
	04096kb								●		●	0.20	0.18	9%	0.19%	0.27%	852.45	883.47	3.77	4.06
	08192kb								●		●	0.20	0.18	10%	0.18%	0.24%	821.45	880.57	3.80	4.02
	16384kb								●		●	0.17	0.18	6%	0.19%	0.21%	765.52	874.80	4.60	4.04
	32768kb									●	●	0.10	0.07	28%	0.19%	0.19%	674.21	863.52	8.68	10.43
Mem. Store Independent	00016kb						●				●	1.13	1.14	1%	0.08%	0.19%	0.12	0.10	0.61	0.66
	00032kb						●				●	1.12	1.13	1%	0.09%	5.66%	0.26	0.11	0.61	0.99
	00064kb							●			●	0.17	0.28	62%	0.11%	3.10%	1754.80	880.40	3.91	2.51
	00128kb							●			●	0.17	0.28	65%	0.11%	1.71%	1759.56	883.44	3.99	2.51
	00256kb							●			●	0.16	0.41	150%	0.85%	0.98%	1760.56	884.97	4.27	1.80
	00512kb								●		●	0.12	0.18	45%	0.49%	0.61%	1756.31	885.73	5.78	3.79
	01024kb								●		●	0.12	0.18	46%	0.31%	0.42%	1749.05	886.12	5.93	3.81
	02048kb								●		●	0.12	0.18	44%	0.23%	0.32%	1733.53	886.31	5.94	3.83
	04096kb								●		●	0.12	0.18	42%	0.19%	0.28%	1704.49	886.40	5.96	3.84
	08192kb								●		●	0.13	0.18	38%	0.19%	0.25%	1645.38	886.45	5.98	3.84
	16384kb								●		●	0.11	0.18	57%	0.20%	0.24%	1537.79	886.48	7.30	3.85
	32768kb									●	●	0.08	0.02	72%	0.21%	0.23%	1357.70	886.49	17.11	28.56
GeoMean												**0.34**	**0.34**	**6%**	**0.07%**	**0.14%**	**8.68**	**6.77**	**4.24**	**4.70**
StDev														**27%**						

Figure 3.2 depicts the IPC results for the real machines and the simulation.

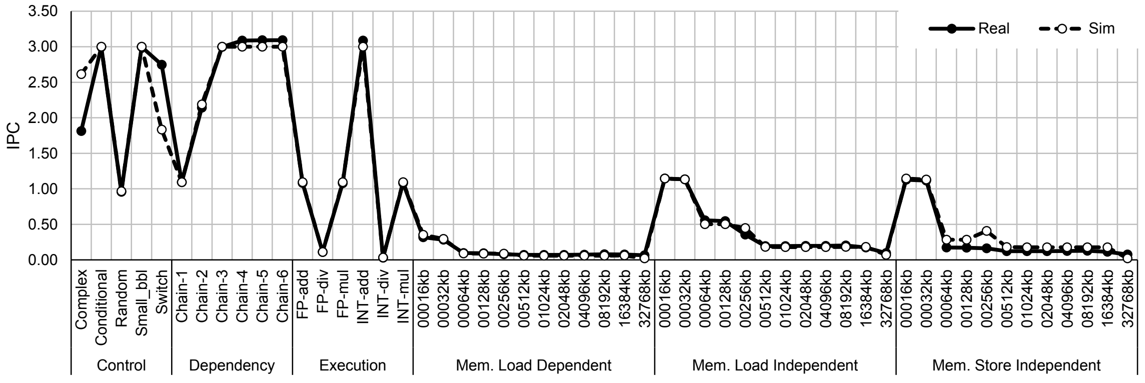
Control benchmarks: Evaluating the control benchmark *Small_ bbl*, we conclude that the maximum number of parallel predicted branches in execution inside the pipeline is equal to 8. With the *Random* benchmark, we calculate the misprediction penalty as 20 cycles. For the other control benchmarks, we can observe the different behavior from our PAs branch predictor to the real machine implementation.

Dependency benchmarks: Observing the behavior of dependency benchmarks, we can notice that for *Chain-1* and *Chain-6* the simulation obtains very close results to the real machines. Inside the simulator, all the dependencies are solved one cycle after the result is available. For *Chain-1*, we can observe that the latency for data forwarding is modeled correctly. However, the results from *Chain-2* to *Chain-5* show that the real machines have

Figure 3.2: Results for the single-threaded microbenchmarks.



(a) Core 2 Duo.



(b) Sandy Bridge.

a limited maximum number of data forwardings per cycle. For *Chain-6*, the instruction parallelism is high enough to hide this bottleneck.

Execution benchmarks: Although we obtained very accurate results for the integer and floating point applications (less than 4% of difference), we notice that *INT-div* has a very low IPC, which is caused by the high latency of the functional unit. Due to the low IPC, the performance difference is high compared to the simulation.

Memory benchmarks: For the memory benchmarks, the execution of the 16 KB sized benchmarks requires 8 repetitions to access the full vector. This happens because inside the application, it was modeled in such a way that every element is in a different cache line (64 bytes), and each loop repetition has 32 memory accesses. In the same way, the 32 KB vector size requires 16 repetitions. The increase in the number of repetitions leads to different behavior from the branch predictor. Since we do not implement a loop predictor, the PAs predictor requires a signature size of 16 bits to correctly predict a pattern of 16 takens and 1 not taken, such as the one present when accessing 32 KB.

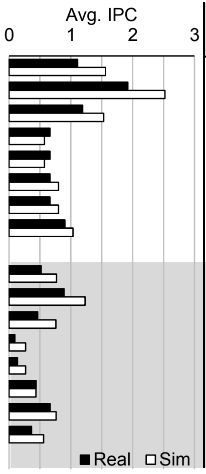
We also notice that the prefetcher has a great influence for the memory benchmarks with 16 KB and 32 KB vector sizes. This is because an aggressive prefetch can start trashing useful data from the L1 data cache.

We can notice that memory applications that stress the DRAM have a lower performance inside the simulator. This can be caused by possible improvements in the memory controller system in the real machine, such as priority schemes for the memory requests.

For the *Memory Store Independent* results on Sandy Bridge, we consider that the real processor counts every store miss twice, one being a write miss and also a read miss, while our simulator counts only the write miss. This explains that the number of misses per thousand instructions is higher than one thousand. Dividing the obtained number of L1 MPKI for the real machine would give us a reasonable approximation of the real L1 MPKI number, which also matches the simulator results.

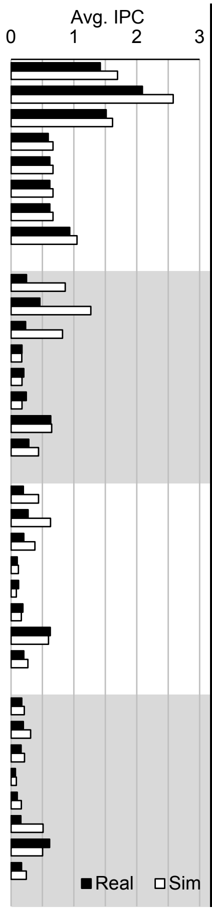
3.4.2.2 Multi-Threaded Microbenchmarks

Table 3.9: Multi-threaded microbenchmarks results for Core 2 Duo.



		Average IPC			Branch Miss		L1 DATA MPKI	
		Real	Sim	Diff	Real	Sim	Real	Sim
1 Thread	barrier_all_together	1.11	1.56	41%	0.00	0.00	0.01	0.02
	barrier_master_slow	1.92	2.52	31%	0.00	0.00	0.01	0.00
	critical	1.19	1.53	28%	0.00	0.00	0.06	0.02
	false_sharing_pad00	0.66	0.57	14%	0.00	0.00	0.02	0.01
	false_sharing_pad08	0.66	0.57	14%	0.00	0.00	0.02	0.01
	false_sharing_pad24	0.66	0.80	21%	0.00	0.00	0.02	0.01
	false_sharing_pad56	0.66	0.80	21%	0.00	0.00	0.02	0.01
	GeoMean	**0.90**	**1.03**	**23%**	**0.00**	**0.00**	**0.02**	**0.01**
	StDev			**10%**				
2 Threads	barrier_all_together	0.52	0.77	48%	0.01	0.00	12.50	38.01
	barrier_master_slow	0.89	1.23	38%	0.00	0.00	5.22	4.39
	critical	0.46	0.76	63%	0.01	0.00	13.88	65.90
	false_sharing_pad00	0.10	0.27	180%	0.00	0.00	224.85	385.55
	false_sharing_pad08	0.14	0.27	95%	0.00	0.00	193.96	399.95
	false_sharing_pad24	0.44	0.43	1%	0.00	0.00	144.41	185.98
	false_sharing_pad56	0.66	0.76	14%	0.00	0.00	0.01	0.01
	GeoMean	**0.36**	**0.56**	**30%**	**0.00**	**0.00**	**13.55**	**23.30**
	StDev			**60%**				

Table 3.10: Multi-threaded microbenchmarks results for Sandy Bridge.



		Average IPC			Branch Miss		L1 DATA MPKI		Energy (Joules)	
		Real	Sim	Diff	Real	Sim	Real	Sim	Real	Sim
1 Thread	barrier_all_together	1.42	1.69	19%	0.00	0.00	0.01	0.00	3.39	1.39
	barrier_master_slow	2.09	2.58	23%	0.00	0.00	0.00	0.00	7.22	5.16
	critical	1.51	1.61	7%	0.00	0.00	0.02	0.00	2.29	1.46
	false_sharing_pad00	0.59	0.67	12%	0.00	0.00	0.01	0.00	11.53	10.49
	false_sharing_pad08	0.62	0.67	8%	0.00	0.00	0.01	0.00	10.91	10.15
	false_sharing_pad24	0.62	0.67	8%	0.00	0.00	0.01	0.00	10.88	10.15
	false_sharing_pad56	0.62	0.67	8%	0.00	0.00	0.01	0.00	10.93	11.72
	GeoMean	**0.94**	**1.05**	**11%**	**0.00**	**0.00**	**0.01**	**0.00**	**7.02**	**5.39**
	StDev			**7%**						
2 Threads	barrier_all_together	0.25	0.86	247%	0.00	0.00	14.05	12.51	4.79	0.87
	barrier_master_slow	0.46	1.27	177%	0.00	0.00	7.04	2.64	6.67	2.76
	critical	0.23	0.82	253%	0.00	0.00	17.13	17.35	5.64	0.91
	false_sharing_pad00	0.17	0.17	2%	0.00	0.00	19.68	291.65	10.63	19.78
	false_sharing_pad08	0.20	0.18	13%	0.00	0.00	16.58	243.93	9.43	19.33
	false_sharing_pad24	0.24	0.18	28%	0.00	0.00	12.44	244.43	7.96	20.75
	false_sharing_pad56	0.63	0.64	2%	0.00	0.00	0.02	0.00	6.26	7.17
	GeoMean	**0.28**	**0.44**	**28%**	**0.00**	**0.00**	**5.45**	**10.25**	**7.09**	**5.35**
	StDev			**117%**						
4 Threads	barrier_all_together	0.19	0.44	124%	0.01	0.00	12.27	32.66	4.52	0.55
	barrier_master_slow	0.27	0.63	133%	0.00	0.00	9.05	8.00	5.49	1.49
	critical	0.20	0.38	87%	0.01	0.02	13.18	43.93	5.98	0.64
	false_sharing_pad00	0.10	0.12	16%	0.00	0.00	24.99	383.32	12.89	15.07
	false_sharing_pad08	0.12	0.08	30%	0.00	0.00	23.30	281.43	10.05	22.13
	false_sharing_pad24	0.19	0.16	13%	0.00	0.00	19.43	222.61	7.33	12.83
	false_sharing_pad56	0.62	0.60	5%	0.00	0.00	0.03	0.00	3.94	4.81
	GeoMean	**0.20**	**0.27**	**33%**	**0.00**	**0.00**	**6.43**	**16.94**	**6.62**	**3.77**
	StDev			**55%**						
8 Threads	barrier_all_together	0.17	0.21	23%	0.00	0.00	9.34	54.50	4.66	0.40
	barrier_master_slow	0.19	0.31	60%	0.00	0.00	8.75	16.22	4.82	0.85
	critical	0.16	0.22	35%	0.01	0.02	10.40	68.88	7.36	0.41
	false_sharing_pad00	0.07	0.08	24%	0.00	0.00	27.51	305.18	15.01	10.36
	false_sharing_pad08	0.10	0.16	67%	0.00	0.00	25.52	178.06	10.52	5.35
	false_sharing_pad24	0.16	0.51	225%	0.00	0.00	20.58	0.00	7.17	1.72
	false_sharing_pad56	0.61	0.50	18%	0.00	0.00	0.12	0.01	2.81	2.86
	GeoMean	**0.17**	**0.24**	**44%**	**0.00**	**0.00**	**7.59**	**5.08**	**6.56**	**1.68**
	StDev			**73%**						

Table 3.9 presents the comparison for the multi-threaded microbenchmarks running on the real Core 2 Duo machine and with SiNUCA. The table shows the average IPC between the cores, the branch miss ratio and L1 data MPKI. The geometric mean of the absolute IPC difference for 1 thread execution is 23%, with a standard deviation of 10%, the difference for 2 threads execution is 30% with a standard deviation of 60%.

Table 3.10 presents the comparison for the multi-threaded microbenchmarks running on the real Sandy Bridge machine and with SiNUCA. The table shows the average IPC between the cores, the branch miss ratio, L1 data MPKI and also the energy consumption. The geometric mean of the absolute IPC difference for 1, 2, 4 and 8 threads are 11%, 28%, 33%, 44%, with standard deviations of 7%, 117%, 55 % and 73% respectively.

The parallel benchmarks have a simple code, formed by easy to predict loops, making the number of branch mispredictions close to zero.

The *False Sharing* application presents the higher numbers of L1 MPKI, where our evaluations changing the pad size, reduces the L1 MPKI as less threads falsely share the same cache line. The L1 MPKI difference between the real processor and SiNUCA is partially explained by the possible differences in the directory line lock policy, where the real machine could give the control of the cache line for a single core for a longer period of time, in order to reduce the successive cache line invalidations. Considering that for this application multiple cache-to-cache operations can occur during cache misses, the real machine can account those misses in a different manner, however no detailed information about the implementation of the hardware counters is available.

During the execution on the simulator of the 8 threaded False Sharing application with a 24 bytes padding, the first L1 caches to request the shared cache line obtained the lock on the directory for most of the time until these threads finish executing. Only then, the other threads can continue their execution, leading to a higher average IPC and a reduced L1 MPKI on the simulator.

3.4.2.3 Commercial Workloads

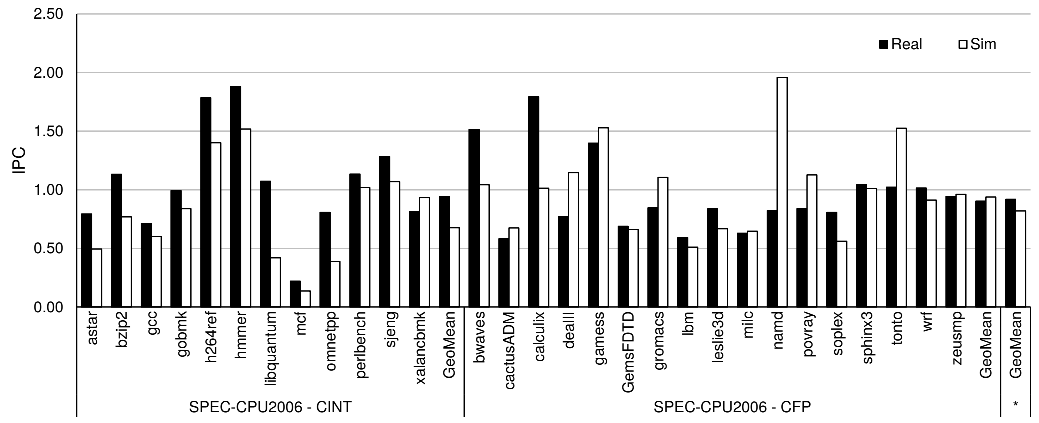
Figure 3.3 presents the IPC for the SPEC-CPU2006 benchmark suite running on the Core 2 Duo and Sandy Bridge machines. For the Core 2 Duo, the geometric mean of IPC difference is 19% for all suite, with a standard deviation of 27%. For Sandy Bridge machine, the geometric mean of IPC difference is 12% for all suite, with a standard deviation of 16%.

For the SPEC-CPU2006 benchmarks, the major source of difference was the branch predictor (see Appendix A). We could see that applications behave differently when switching between PAs and GAs branch predictors, in such way that a hybrid predictor is being considered to reduce the gap between the real and simulator difference.

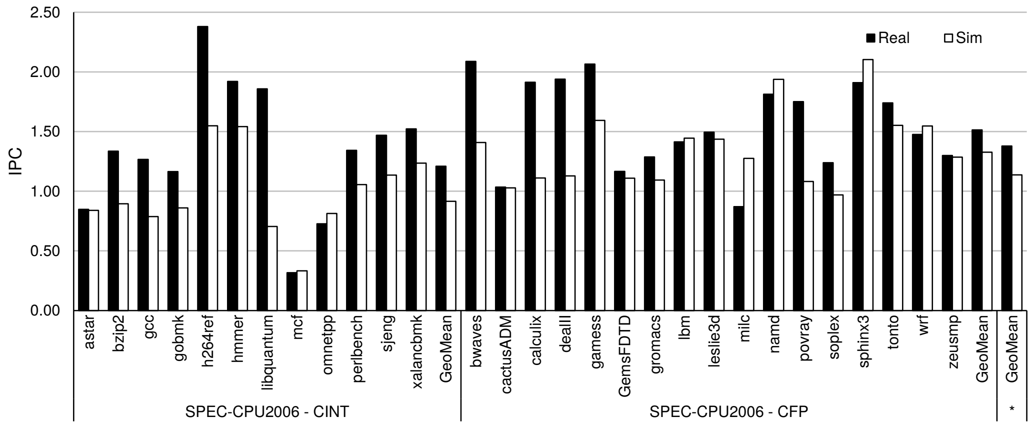
Figure 3.4 presents the speed-up for the SPEC-OMP2001 benchmark suite running on Core 2 Duo and Sandy Bridge machines. For the Core 2 Duo, the geometric mean of speed-up difference is 6% for these applications executing with 2 threads, with a standard deviation of 10% for all suite applications. For the Sandy Bridge, the geometric mean of speed-up difference for 2, 4 and 8 threads are 10%, 27%, 41%, with standard deviations of 17%, 55%, 70% respectively.

Figure 3.5 presents the speed-up for the NAS-NPB benchmark suite running on the Core 2 Duo and Sandy Bridge machine. For Core 2 Duo, the geometric mean of speed-up difference is 7% for these applications using 2 threads, with a standard deviation of 8% for all suite applications. For Sandy Bridge, the geometric mean of speed-up difference

Figure 3.3: Results for the SPEC-CPU2006 suite.



(a) Core 2 Duo.



(b) Sandy Bridge.

for 2, 4 and 8 threads are 6%, 19%, 23% with standard deviations of 5%, 23% and 58%, respectively.

It is possible to observe that the speedup and IPC results for 2 threads have a lower difference between the real and simulated processor, compared to an increasing number of threads. As the number of threads increases, the synchronization has an increased effect on the execution time. This difference can change the threads lock priority, the cache misses that each thread causes, and others.

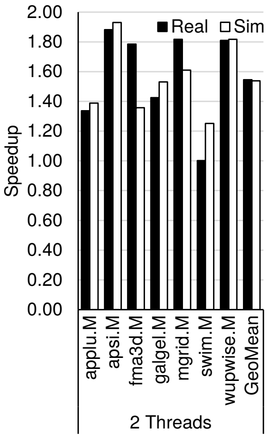
3.4.3 Sources of Difference

There are multiple sources of difference between SiNUCA and the real processor, because commercial processors have undisclosed details which we tried to approximate. However, we believe that the main differences were caused by the following components:

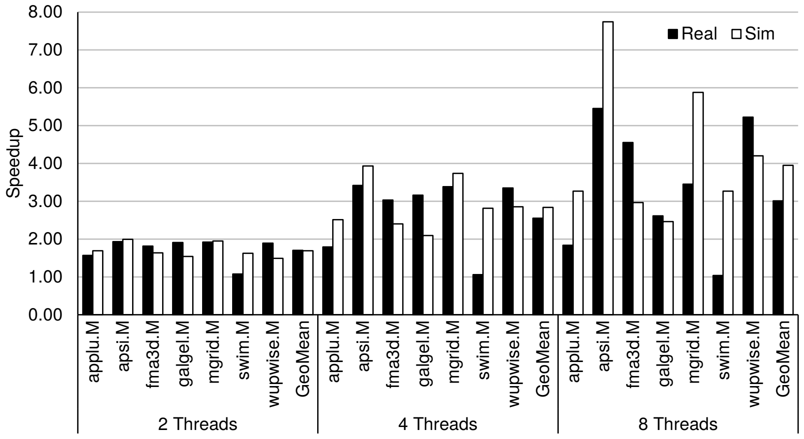
Branch Predictor: The first source of differences regarding the two-level branch predictor implemented inside SiNUCA is that this well known mechanism is not guaranteed to be exactly the same as the one implemented in a real machine. Since this mechanism's prediction is sensitive to its implementation, small implementation differences can lead to different predictions.

During branch mispredictions, the simulator stops fetching new instructions until the branch is solved. However, in a real machine, the processor would continue fetching and executing instructions until detection of a misprediction causes a rollback. In such cases,

Figure 3.4: Results for the SPEC-OMP2001 suite.

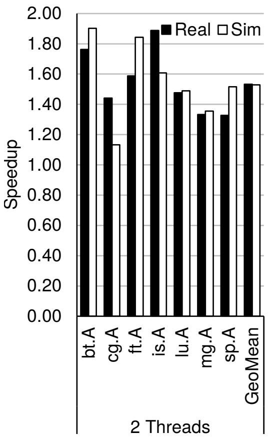


(a) Core 2 Duo.

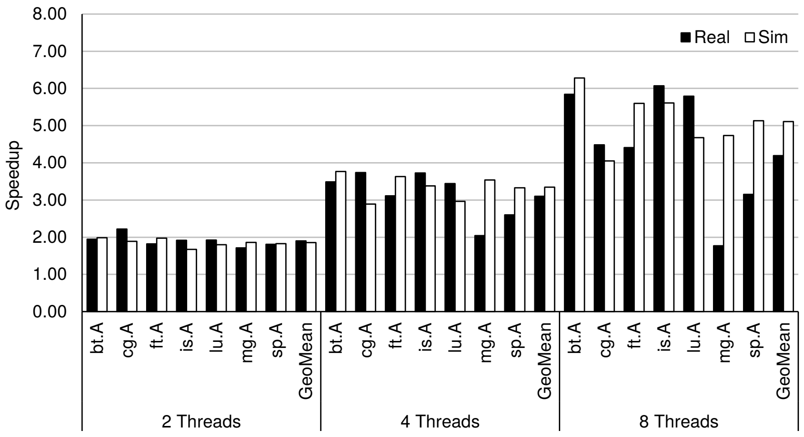


(b) Sandy Bridge.

Figure 3.5: Results for the NAS-NPB suite.



(a) Core 2 Duo.



(b) Sandy Bridge.

the cache memory can be in a different state, since the wrong path may perform memory loads. These operations will change the cache state.

Functional Units: The real machine contention in the number of ports connecting the reservation station and functional units is not modeled inside SiNUCA, thus representing a source of inaccuracy.

Additionally, the latencies for instructions that use the same functional unit can vary in a real machine depending on the type of the operands and their precisions. However, inside SiNUCA, the execution latencies are defined by the functional unit used, and not by the instruction itself. Such modification in the simulator would be impractical since there are hundreds of instructions inside the x86-64 Instruction Set Architecture (ISA). Thus, latencies for single precision and double precision floating point are the same. The latencies configured for our evaluations were chosen to take into account that most benchmarks use double precision floating point.

Memory Disambiguation: A real processor memory disambiguation unit can cause differences in the performance. However, the implementation details for this unit are unknown for commercial processors, and this could cause performance differences when comparing the simulator to the real machine (DOWECK, 2006). During our experiments, perfect disambiguation was modeled, enabling reads to be scheduled before writes, as long as the operation addresses do not overlap.

Translation Lookaside Buffer: Although the Translation Look-aside Buffer (TLB) is present in most real processors, SiNUCA does not implement this component. This simplification was made because we consider that for most of the benchmarks this component would not influence performance. However the energy consumption of this component was fully modeled, considering the number of L1 accesses.

A second source of difference that can be caused by the address translation, is the different virtual to physical address mapping present in the real and simulated processor. Any different address mapping can cause different cache statistics. However, it is expected that the general behavior of the cache system would be the same, even for different address mappings.

Cache Prefetcher: The memory prefetching algorithms implemented in SiNUCA are well known techniques presented in multiple papers. However, the information about the real processor prefetcher is missing some details. Thus, it is not guaranteed that the simulated prefetchers would perform exactly the same as the real machine. This difference also happens because small changes in the prefetcher parameters, such as number of strides, can cause a high impact on the final MPKI.

Trace Cache: The real machine used in our experiments has a trace instruction cache, which reduces the fetch and decode latency when executing loops. Such mechanism is also able to reduce the energy consumption. However, it is not yet implemented in SiNUCA.

Cache Memories: Different MSHR scheduling policies can lead to different performance results. If the policy gives priority to processor load requests over the prefetches or over write transactions, this can generate different performance results. The actual policy implemented in SiNUCA is based on First-Come First-Serve (FCFS). This policy is only not respected when the request corresponds to a cache line that is locked in the directory by another cache, in this case the cache serves the next oldest request.

Memory Controller: The memory controller details in a real hardware were simplified in the SiNUCA implementation, in such a way that the main operations would perform in the same way, but some corner cases can perform differently. However, most of the DDR latencies are being simulated, such as Added Latency for column accesses (AL), Column Address Strobe (CAS), Column to Column Delay (CCD), Column Write Delay (CWD), Four row Activation Window (FAW), Row Address Strobe (RAS), Row Cycle (RC), RAS to CAS Delay (RCD), Row Precharge (RP), Row to Row activation Delay (RRD), Read To Precharge (RTP), Write To Read delay time (WR) and Write Recovery time (WTR).

The memory scheduling policy can also cause performance differences, depending on the real hardware implementation. For our experiments, the simple open-row first policy was used.

General Parameters: Parameters, such as the size of internal buffers, size of tables and number of bits used for branch predictors and prefetchers may affect the final performance of a system in different ways. Considering that most of the processor internals are intellectual property, the internal parameters are not public. Such unknown parameters were modeled considering a balanced design and also considering the parameters commonly used in papers from main conferences.

Operating System Interference: The traces generated to feed the simulator are always generated considering only the application instructions. No operating system instructions are tracked. The lack of operating system in our traces guarantee that only the application behavior will interfere in the final performance, without external noise.

Multi-Thread synchronization: OpenMP offers three thread wait policies (active, passive and hybrid wait). For the active wait, it uses a spin lock to keep the threads running waiting in the barrier. The passive wait automatically puts the thread to sleep when it reaches any barrier. The hybrid wait policy offers a balanced design which makes the threads wait actively for a period of time before the thread is sent to sleep.

For multi-threaded benchmarks, the simulator respects the synchronization barriers and atomic points. However, it does not simulate active wait nor the sleep and wake-up instructions from the OS. We observed in our experiments that when using the passive wait policy, the fact of not simulating the sleep/wakeup instructions could cause differences in our multi-threaded microbenchmarks. In order to reduce this source of difference, the hybrid OpenMP wait policy was used during our evaluations.

3.4.4 Simulator Performance

The results regarding the simulator performance refer to how fast the traces can be simulated. For this evaluation, all the benchmarks from SPEC-CPU2006 and SPEC-OMP2001 suites were simulated with SiNUCA modeling a Sandy Bridge processor, executing on a real Intel Sandy Bridge machine.

Figure 3.6: Sandy Bridge simulation performance for the SPEC-CPU2006 suite.

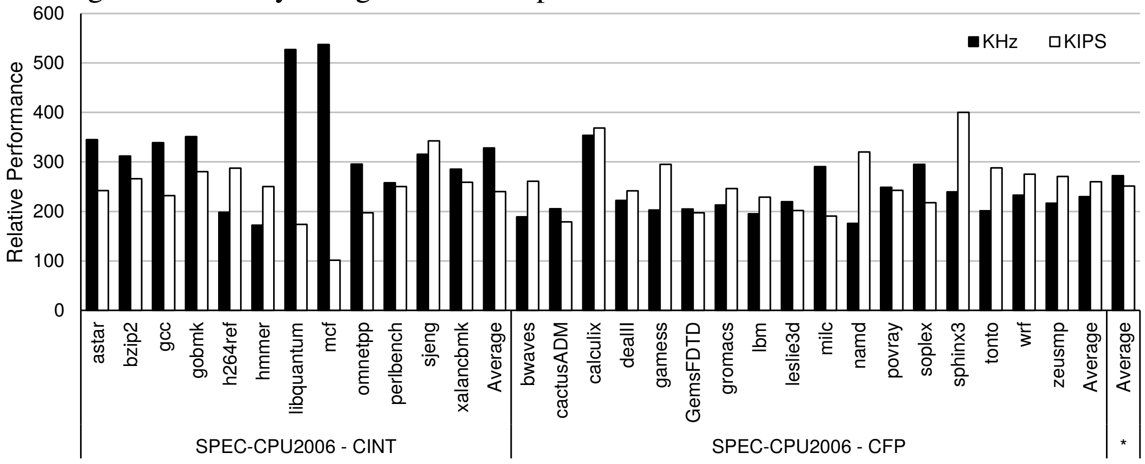
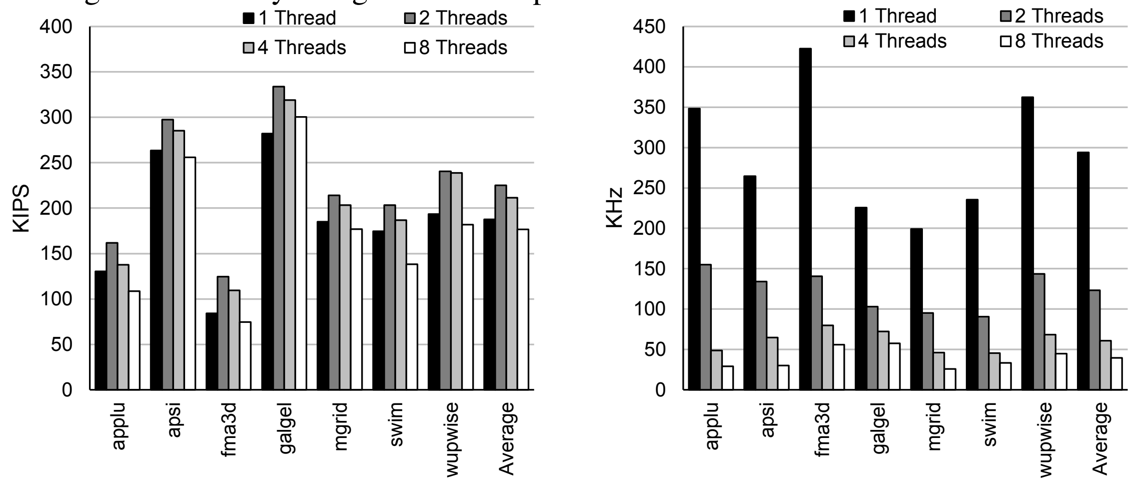


Figure 3.6 presents the simulator performance when executing the SPEC-CPU2006 applications. It is possible to see that SiNUCA varies from 100 to 400 Kilo Instructions per Second (KIPS), with an average of 250 KIPS. Considering the number of cycles being simulated and the simulation time, we obtain the maximum frequency at which our simulator simulates the processor clock, which varies from 170 kHz to 530 kHz, with an average of 270 kHz.

To evaluate the performance when simulating multi-threaded benchmarks, Figure 3.7 presents the SPEC-OMP2001 simulation performance results for different numbers of threads. The KIPS varies from 75 to 330, and the simulation frequency varies from 25 to 420 kHz.

Figure 3.7: Sandy Bridge simulation performance for the SPEC-OMP2001 suite.



Although the increase on number of threads reduces the cycles simulated per second, SiNUCA keeps the number of instructions simulated at the same level (200 KIPS).

To give one example of the simulator performance, to execute a single threaded benchmark with 200 million instructions with SiNUCA takes on average only 14 minutes. To simulate 8 threads executing 200 million instructions each, takes on average 19 minutes. Such performance enables the user to perform multiple experiments in a reasonable time, eliminating a very common problem in computer architecture evaluations.

3.5 Summary

We presented SiNUCA, a performance and energy validated simulator, which is integrated with McPAT to enable energy modeling. It supports simulation of emerging techniques and it is easy to extend.

Our validation process shows an average difference on performance of 12% for single-threaded and 26% for multi-threaded microbenchmarks when simulating the Core-2Duo machine. An average difference on performance of 6% for single-threaded and 29% for multi-threaded benchmarks when simulating a Sandy Bridge architecture.

Considering the energy modeling using McPAT together with our simulator, we obtain an average difference of 18% for single-threaded and 46 % for multi-threaded microbenchmarks related to energy consumption when modeling the Sandy Bridge architecture.

The results regarding the simulator performance show that SiNUCA enables computer architects to evaluate new techniques in a reasonable time, simulating at 270 kHz (250 KIPS) on average.

4 DEAD SUB-BLOCK PREDICTOR (DSBP)

As explained in Section 2.1, energy inefficiency occurs on two levels: 1) on the cache line level, where a line is kept alive in the cache for much longer than it is needed, and 2) on the sub-block level, when parts of a cache line which will never be used are brought into the cache, and also when active sub-blocks become dead after a few accesses but are kept alive until the line is evicted.

Prior work has made attempts at achieving these benefits by, for example, predicting when a line is last accessed, and then powering it off (KAXIRAS; HU; MARTONOSI, 2001; ABELLA et al., 2005). On the sub-block level a prior work presented a mechanism (CHEN et al., 2004) that only brings into the cache the useful sub-blocks. In this chapter, we show that turning off dead sub-blocks can significantly improve energy savings in the cache hierarchy. In fact, by considering sub-block usage patterns, our mechanism increases the potential for energy savings compared to a perfect hypothetical mechanism that turns off an entire cache line immediately after it is last accessed.

We propose the Dead Sub-Block Predictor (DSBP) (ALVES et al., 2012) to improve energy efficiency of cache memories. DSBP uses recent history information to predict which sub-block(s) will be useful and how many accesses each sub-block will receive before it becomes dead. DSBP's main goal is to reduce dynamic and static energy consumption by bringing only useful sub-blocks into the cache, and also by turning off active sub-blocks after their predicted number of accesses. We also use DSBP to improve the existing cache replacement policy by prioritizing dead lines (that is, cache lines with all sub-blocks turned off) for eviction. We find that this policy effectively offsets the additional cache misses DSBP may cause when it mispredicts the usage pattern of a cache line.

The main contributions of the DSBP mechanism are:

Sub-block usage predictor: We present a mechanism to predict and allocate only the useful sub-blocks of each cache line. Unlike prior work, which requires an access to the prediction table after each cache line access, we access the predictor structure only when the mechanism is training a new usage pattern. On average, our mechanism accesses its global structure on only 60% of the cache memory accesses.

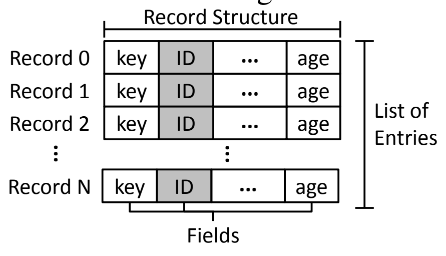
Dead sub-block predictor: Our mechanism also predicts when each sub-block inside a cache line becomes dead. To our knowledge, this is the first usage predictor that acts on a sub-block level, turning off dead sub-blocks and saving 36% energy on average compared to a traditional cache design.

Earlier eviction of dead lines: Our mechanism improves the cache replacement algorithm. The sub-block predictor gives feedback to the replacement algorithm by marking dead lines as future victim lines as soon as they become dead.

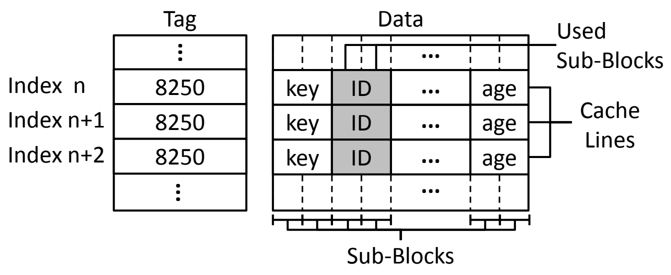
Figure 4.1: Scenario with low cache sub-block usage.

```
for (i=0; i<10000; i++){
  if (record[i].ID == 100){
    break;
  }
}
```

(a) Code example.



(b) Data structure layout in memory.



(c) Cache memory, way number 0.

4.1 Motivation

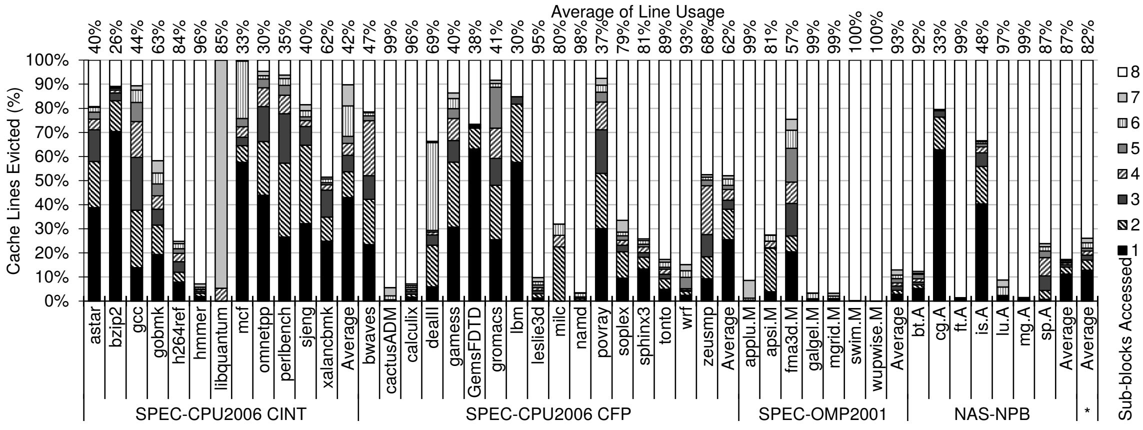
We present an example that illustrates the need for sub-block level cache management, and we also present statistics about cache line usage for different benchmark applications from the SPEC-CPU2006, SPEC-OMP2001 and NAS-NPB suites, presented in Section 3.3. The results are based on experiments simulating a Sandy Bridge machine, using an 8 byte sub-block size on the cache memories. Further simulation details can be found in Section 4.3.

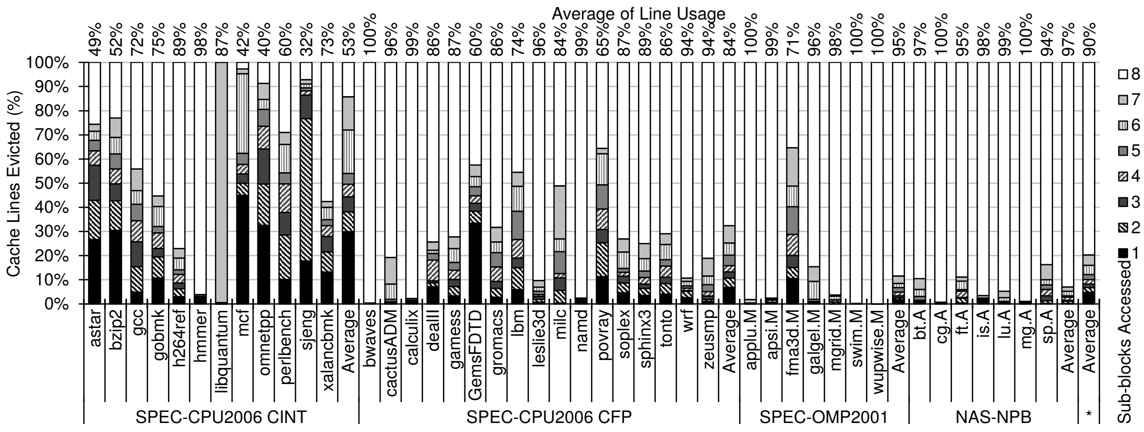
During program execution, cache lines typically have a low sub-block usage. As an example, consider the code fragment shown in Figure 4.2(a). The memory access behavior of this code is illustrated in Figure 4.1. The code iterates over a list of records, searching for the record whose ID is equal to 100. Each record contains several fields and occupies 64 bytes, as shown in Figure 4.2(b). The code accesses only the ID field of each record, and therefore only that field is accessed by the processor. However, traditional caches will bring in all fields of each record as shown in Figure 4.2(c). Only the sub-blocks which store the ID field are useful, while the other sub-blocks remain unused until the line is evicted, resulting in considerable energy waste.

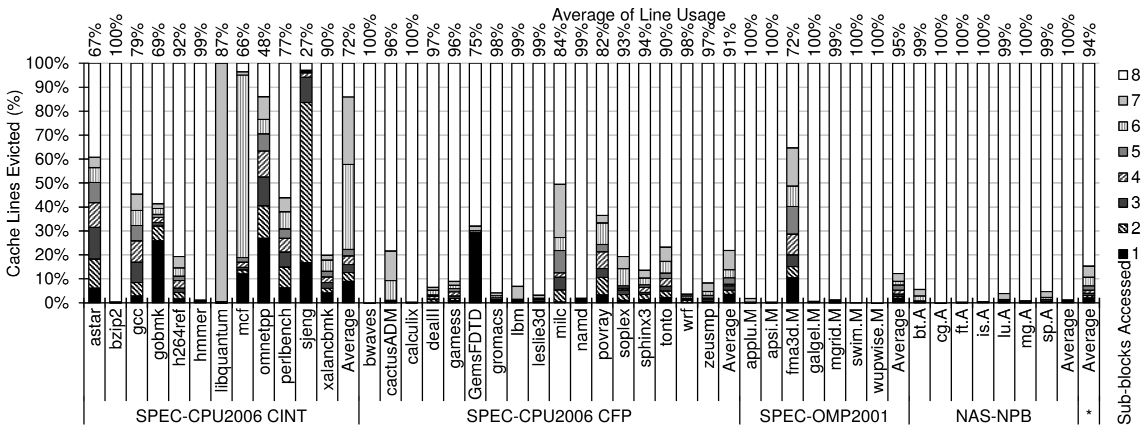
4.1.1 Cache Line Usage on the Sub-Block Level

Figure 4.2 shows how cache lines of 64 bytes are used at the sub-block granularity of 8 bytes per block for the L1, L2 and LLC across multiple benchmark suites. For example, looking at the leftmost bar from Figure 4.3(a), one can see that 38% of L1 cache lines were evicted with only one sub-block accessed (for the *astar* application from SPEC-CPU2006 integer). The average number of sub-blocks accessed in a cache line for each application is presented at the top of each bar. We can therefore conclude that a significant number of cache sub-blocks that are brought into the cache are never used, thereby wasting cache energy, capacity and bandwidth.

Figure 4.2: Number of sub-blocks accessed per cache line.


(a) L1 cache.

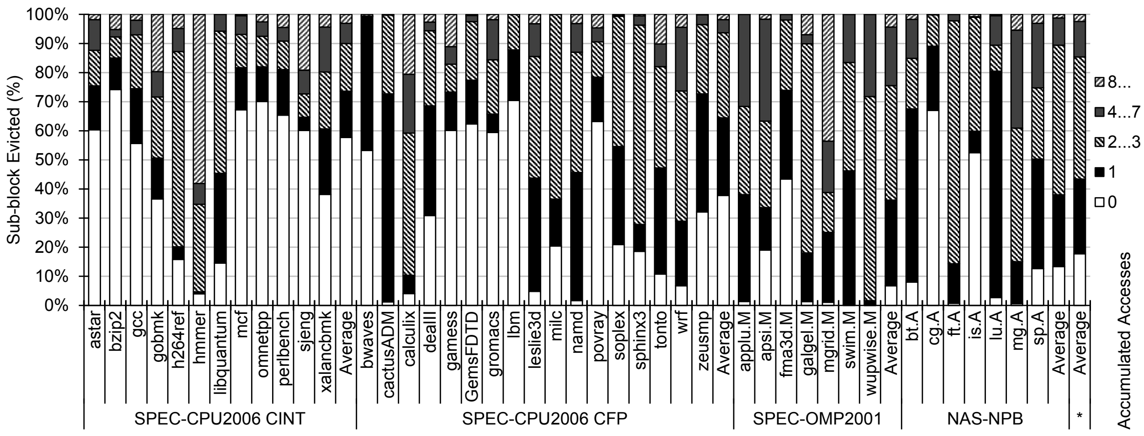

(b) L2 cache.


(c) LLC cache.

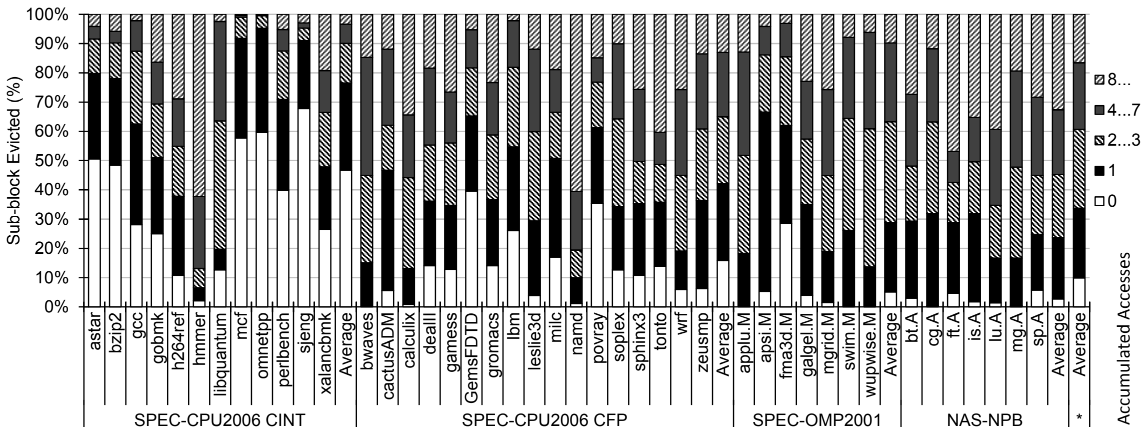
We can observe that the percentage of cache line usage increases over the cache levels, from the L1 to the LLC. It happens because the line stays for a longer time in the lower level caches (closer to the main memory) than in the higher level caches (closer to the processor). This means that cache lines present in the LLC have a higher probability of being completely accessed.

Comparing the single and multi-threading benchmarks, we can see that the usage is higher for the multi-threaded. This happens because the scientific applications act more well behaved (for example, accessing all the elements of the matrix).

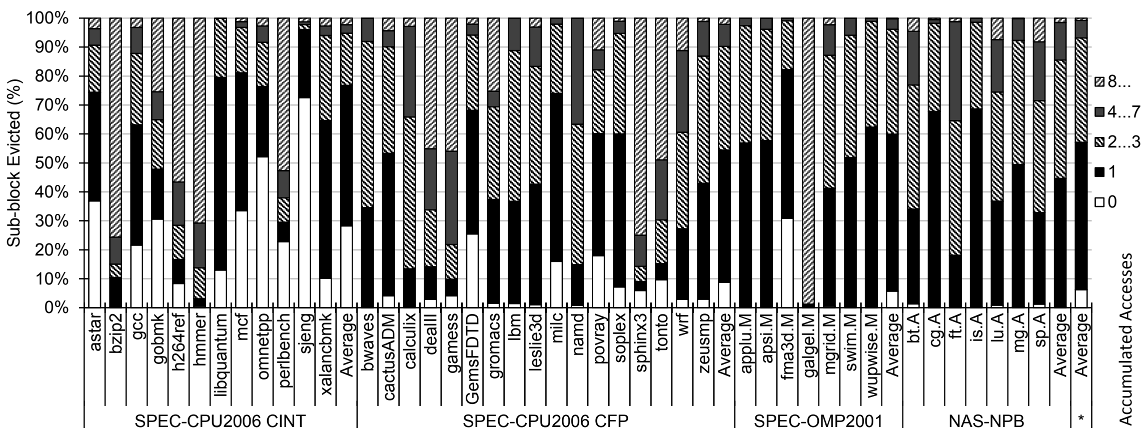
Figure 4.3: Number of sub-block accesses before cache line eviction.



(a) L1 cache.



(b) L2 cache.



(c) LLC cache.

Figure 4.3 shows the number of accesses a sub-block receives before the line is evicted from the L1, L2 and LLC. The figure shows that, on average for the L1 data cache, 18% of the sub-blocks are never used, and about 68% of sub-blocks are used between one and three times. This once again shows opportunity for energy savings. Sub-blocks that are never used should not be brought into the cache. Furthermore, most of the active sub-blocks can be powered off after just a few accesses, saving even more energy. This holds true for the lower level caches (L2 and LLC) as well. On average, the percentage of sub-blocks that receive less than 4 accesses before their eviction on the L1, L2 and LLC is 85%, 61% and 93% respectively.

Although the LLC cache holds the cache lines for a longer period of time, the L1 cache filters most of the processor requests. For this reason, the number of accesses per sub-block on the L1 tends to be higher than on lower level caches. Similar to the number of sub-blocks used per line, the single-threaded benchmarks have a lower number of accesses per sub-block than the multi-threaded benchmarks.

4.1.2 Potential for Energy Savings in L1 Cache

Figure 4.4: Potential for L1 cache energy savings for two oracle predictors.

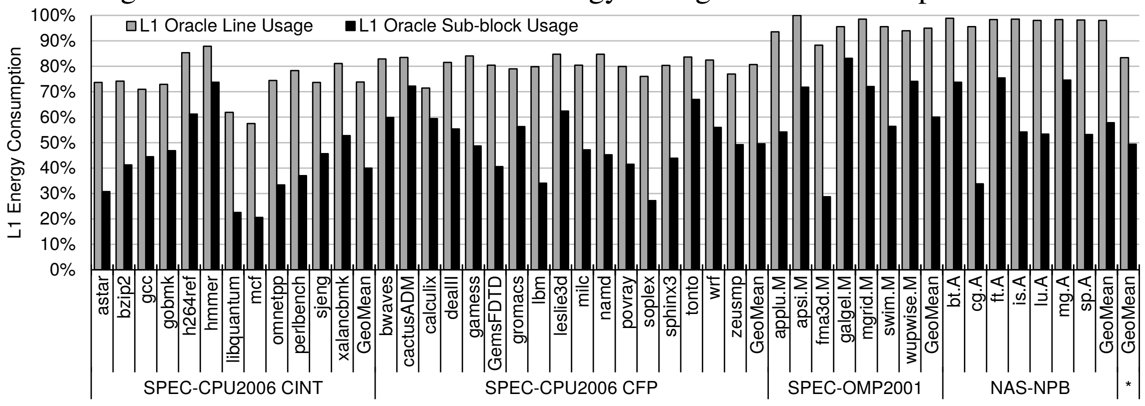


Figure 4.4 shows the potential for L1 static and dynamic energy savings by presenting results for two oracles implemented on the L1 cache: 1) An oracle dead line predictor that saves static power by turning off cache lines as soon as they receive their last access. 2) An oracle dead sub-block predictor that saves both static and dynamic power by never bringing and never turning on unused sub-blocks in the cache and in addition powers off active sub-blocks as soon as they are last accessed.

The results are normalized to a baseline without any dead line predictor. Note that these oracles operate without the use of any additional prediction structures (for example, they consume no additional energy and can only save energy) and therefore truly represent ideal scenarios.

The oracle dead line predictor reduces L1 cache energy (17% on average) by turning off dead lines as soon as possible. However, its effectiveness is limited by the fact that it operates only at the cache line level. The oracle dead sub-block predictor, which does not bring unused sub-blocks into the cache and in addition turns off active sub-blocks once they are dead, reduces cache energy by 51% on average. For most benchmarks, the energy savings correlate with the cache line usage. These results show that significant energy reduction can be achieved by operating at the sub-block level.

4.2 The Dead Sub-Block Predictor (DSBP)

This chapter proposes the Dead Sub-Block Predictor (DSBP) for detecting when a given cache line sub-block is dead (that is, it will not be accessed again). We use recent history information stored in a pattern history table to predict usage patterns. Traditional gated V_{DD} circuit techniques (POWELL et al., 2000) are used to power off sub-blocks once they are predicted to be dead to save energy.

Figure 4.5: DSBP: Mechanism architecture including cache metadata and PHT.

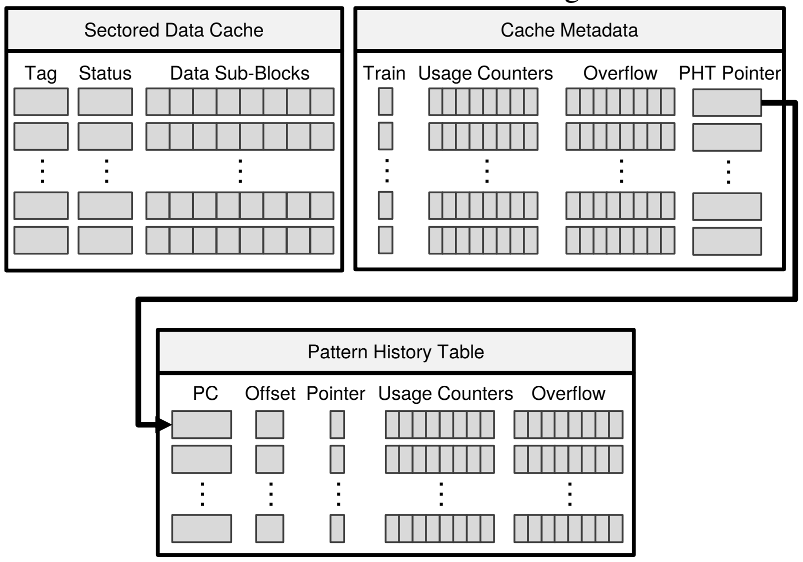


Figure 4.5 shows a traditional *sector cache architecture* (IBM, 1974), and the additional *cache metadata* and *PHT* required by our mechanism. The *sector cache* containing tag and data arrays in the left part of the figure shows how cache lines are divided into sub-blocks. The *cache metadata* contains information to guide our predictor. Each cache metadata line includes the following fields:

- A *train flag* to indicate if accesses on that specific cache line should update the pattern in the PHT.
- Sub-block *usage counters* to store the number of accesses the sub-block is predicted to receive before it becomes dead,
- A set of *overflow bits* to indicate if the predicted number of sub-block accesses exceeds the maximum value the *usage counters* can hold. If set, the sub-block will remain powered on until the line is evicted.
- A *PHT Pointer* linking a cache line to its respective entry in the PHT.

The PHT is used to store previous sub-block usage patterns. It is indexed by the PC of the load/store instruction that caused the cache miss and the requested cache line offset (byte within the line) of the miss address. The key observation behind using the PC along with the line offset as the index is that a given sequence of memory instructions often accesses the same fields of a record (see the example in Figure 4.1). Although different instances of a record may have different offsets within the cache line, the number of different offsets of an instance is bounded (CHEN et al., 2004; KUMAR; WILKERSON, 1998; PUJARA; AGGARWAL, 2008). Therefore, the PC+offset combination provides a high coverage of patterns even with moderately sized Pattern History Tables.

Each entry in the PHT includes: 1) a *Pointer flag* indicating that some cache line has a pointer to that specific PHT entry, 2) a set of *usage counters* and 3) *Overflow bits*. The usage counters and overflow bits are identical to those in the cache metadata and will be copied from the PHT to the metadata as our mechanism operates.

The main operations performed by the mechanism during cache accesses are:

Cache Line Miss: The PHT is searched for an entry matching the PC and offset of the instruction that caused the miss. For a PHT *hit*, the mechanism will copy the PHT's usage counter and overflow bits into the cache metadata and only the sub-blocks predicted to be used are fetched and stored into the cache line. The pattern in the PHT is kept intact. If the PHT entry indicates that no other pointer exists to that entry (pointer flag field zero), the new cache line is linked to the PHT pattern. In the case of a PHT *miss*, the train flag is set, and all usage counter and overflow bits are reset in the cache metadata. The PHT will reset all the usage counter and overflow bits and evict the LRU entry to make room for a new pattern. A PHT pointer is created to link the cache metadata and the new entry. Since the train flag is set, subsequent accesses to this line will update the usage counters in the PHT.

Cache Line Hit and Sub-Block Hit: If the train flag is disabled, the sub-block usage counter in the metadata is decremented and the sub-block is turned off if its usage counter and overflow bit are zero. The PHT will be updated only when the train flag is enabled. In case our mechanism predicts a dead sub-block in a dirty cache line, the sub-blocks remain powered on until all the sub-blocks inside that line are predicted to be dead. Afterwards, a write-back operation will be issue for this cache line, and all the sub-blocks will be turned off.

Cache Line Hit and Sub-Block Miss:[1] The requested sub-block will be brought into the cache line and its overflow bit will be set. If the cache metadata has a valid pointer to a PHT entry, then the mechanism will increment the corresponding usage counter in the PHT entry.

Cache Line Eviction: If the cache line contains a valid link to a PHT entry, the flag that indicates that a valid pointer exists must be disabled in the PHT. Also, if any of the usage counters in the metadata are non-zero (indicating that the corresponding sub-block was accessed fewer than the predicted number of times), the corresponding usage counters in the PHT entry are updated by decrementing each counter by the non-zero value. For this reason, we avoid using saturated counters.

Cache Line Invalidation: All the prediction information is kept intact. However, all the sub-blocks inside the cache line are turned off until the line receives valid data.

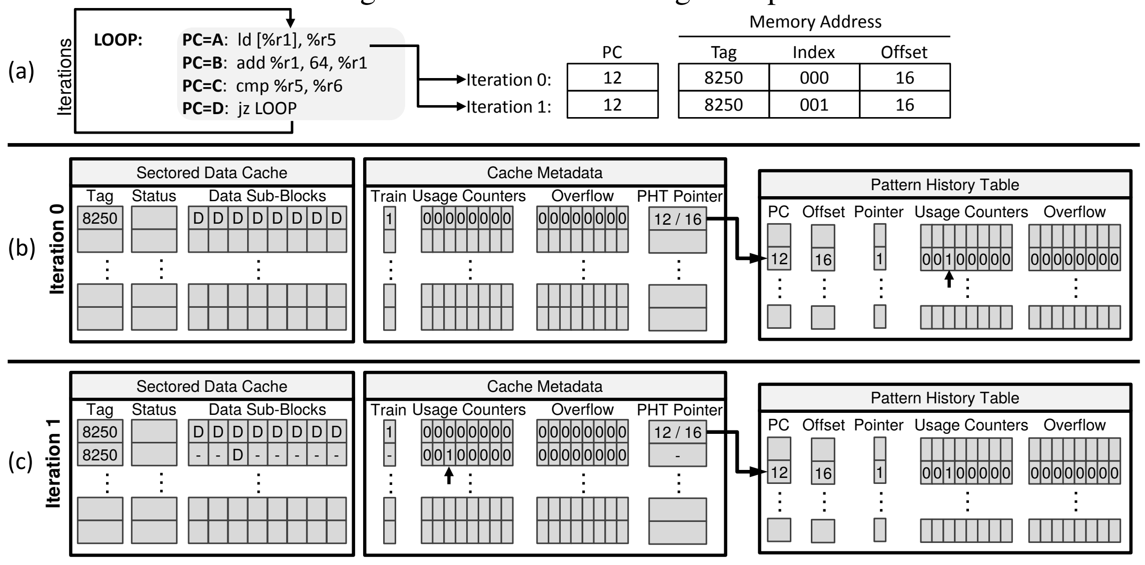
4.2.1 Usage Example

Figure 4.6 illustrates how DSBP learns and predicts access patterns based on previous usage. In the piece of code present in Figure 4.6(a), the program is iterating over a list of records of 64 bytes each, but is accessing only a single field of the record (similar to the example shown in Figure 4.1). Therefore, just the sub-block starting from the offset value 16 is being loaded into a register. For this example, the cache and PHT are initially empty.

Figure 4.6(b) presents the state of the cache, metadata, and PHT after the first iteration of the loop. Since there was no matching entry in the PHT, a new PHT entry is allocated, all the usage counters and overflow bits are reset, and since no other pointer exists to that PHT entry, a new pointer will be stored in the cache metadata and the pointer flag in the PHT entry is set. Because no previous pattern was available for the prediction (that is, PHT miss), all sub-blocks are brought into the cache line. The train flag is also

[1]The term "sub-block miss" applies to the situation where the cache line is present in the cache (that is, a matching tag is found), but the requested sub-block is not. This is in contrast to a "cache line miss" where no matching tag is found (that is, the entire line is not in the cache)

Figure 4.6: DSBP: Working example.



set in order to capture all the subsequent access to that line and learn the usage pattern. Assuming that a single access was made to the 3rd sub-block, the corresponding usage counter in the PHT entry is incremented to one.

Figure 4.6(c) shows a cache miss on the second iteration. This time, the predictor accesses the PHT and finds a matching entry. The usage bits indicate that only the 3rd sub-block will be used, and therefore only a single sub-block will be brought into the cache. Since the PHT pointer flag is already set, no new pointer will be generated. After the mechanism copies the usage counters and overflow bits to the metadata, the data can be used. Once the sub-block is used, the usage counter will be decremented to zero and the sub-block will be turned off. Subsequent loop iterations would operate in exactly the same way.

4.2.2 Improving the Cache Replacement Policy

We also use our mechanism to improve the traditional LRU cache replacement policy by prioritizing lines with all sub-blocks powered off for eviction. If a line has all sub-blocks powered off, that means our predictor has identified this line as dead (that is, it will not be accessed again). Evicting dead lines early, before they actually become a victim (being at the LRU position), can reduce the cache miss ratio by letting the not-dead lines stay longer in the cache. This also offsets the additional sub-block misses our predictor may cause when it underpredicts the usage pattern of a cache line.

4.2.3 Prefetching Adaptations

In order to obtain correct predictions with our mechanism, it is essential to have the instruction address (PC) that caused the cache miss, together with the offset requested on this cache miss. For the processor requests, this correlation information is easily obtained. However, in a presence of a prefetcher, this information may not be available for the cache misses caused when prefetching data. Different approaches can be planned to include the information required to the predictor inside the prefetcher structure.

The *stride prefetch* (BAER; CHEN, 1991) already stores the instruction address from the memory request which started a new prefetch pattern. We adapted this mechanism to also store the first memory address for each pattern. Thus, all the prefetched lines include the instruction address and an offset used to index the PHT inside our mechanism, in order to generate predictions. For the *stream prefetch* (JOUPPI, 1990), we included the instruction address and the offset which started a new stream. This enables the prefetch to send this extra information together to the prefetched lines in order to generate usage information for these lines.

4.2.4 Implementation on Multiple Cache Levels

Our mechanism operates at the sub-block level and therefore implementing our predictor on first level caches is straightforward since requests from the processor are also made at the sub-block level. However, next level caches receive requests from the previous level at a cache line granularity. Therefore, in order to implement our mechanism on systems with multi-level cache hierarchies, miss requests must be forwarded from one level to the next on a sub-block granularity. This also implies that our mechanism can be applied to a cache level only if applied to all previous levels. For example, in a 3-level cache hierarchy, we could apply our mechanism to just the L1, both the L1 and L2 but not the LLC, or all three levels.

4.3 Methodology

The baseline configuration for the processor is based on the Intel Sandy Bridge as shown in Table 3.6. The single-threaded applications from SPEC-CPU2006 and multi-threaded (8 threads) applications from SPEC-OMP2001 and NAS-NPB suites were used as workloads to evaluate DSBP. All the results are normalized to a baseline system without any predictor.

In order to improve the energy consumption at a finer granularity than current cache memories, we turn off sub-blocks from the cache line using gated V_{DD} circuit techniques as in (POWELL et al., 2000). Gated V_{DD} techniques use a transistor to gate the supply voltage (V_{DD}) of the cache SRAM cells. Previous work reports that the transition delay of turning on a gated-ground transistor shared by a 16 byte sub-block is only 0.20 ns (that is, one clock cycle in a 5 GHz microprocessor) (CHEN et al., 2004). Therefore, we assume our 8-byte sub-blocks can be powered on in a single cycle. Furthermore, this single-cycle latency can be hidden during a cache miss, since a sub-block can be powered on while the data being requested are fetched from the next level in the cache hierarchy.

In order to model the dynamic and static energy savings due to dead sub-block prediction, we model both the baseline cache architecture and our proposed mechanism with CACTI 6.5++ (MURALIMANOHAR; BALASUBRAMONIAN; JOUPPI, 2008) which is available inside McPAT version 1.0 (LI et al., 2009, 2013). To take into account a sector cache memory, we model a cache with 8 sectors (that is, sub-blocks) and the additional bits required to control the sub-blocks. Since our proposed mechanism requires extra metadata, the cache lines were also modeled with the extra bits necessary to support the usage counters, the overflow bits, the learn flag and the PHT pointer. After modeling the 8 sub-blocked cache with all the metadata, the CACTI power model was used to compute the dynamic energy and static when all sub-blocks are enabled. We also modeled the PHT table as a cache memory of the same size.

To compute the energy when just a part of the cache line is turned on, we modeled cache architectures with the same number of lines but fewer 8-byte sub-blocks (from 1 to 7 sub-blocks). The energy consumed by these smaller cache lines is used to model the energy consumption when just a few sub-blocks in the cache line are enabled.

Gated V_{DD} techniques require a 3% area overhead on the data array (POWELL et al., 2000). In order to model this overhead, two extra bytes (3.1% of line size) were added to the cache line in our mechanism's cache tag as the V_{DD} technique overhead.

4.4 Evaluation

The DSBP evaluated in this chapter uses 2-bit usage counters per sub-block and a 512-entry PHT organized as an 8-way set associative cache. To maintain the metadata information, 34 bits per cache line were added, which represents an overhead of 6.25% of the total cache size, assuming a tag size of 32 bits.

For the PHT, using only 16 bits to store the least significant part of the PC demonstrated to be enough to obtain accurate results. Moreover, since most of the accesses are aligned inside the cache line in sub-blocks of 8 bytes, only 3 bits are necessary to store the sub-block accessed inside the cache line (instead of using the full offset). The total size of the PHT used in our experiments is 2.75 KB per cache bank, which represents 8.60 % of the L1, 1.10% of the L2 and less than 0.30% of the total LLC size.

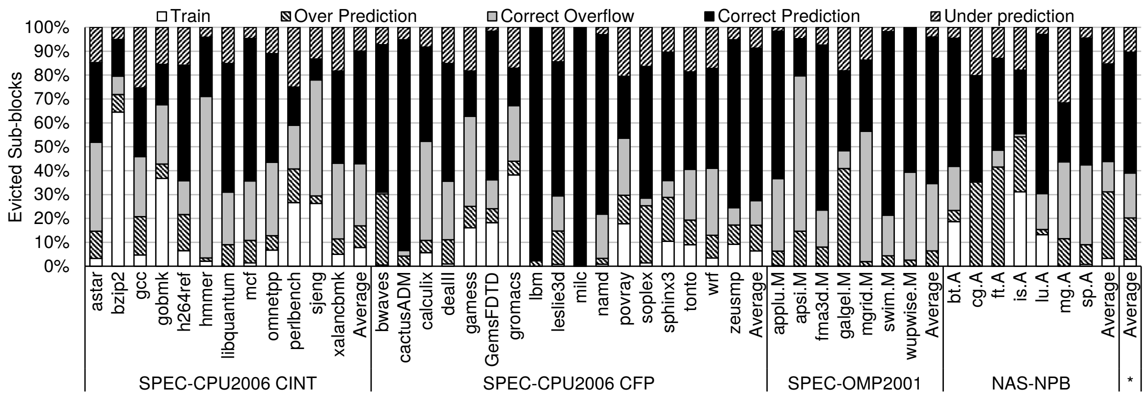
In order to evaluate our proposed mechanism, we compare it with the SKEWED mechanism, explained in Section 2.2.6. We implemented SKEWED using three tables with 8192 entries each, with a 2 bit counter per entry to perform the prediction. Each table is indexed using a different hash function, in order to obtain three different sub-predictions, which are combined to form the final prediction.

4.4.1 Mechanism Accuracy

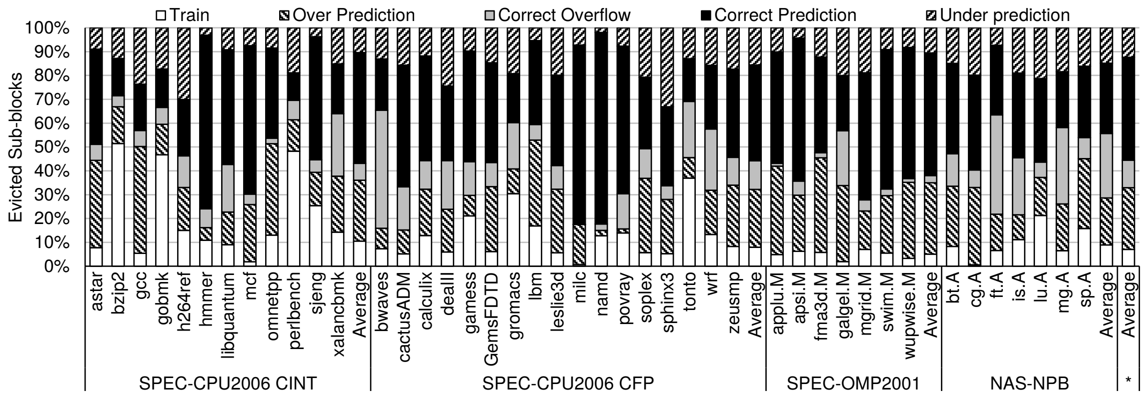
Figure 4.7 presents the accuracy results for our mechanism applied on L1, L2 and LLC. The figure is divided into correct prediction, correct overflow, over prediction, under prediction, and train. Each percentage corresponds to the fraction of evicted sub-blocks that fall into the corresponding category. A *correct prediction* occurs if a sub-block is evicted with its usage counter and overflow bit (in the metadata) equal to zero. This means that the sub-block was accessed exactly as many times as we predicted. *Correct overflow* occurs when a sub-block is evicted with its usage counter at zero but the overflow bit is set. *Over prediction* happens when a sub-block is evicted with its usage counter greater than zero. This implies that the sub-block was accessed fewer times than what we predicted. *Under prediction* occurs when a request is made to a sub-block that is powered off (that is, a sub-block miss). We keep track of such sub-blocks, and upon eviction, count them as an under predicted. *Train* means that no information about cache line usage was available in the PHT (that is, PHT miss). In this case, all 8 sub-blocks within the line are classified under this category. To generate complete statistics, we force an eviction of all the cache lines at the end of the execution.

These results show that our mechanism achieves a good coverage of 61% correct predictions on average. Note that the percentage of sub-blocks that are classified as correct overflow reduces for the LLC to the L1. This is because the L1 receives the most sub-block accesses and serves as a filter for the next level caches. Therefore, it is unlikely for sub-blocks in the LLC to receive enough accesses during training to have their overflow bits ever set. The opposite is true for over predictions. Recall that our mechanism corrects

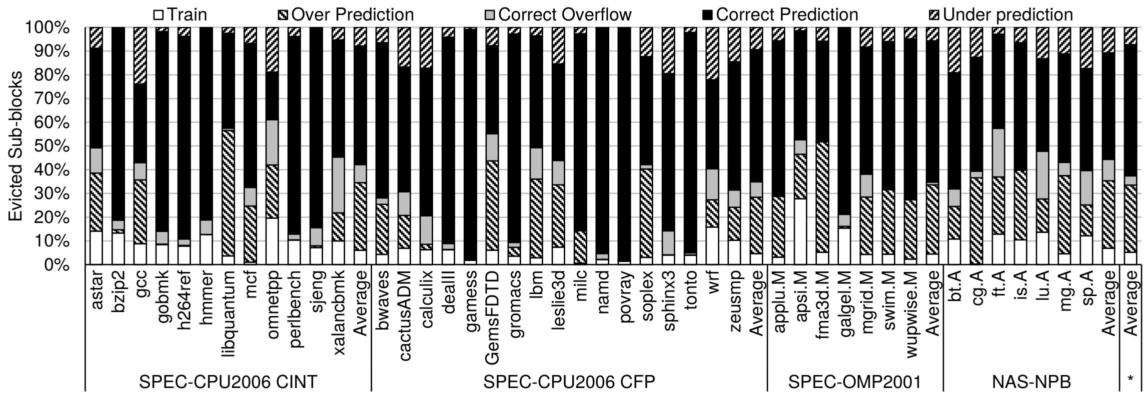
Figure 4.7: DSBP: Mechanism accuracy results.



(a) L1 cache.



(b) L2 cache.



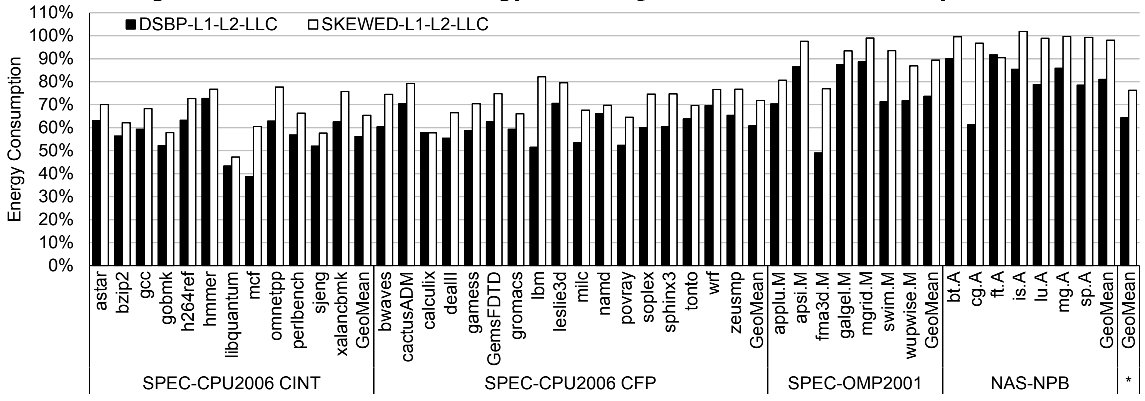
(c) LLC cache.

over prediction by updating the PHT only when the line containing the over predicted sub-block(s) is evicted. Since evictions are much less frequent for the L2 and LLC than the L1, it takes longer for the L2 and LLC to adapt to a new usage pattern, resulting in a greater percentage of over predictions. The under predictions account for 10% of the sub-blocks evicted on average, this avoids high performance degradation due to sub-block misses.

4.4.2 Energy Savings

Figure 4.8 presents total energy consumption for each predictor. The results are shown applying each mechanism in all the cache levels. DSBP achieves energy savings of 36% on average considering all the cache levels, it outperforms the previous proposal in terms

Figure 4.8: DSBP: Total energy consumption of the cache sub-system.



of energy savings. It improves by 12% even the best result obtained with SKEWED. DSBP has higher gains because 1) it powers off data stored in the cache at the sub-block level, and 2) it does so with relatively few updates to the PHT.

Table 4.1 presents the results for the DSBP and SKEWED mechanisms when the predictors are applied to each cache level in isolation (if applicable), and also when applied to multiple levels, also comparing with the oracle mechanisms applied on the L1 cache. Notice that the table shows the energy consumption considering the overall cache system.

It is possible to see that both mechanisms obtain the best results when applied in all the cache levels. Comparing DSBP L1 to the oracle dead sub-block predictor L1, we can observe that our mechanism could achieve 84% of the energy savings achievable with a perfect mechanism. Comparing SKEWED with the oracle dead line predictor L1, we see that this related work outperformed its oracle, this is because SKEWED improved the replacement policy and performs early write-back, what changes the operations performed inside the cache, leading to a different critical path of the application execution.

4.4.3 Performance Impact

As shown in the previous section, successfully predicting dead sub-blocks can significantly reduce cache energy consumption. However, incorrect predictions may introduce a negative impact on cache performance and actually increase energy consumption due to extra cache sub-block misses. Figure 4.9 shows the total number of extra cache misses normalized to the number of cache misses from the baseline cache architecture.

The result bars shown in Figure 4.9 present the percentage of extra sub-block misses caused by under predictions. Cache line misses are accesses where no matching tag entry was found in the cache (that is, the entire line is not present in the cache). Sub-block misses occur when the requested tag is present in the cache, but the requested sub-block is not available. Note that these sub-block misses only happened because the mechanism incorrectly identified a sub-block as dead and powered it off prematurely.

Figure 4.9 shows that DSBP increased on average the number of cache line misses due to sub-block misses by 10%, 19% and 12% for the L1, L2 and LLC respectively. However, a part of these extra cache misses are offset by the improved cache line replacement policy, preserving the level of performance that the baseline provides while significantly reducing cache energy.

Figure 4.10 shows a comparison of normalized execution time for the simulated mechanisms. The execution time correlates well with the number of misses in the last level

Table 4.1: DSBP: Total energy consumption of the cache sub-system.

		DSBP			SKEWED					ORACLE	
		L1	L1-L2	L1-L2-LLC	L1	L2	LLC	L1-L2	L1-L2-LLC	L1 Line Usage	L1 Sub-Block Usage
SPEC-CPU2006 - CINT	astar	76%	68%	63%	85%	93%	93%	78%	70%	86%	58%
	bzip2	80%	73%	56%	85%	94%	84%	79%	62%	86%	63%
	gcc	76%	68%	59%	84%	93%	90%	77%	68%	85%	65%
	gobmk	80%	72%	52%	85%	93%	80%	78%	58%	86%	66%
	h264ref	81%	76%	63%	90%	95%	88%	85%	73%	90%	71%
	hmmer	88%	83%	73%	92%	96%	89%	88%	77%	92%	79%
	libquantum	59%	50%	43%	76%	90%	76%	67%	47%	82%	56%
	mcf	61%	51%	39%	80%	91%	86%	71%	61%	81%	58%
	omnetpp	71%	63%	63%	87%	93%	99%	80%	78%	86%	58%
	perlbench	79%	73%	57%	88%	94%	85%	82%	66%	88%	57%
	sjeng	79%	71%	52%	85%	93%	80%	78%	58%	87%	67%
	xalancbmk	76%	71%	62%	89%	95%	88%	83%	76%	89%	66%
	GeoMean	**75%**	**68%**	**56%**	**85%**	**93%**	**86%**	**79%**	**65%**	**86%**	**63%**
SPEC-CPU2006 - CFP	bwaves	78%	72%	60%	89%	95%	86%	84%	75%	90%	72%
	cactusADM	82%	77%	70%	89%	95%	93%	84%	79%	90%	79%
	calculix	80%	72%	58%	84%	93%	82%	76%	58%	86%	74%
	dealII	77%	72%	55%	88%	95%	84%	82%	66%	89%	68%
	gamess	79%	74%	59%	90%	95%	85%	85%	70%	90%	65%
	GemsFDTD	72%	66%	63%	87%	95%	89%	82%	75%	89%	61%
	gromacs	82%	76%	59%	87%	94%	85%	81%	66%	88%	71%
	lbm	61%	56%	51%	88%	98%	96%	84%	82%	89%	60%
	leslie3d	79%	74%	71%	88%	96%	96%	83%	80%	90%	73%
	milc	72%	65%	53%	87%	94%	84%	81%	68%	89%	65%
	namd	86%	81%	66%	90%	95%	85%	84%	70%	91%	61%
	povray	77%	71%	52%	88%	94%	83%	82%	65%	88%	60%
	soplex	73%	66%	60%	85%	95%	87%	80%	75%	87%	53%
	sphinx3	75%	70%	61%	88%	95%	92%	83%	75%	89%	63%
	tonto	84%	79%	64%	89%	95%	85%	84%	70%	90%	76%
	wrf	81%	75%	70%	88%	95%	94%	83%	77%	89%	70%
	zeusmp	79%	73%	65%	86%	94%	92%	80%	77%	88%	69%
	GeoMean	**77%**	**71%**	**61%**	**88%**	**95%**	**88%**	**82%**	**72%**	**89%**	**67%**
SPEC OMP2001	applu.M	78%	76%	70%	92%	98%	90%	88%	81%	95%	65%
	apsi.M	88%	88%	86%	101%	99%	100%	100%	98%	100%	77%
	fma3d.M	58%	56%	49%	90%	98%	85%	88%	77%	93%	52%
	galgel.M	93%	92%	87%	99%	99%	95%	98%	93%	97%	85%
	mgrid.M	90%	90%	89%	99%	100%	101%	99%	99%	99%	76%
	swim.M	75%	75%	71%	94%	99%	98%	94%	93%	97%	66%
	wupwise.M	80%	78%	72%	95%	97%	88%	93%	87%	96%	73%
	GeoMean	**80%**	**78%**	**74%**	**96%**	**99%**	**94%**	**94%**	**89%**	**97%**	**70%**
NAS-NPB	bt.A	91%	91%	90%	100%	100%	99%	100%	100%	99%	79%
	cg.A	62%	63%	61%	98%	100%	95%	98%	97%	96%	43%
	ft.A	93%	92%	92%	92%	100%	99%	92%	90%	99%	80%
	is.A	81%	83%	85%	101%	100%	97%	101%	102%	99%	57%
	lu.A	79%	79%	79%	100%	99%	99%	99%	99%	98%	63%
	mg.A	87%	87%	86%	100%	100%	102%	100%	100%	99%	78%
	sp.A	79%	79%	79%	99%	100%	99%	99%	99%	99%	63%
	GeoMean	**81%**	**82%**	**81%**	**99%**	**100%**	**99%**	**98%**	**98%**	**98%**	**65%**
*	**GeoMean**	**78%**	**73%**	**64%**	**90%**	**96%**	**90%**	**86%**	**76%**	**91%**	**66%**

cache since these misses are the most costly in terms of latency. Sub-block extra misses introduced by the L1 and L2 predictors can be serviced by the last level cache with a minimal impact on execution time.

As mentioned before, DSBP has a negligible impact on system performance (equal to 2.25% on average). This is because the additional sub-block misses DSBP may cause are largely offset by the improved replacement policy DSBP offers. Our mechanism impacted the performance less than the evaluated related work.

Figure 4.9: DSBP: Normalized extra cache misses.

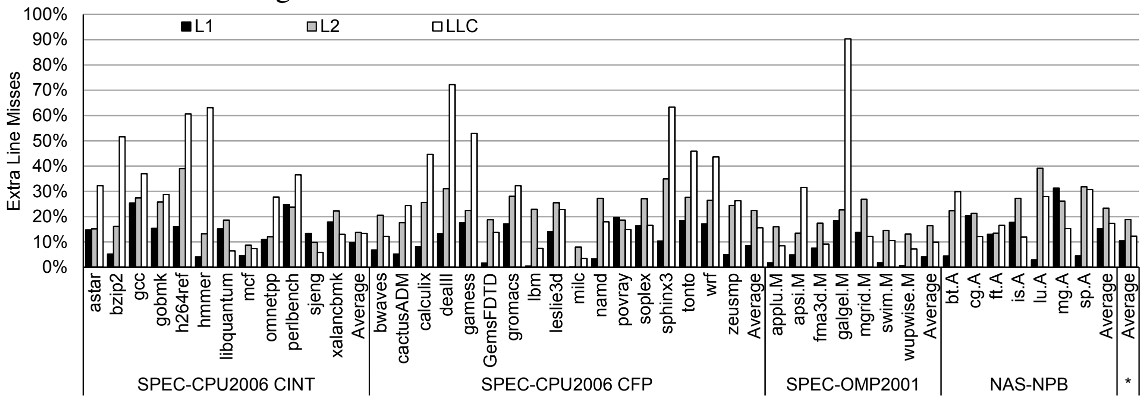
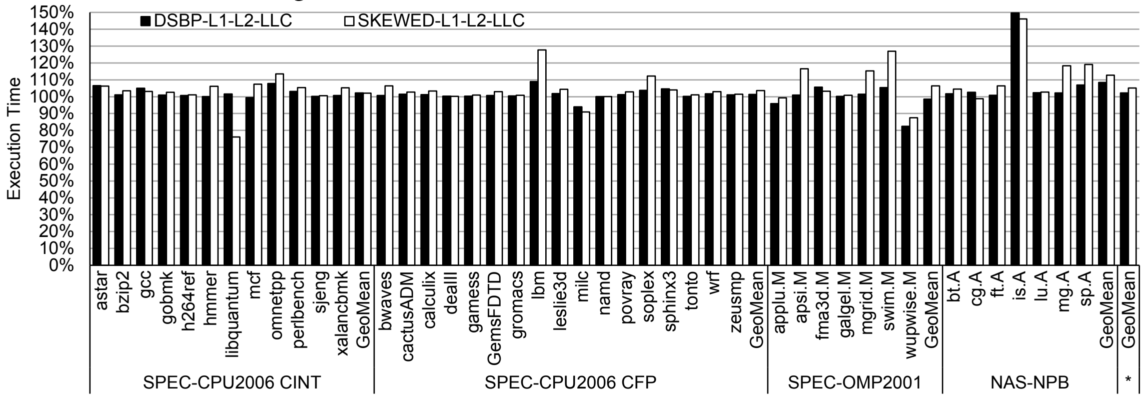


Figure 4.10: DSBP: Normalized execution time.



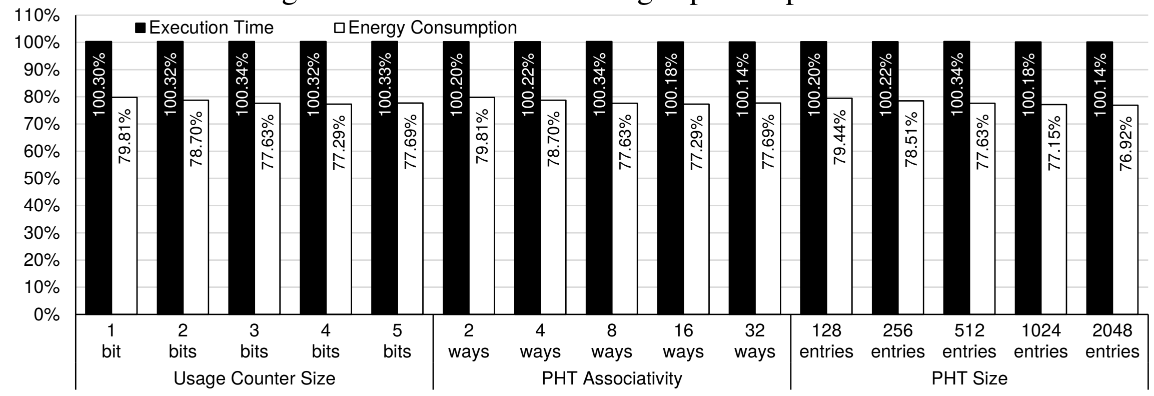
4.5 Design Space Exploration

This section presents a parameter sensitivity exploration for our DSBP mechanism. We study the effect of varying the usage counter size, the PHT associativity, and the PHT size of our mechanism. To reduce the number of experiments, we evaluate each parameter separately. First, the usage counter size is varied while fixing the PHT size and associativity at 512 entries and 8-way respectively (the average size and associativity we evaluate). Once the best usage counter size is identified, we fix its size, and vary PHT associativity while keeping the PHT size with 512 entries. Lastly, we vary PHT size, keeping the usage counter size and PHT associativity constant at their previously identified best performing values.

Figure 4.11 shows the results for the parameter sensitivity study. The results are presented in terms of energy consumption and execution time normalized to the baseline cache architecture executing all the benchmarks. For these experiments, DSBP was used only in the L1 cache memory, reducing thus the interference between the predictions in multiple cache levels.

First, we analyze the effect of varying the usage counter size. A larger counter size increases the potential to save energy by turning off sub-blocks. For example, with 4-bit counters, DSBP can predict that a sub-block will be touched exactly 15 times and then turn it off after it receives 15 accesses. However, with fewer bits per counter, the sub-block's overflow bit would be set and therefore the sub-block would never be turned off.

Figure 4.11: DSBP: PHT design space exploration.



On the other hand, increasing the usage counter size increases the energy overhead of the cache metadata. From the cache miss perspective, with smaller counters, the overflow bit is more likely to be set. As such, DSBP's replacement policy will not be able to identify lines as dead and therefore will not be as effective in reducing line misses compared to larger counter sizes. For the same reason, the number of sub-block misses increases as usage counter size increases. Overall, the total number of misses does not vary much. The results show that using 2 bits per usage counter balances these trade-offs well and achieves the best results in terms of energy reduction and performance.

Next, we evaluate the associativity of the PHT. A large number of ways reduces the conflicts inside the table. However, it also increases PHT energy consumption. Our results show the best results are obtained using an 8-way set-associative PHT.

Finally, we analyze the effect of varying the number of entries in the PHT. Increasing the number of entries in the PHT allows it to store more usage patterns and therefore fewer patterns will have to be re-learned. This again provides more opportunities to save energy by turning off dead sub-blocks. However, a larger PHT also consumes more energy each time it is accessed. We find a PHT size of 512 entries is best in terms of energy reduction and performance.

In summary, the sensitivity studies shows that DSBP does not require large PHTs or a large number of bits in the usage counter to provide benefits. Moderately sized Pattern History Tables provide good results since PC and offset are used to index the table as explained in Section 4.2. Furthermore, as shown in Section 4.1, less than 10% of the sub-blocks receive more than 3 accesses before their eviction, which justifies our result that 2-bit usage counters work best in terms of energy savings and performance.

4.6 Summary

To our knowledge, DSBP is the first proposal to exploit dead cache line prediction at the sub-block granularity. DSBP is used to reduce energy consumption in the cache subsystem by loading into the cache only those sub-blocks predicted to be useful, and turning off active sub-blocks as they are predicted dead. In addition, the LRU replacement policy is improved by prioritizing dead lines for eviction. This modification reduces the number of dead lines that remain inside the cache which leads to better utilization of cache space.

Our evaluations found that DSBP reduces energy consumption of the cache hierarchy by 36% on average compared to the baseline (without any dead line predictor in any cache level), and by 12% compared to a technique that turns off dead lines as predicted by a state-of-the-art predictors similar to those present in the LTP (LAI; FALSAFI, 2000; LAI; FIDE; FALSAFI, 2001), SDP (KHAN et al., 2010) and LWP (WANG; KHAN; JIMÉNEZ, 2012a).

5 DEAD LINE AND EARLY WRITE-BACK PREDICTOR (DEWP)

Several prediction mechanisms to keep only useful information in the cache have been proposed (ABELLA et al., 2005; WANG; KHAN; JIMÉNEZ, 2012a). However, previous approaches do not take into account that modified or dirty cache lines remain turned on for long periods of time, wasting energy while they could be evicted earlier. The gains can be increased even for dirty cache lines, by performing an early write-back to the memory before turning off the line. In this way, energy consumption, as well as pressure on the memory controller, can be reduced. Lee et al. (LEE; TYSON; FARRENS, 2000) proposed to early write-back dirty lines at the LRU position, but their proposal loses opportunities for energy savings by not evicting the line when the last write operation occurs. Using a perfect mechanism, we show that turning off invalid lines and dead lines can save 65% of energy from the LLC on average (see Section 5.1).

This chapter presents the Dead Line and Early Write-Back Predictor (DEWP) mechanism (ALVES et al., 2013), which consists of a last read/write predictor operating at the cache line granularity. The *last read* prediction aims to save energy by turning off dead or invalid cache lines. The *last write* prediction performs early write-backs of dirty cache lines to the lower memory level, since these lines will not be modified anymore. Both last read and last write predictions detect whenever a line receives its last access, prioritizing those lines for early eviction.

The last read predictor uses the access history to predict when a cache line becomes dead and can be turned off. The line is considered *dead* whenever the cache line receives its last read before it gets evicted or invalidated.

The last write predictor allows dirty cache lines to be early written back when it detects the last write operation, reducing the pressure on the memory controller between reads and writes during bursts of requests. Furthermore, performing the early write-back of dirty lines also enables those lines to be turned off whenever a last read is predicted.

Both predictors reduce cache pollution, prioritizing the eviction of dead lines. All the cache lines that would normally be evicted from the cache memory by the replacement policy are considered dead since their last access. By early evicting these lines, other cache lines that are still alive can stay longer inside the cache.

The main contributions of this Dead Line and Early Write-Back Predictor (DEWP) mechanism are:

Last read predictor: We turn-off cache lines after they receive the last read before the line gets evicted. This translates into 25% of cache energy savings.

Last write predictor: Our mechanism can write-back the dirty cache lines after they receive the last write (before they are evicted). Therefore, we increase the time window to write-back the cache line to memory and reduce the pressure in the memory controller.

Last access predictor: Combining both prediction results, our mechanism detects the last access to a cache line, prioritizing it for early eviction, thus, improving the cache utilization. The predictor achieves 73% of correct predictions with 14% of false positives on average for all the cache levels.

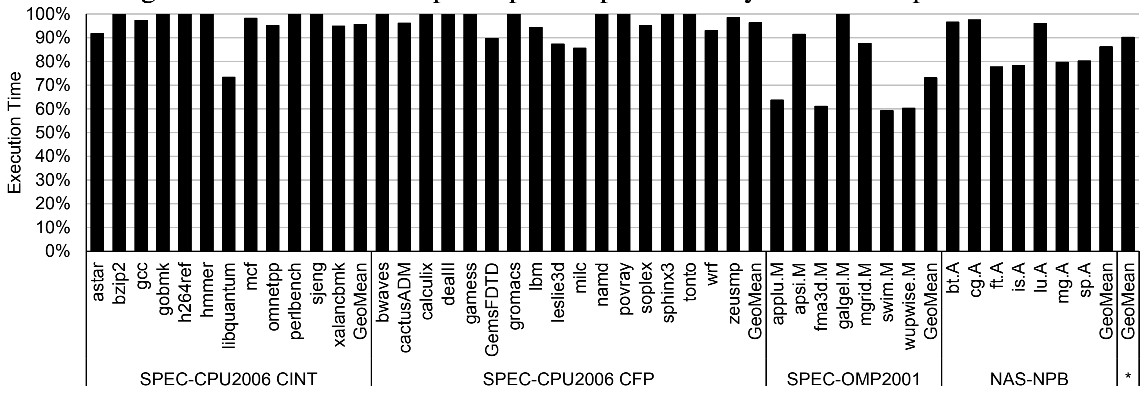
5.1 Motivation

This section presents motivation results in order to evaluate the possible gains in terms of performance and energy consumption when using DEWP. Our mechanism aims to early write-back lines after its last update in order to reduce the memory pressure, also reduce the cache pollution, increasing the virtual cache size. The final goal is to reduce the static energy consumption by turning off the cache lines after their last access.

5.1.1 Sensitivity to Early Write-Back

Dirty cache lines remain in the cache until they are evicted by another line request. However, they can be sent to write-back earlier when the last write operation is detected (WANG; KHAN; JIMÉNEZ, 2012a). When predicting the last write operation, a dirty cache line is available for write-back earlier. Thus, the time window to write it back to memory becomes longer, creating an opportunity to reduce pressure to the memory controller. Additionally, by using a last write predictor, dirty lines can be evicted earlier, thereby increasing the potential cache capacity.

Figure 5.1: Potential of speedup for a perfect early write-back predictor.



In order to show the potential benefit of a perfect last write predictor, Figure 5.1 shows the performance improvement of a system with instant LLC write-back to memory normalized to a conventional CMP with write-back. This shows the potential of performing write-back operations without interfering with read requests in the memory controller. Since most data accesses tend to occur in bursts (WANG; KHAN; JIMÉNEZ, 2012a), reducing memory pressure during those bursts is a key for memory performance.

Figure 5.1 shows an average 4% performance improvement for single threaded benchmarks (SPEC-CPU2006) and 21% for multi-threaded benchmarks (SPEC-OMP2001 and NAS-NPB). As expected, multi-threaded applications present higher performance gains due to their higher memory pressure.

Figure 5.2: Memory requests and write-back operations over time for *libquantum*.

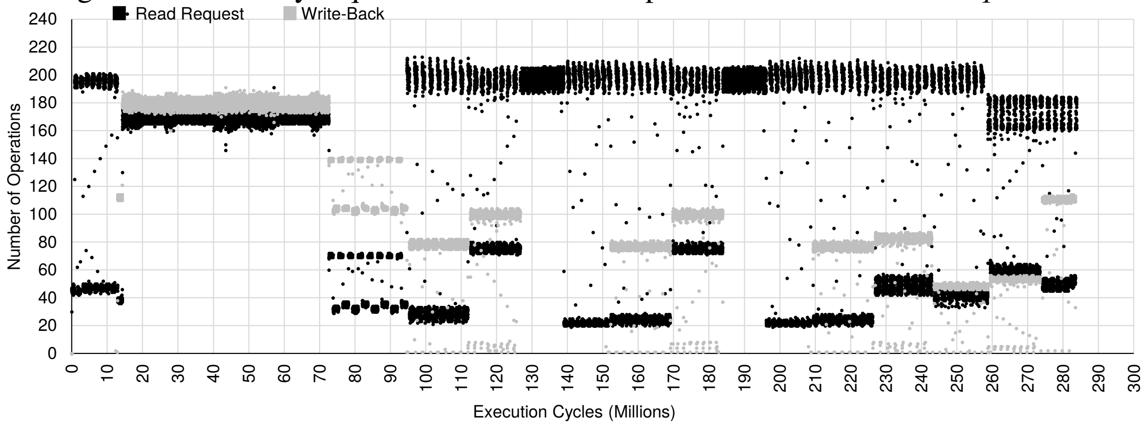
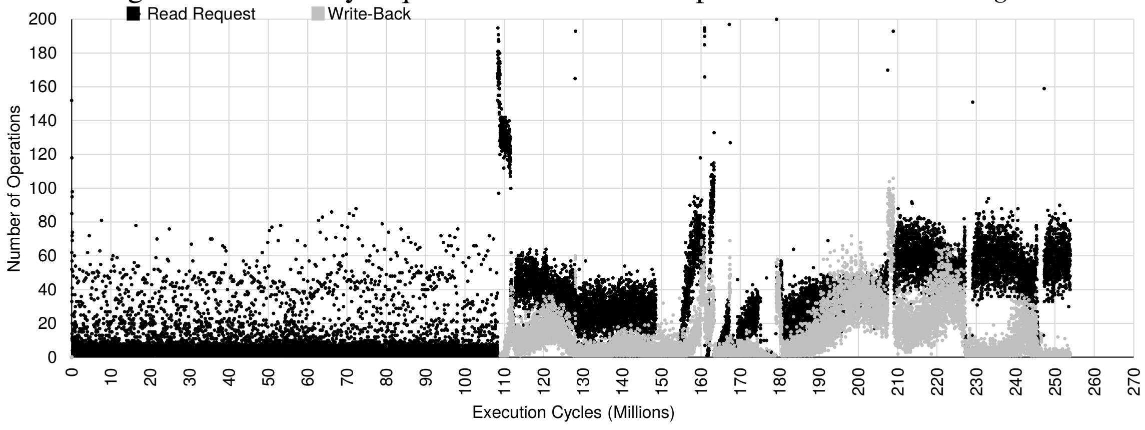


Figure 5.3: Memory requests and write-back operations over time for *gcc*.



Figures 5.2 and 5.3 show the concentration of write-back operations on the main memory controller during the application execution for two benchmarks *libquantum* and *gcc* from SPEC-CPU2006. Each dot shows the number of memory operations on the memory controller (read requests and write-backs) for 10,000 cycles.

For instance, we can observe bursts of read requests from cycle 90 M to cycle 110 M for *libquantum*, where the number of memory operations spikes multiple times from less than 50 operations to 200 operations per 10,000 cycles. In the same period, the write-backs also increase from almost 0 to 80 operations per 10,000 cycles.

We can observe that *libquantum* achieves more bursts of operations compared to *gcc*. This difference reflects the potential of each application to benefit from the early write-back, *libquantum* achieved 27% of performance improvement while *gcc* achieved only 3%.

5.1.2 Potential for Energy Savings in LLC

The LLC static energy usage can account for more than 90% of the total energy consumption of the LLC (LI et al., 2013) (see also Chapter 1). Figure 5.4 shows the maximum theoretical energy savings in the LLC, considering that cache lines could be turned off after their last access or whenever the cache line becomes invalid. For this experiment, we consider a perfect oracle mechanism without any overhead in terms of energy consumption.

Figure 5.4: Potential for LLC energy savings for an oracle line usage predictor.

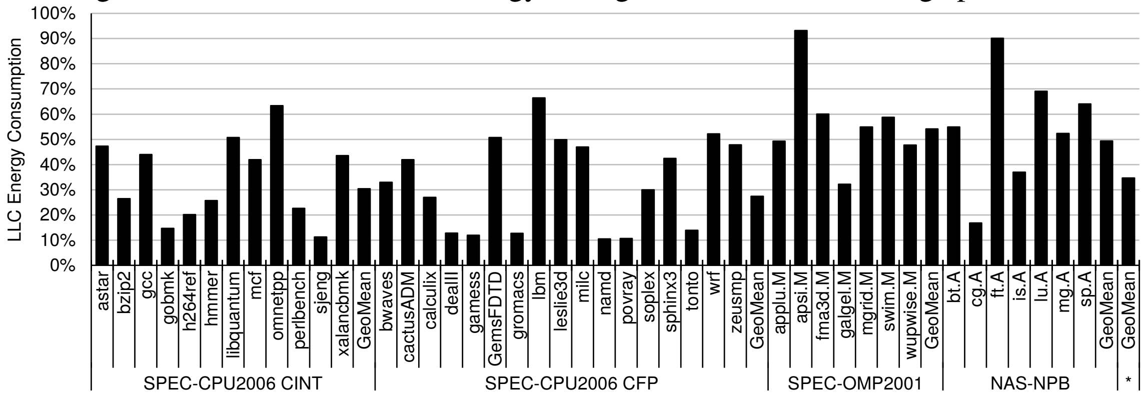


Figure 5.4 shows an average of 65% LLC energy reduction for all benchmarks evaluated. The results show that benchmarks with higher energy savings are those which have the least amount of accesses per cache line on average (see Section 4.1). These benchmarks have a low data reuse ratio and therefore offer higher opportunities for energy savings. For instance, the *sphinx3* benchmark has a high cache line reuse ratio, with more than 50% of the cache lines accessed more than 16 times before the line gets evicted. Therefore, this benchmark shows small energy savings with a perfect mechanism.

On average, more than 90% of the LLC lines of the evaluated benchmarks receive only one access before the line gets evicted (ALVES et al., 2012), which shows the high potential for energy savings for all the benchmarks, since a large part of the LLC lines are dead after the first access (dead on arrival).

5.1.3 Overall Potential Benefits

This section has shown the potential benefits of a perfect early write-back predictor in terms of performance and energy consumption. We showed average performance improvements of 10% and energy savings of 65% for all benchmark suites. All the experiments above show the potential benefits of the mechanism proposed in this chapter.

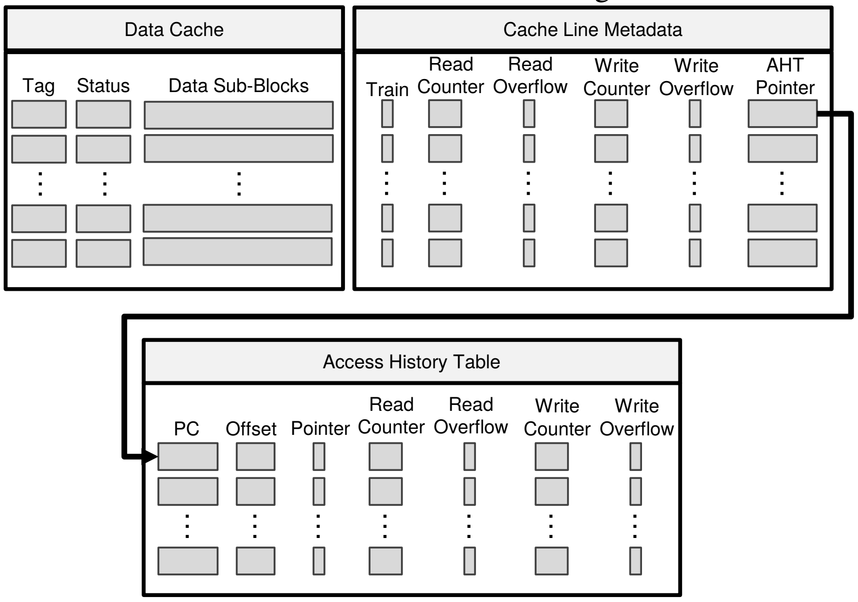
5.2 The Dead Line and Early Write-Back Predictor (DEWP)

Dead Line and Early Write-Back Predictor (DEWP) is a predictor to detect last read and write accesses to cache lines. DEWP uses recent access information stored in an Access History Table (AHT) to predict usage patterns. The combination of traditional gated V_{DD} circuit techniques (POWELL et al., 2000) and DEWP allows to power off cache lines once they are predicted dead, therefore saving static energy.

5.2.1 Overview of the Mechanism

Figure 5.5 shows for a set of cache lines the structures required to build DEWP. These structures are: 1) *Cache line metadata* which adds information for every cache line. 2) An *Access History Table (AHT)* that stores the prediction information. The cache line metadata guides the cache line predictions. Each metadata line consists of the following fields:

Figure 5.5: DEWP: Mechanism architecture including cache metadata and AHT.



- A *Train flag* to indicate if accesses to the cache line should update the pattern in the AHT.
- A *Read Counter* to store the number of read accesses the cache line is predicted to receive before it becomes dead.
- A *Read Overflow* bit to indicate if the predicted number of read accesses exceeds the maximum value the *Read Usage counter* can hold. If set, the cache line remains powered on until the line is evicted.
- A *Write Counter* to store the number of write accesses the cache line is predicted to receive before it gets evicted.
- A *Write Overflow* bit to indicate if the predicted number of write accesses exceeds the maximum value the *Write Counter* can hold. If set, the cache line will not be sent to early write-back.
- An *AHT Pointer* linking a cache line to its respective entry in the AHT.

The AHT is indexed by the program counter (PC) of the memory instruction that caused the cache miss and the requested cache line offset (byte within the line) of the address. The PC+offset combination has been shown to provide high accuracy and high coverage of patterns even with moderately sized AHTs (CHEN et al., 2004; KUMAR; WILKERSON, 1998; PUJARA; AGGARWAL, 2008). Each entry in the AHT consists of a *Pointer flag*, which indicates that a cache line has a pointer to that specific AHT entry, as well the read and write counters and their respective overflow bits, which have the same semantics as in the cache line metadata.

5.2.2 Mechanism Operations

The main operations performed by DEWP are triggered by the following cache operations:

Cache Line Miss: We search for an entry matching the PC and offset of the instruction that caused the miss. On an *AHT hit*, the mechanism copies the AHT's read/write counters and overflow bits into the cache metadata and resets the train flag. In the case of an *AHT miss*, a new entry is created. The train flag is set, and all usage counters and overflow bits are reset in the cache metadata. The Access History Table (AHT) resets the read/write counters and overflow bits and evicts the LRU entry to make room for the new pattern (AHT line). An AHT pointer is created linking the cache metadata and the new entry. Because the train flag is set, future accesses to this line update the counters only in the AHT. In order to avoid multiple lines updating the same AHT entry, the Pointer flag is used to inform if another cache line is already linked to that entry. In this case, the new link is not created.

Tag Hit and Data Cache Line On: If the train flag is disabled, the read counter in the metadata is decremented and the cache line is turned off if its read counter and overflow bit are zero. The AHT will only be updated when the train flag is enabled. The AHT to be updated is determined by the pointer in the metadata.

Tag Hit and Data Cache Line Off: The requested cache line is brought into the cache and its read overflow bit is set. If the cache metadata has a valid pointer to an AHT entry, the train flag is enabled and the mechanism increments the corresponding usage counter in the AHT entry.

Cache Line Eviction: If the read/write counters in the metadata is non-zero (indicating that the cache line was accessed less than the predicted number of times), the usage counter in the AHT entry is updated by decrementing the counter by the non-zero value. Also, if the cache line contains a valid link to an AHT pointer, the pointer flag must be disabled in the corresponding AHT.

Cache Line Invalidation: If the write counter is zero, it means that the last write was mispredicted. In this case, similar to the tag hit and data turned off case, the write overflow bit is set. If the cache metadata has a valid pointer to an AHT entry, the train flag is enabled, so future writes increment the write counter. Moreover, the cache line is turned off until it receives valid data.

Cache Line write-back: Similar to what happens during a tag hit and data turned on case, the write counter is decremented and the cache line is sent to early eviction in the case its write counter and overflow bit are zero. The AHT is only updated when the train flag is enabled. The AHT to be updated is determined by the pointer in the metadata. If the cache line was turned off, it is turned on again.

The proposed mechanism does not modify the coherence protocol at all. The protocol states are kept untouched even when the cache line is turned off. The tag store is always kept turned on.

Although DEWP is proposed to be used in the LLC, it can also be used at any cache level. In order to predict prefetched cache lines, DEWP requires small modifications into the prefetch structures to store the instruction address and the memory address which triggered the prefetcher as explained in Section 4.2.3.

5.2.3 Improving the Cache Replacement Policy

We also use our mechanism to improve the traditional LRU cache replacement policy by prioritizing lines that are turned off. Evicting dead lines early before they are at the LRU position can reduce the cache miss ratio by letting the alive lines stay longer in the cache (ALVES et al., 2012). Although we use DEWP with the LRU replacement policy, other policies could also be easily modified to take advantage of our mechanism.

5.3 Methodology

The baseline configuration for the processor is based on the Intel Sandy Bridge as shown in Table 3.6. The single-threaded applications from SPEC-CPU2006 and multi-threaded (8 threads) applications from SPEC-OMP2001 and NAS-NPB suites were used as workloads to evaluate DEWP. All the results are normalized to a baseline system without any predictor.

In order to increase the static energy efficiency of the cache memories, we turn off the data array part of the cache line using gated V_{DD} circuit techniques, as in (POWELL et al., 2000). Gated V_{DD} techniques use a transistor to gate the supply voltage (V_{DD}) of the cache SRAM cells.

Gated V_{DD} techniques require an extra latency (1 cycle) to turn on and off the cache line. The latency to turn off the cache line does not impact the performance, it only reduces the energy savings slightly. When turning on the cache line, it is important to note that such latency would appear only during read misses and when receiving a cache line write-back. However, during read misses, the high latency imposed by the main memory hides all the latency to turn the cache line on. On the other hand, cache line write-backs do not represent a critical path for the program execution and only cause a stall when all the write-back buffers are full, which occurs for less than 0.01% of cache accesses for all benchmarks that we evaluated.

In order to model the static energy savings using the DEWP predictor, we model both the baseline cache architecture and our proposed mechanism with CACTI 6.5++ (MURALIMANOHAR; BALASUBRAMONIAN; JOUPPI, 2008) which is available inside McPAT version 1.0 (LI et al., 2009, 2013). We model tag and data power consumption. We also modeled the AHT table as a cache memory of the same size.

Since our proposed mechanism requires extra metadata, the cache lines were also modeled with the extra bits necessary. We also consider that the metadata and the tag array are always turned on, because they are used by DEWP during the *Cache Line Eviction* operation to fix possible mispredictions. The additional energy consumption of the AHTs is also modeled and added together with the overall energy consumption.

5.4 Evaluation

For our mechanism, experimental evaluation showed that using 2 bits in the read and write counter covers more than 98% of the LLC lines. This means that those lines receive less than 4 accesses before their eviction. In our experiments, we used 512 entries per AHT which proved to be enough to generate accurate results. To maintain the metadata information, 16 bits per cache line were added, which represents an overhead of 3.00% of the total cache size, assuming a tag size of 32 bits.

For the AHT, using only 16 bits to store the least significant part of the PC demonstrated to be enough to obtain accurate results. Moreover, since most of the accesses are aligned inside the cache line in sub-blocks of 8 bytes, only 3 bits are necessary to maintain the sub-block accessed inside the cache line (instead of using the full offset). The total size of the AHT used in our experiments is 1.6 KB per cache bank, which represents 5.10% of the L1, 0.65% of the L2 and less than 0.15% of the total LLC size. Each AHT is organized as an 8 way set-associative cache in order to reduce the conflicts and increase the accuracy of the predictions.

In order to evaluate our proposed mechanism, we compare it with the SKEWED mechanism, explained in Section 2.2.6. We implemented SKEWED using three tables with 8192 entries each, with a 2 bit counter per entry to perform the prediction. Each table is indexed using a different hash function, in order to obtain three different sub-predictions, which are combined to form the final prediction.

5.4.1 Mechanism Accuracy

To analyze the accuracy of our mechanism, every time a line is evicted from the cache, we classify the line as: correct prediction, correct overflow, over prediction, under prediction, and train. To generate complete statistics, we force an eviction of all the cache lines at the end of the execution.

A *correct prediction* occurs if a line is evicted with its usage counters (read and write) and overflow bits (in the metadata) equal to zero. This means that the line was correctly turned off/written back. *Correct overflow* occurs when a line is evicted with its usage counters at zero but at least one of the overflow bits is set. *Over prediction* happens when a line is evicted with at least one usage counter greater than zero. This implies that the line was accessed fewer times than what we predicted. *Under prediction* occurs when a request is made to a line that is powered off or when a write arrives to a line already early written back. *Train* means that no information about cache line usage was available in the AHT (that is, AHT miss). In this case, line is classified under this category.

Notice that under predictions can hurt the performance, by early evicting or turning off alive lines and thus generating extra cache misses for those lines that have a clean copy of the data, and also generate extra write-backs for dirty lines.

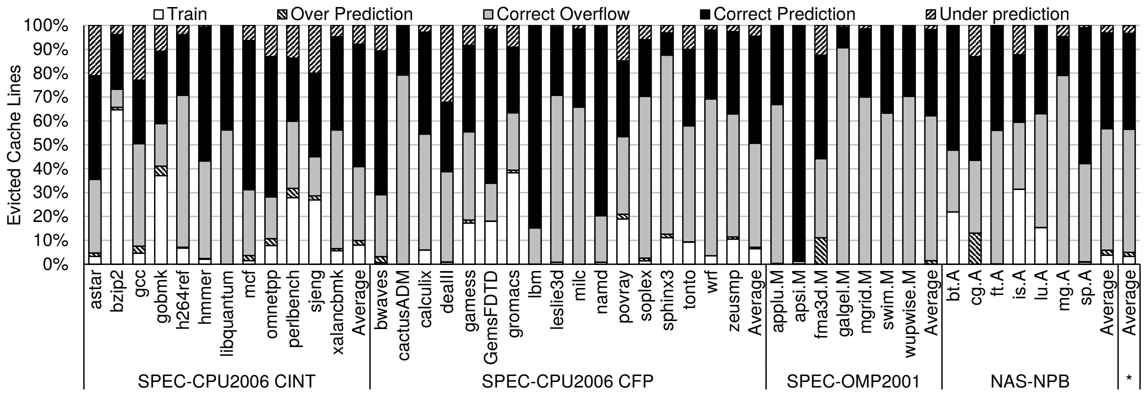
Figure 5.6 presents the accuracy results for our mechanism applied on L1, L2 and LLC, each percentage corresponds to the fraction of evicted cache lines that fall into the corresponding category. It shows that DEWP requires an average of 9% of cache line invalidations to train the mechanism. For 73% of the invalidations, DEWP correctly predicted the line usage. DEWP overpredicts in 4% of the invalidations, and underpredicts in 14%.

5.4.2 Energy Savings

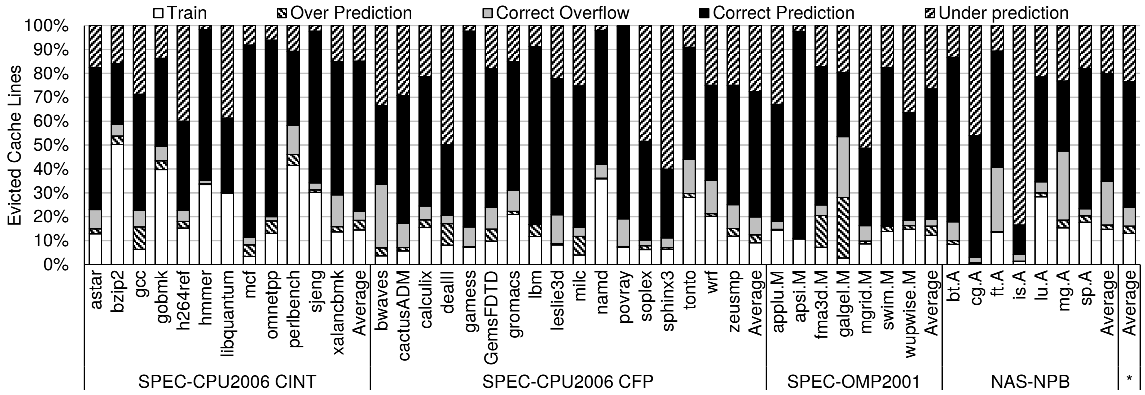
The cache energy efficiency is increased by using our mechanism to turn off dead and invalid lines. The results in Figure 5.7 are shown in terms of overall energy savings for the cache memories, normalized to the baseline. DEWP achieves on average a 25% energy savings compared to the baseline. Our mechanism outperforms the related work by 4% on average. As expected (see Figure 5.4) higher gains were obtained for the single-threaded applications.

Table 5.1 presents the results for the DEWP and SKEWED mechanisms when the predictors are applied to each cache level in isolation, and also when applied to multiple levels, also comparing with the oracle line usage mechanisms applied on the L1 and LLC cache. Notice that the table shows the energy consumption considering the overall cache system. DEWP achieved energy savings close to the oracle mechanism for L1 and LLC, while achieving better results than SKEWED.

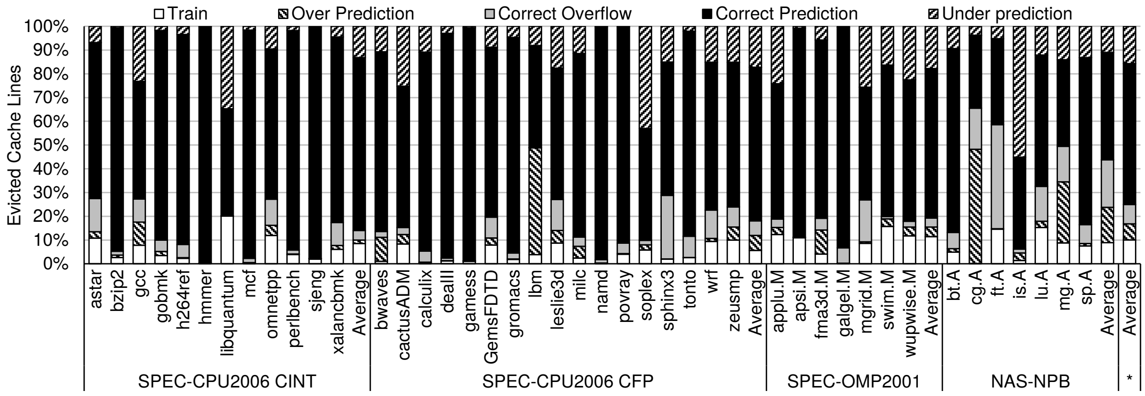
Figure 5.6: DEWP: Mechanism accuracy results.



(a) L1 cache.



(b) L2 cache.



(c) LLC cache.

5.4.3 Performance Impact

Our mechanism can influence the execution time of an application in different ways. DEWP can increase the performance with early evictions of dead lines, enabling more effective space in the cache memory, while early write-backs of last written lines can potentially reduce the memory controller contention. On the other hand, our mechanism can hurt the performance by causing extra cache misses because of underpredictions.

As shown in the previous section, successfully predicting dead cache lines can significantly reduce cache energy consumption. However, incorrect predictions may introduce a negative impact on cache performance and actually increase energy consumption due

Figure 5.7: DEWP: Total energy consumption of the cache sub-system.

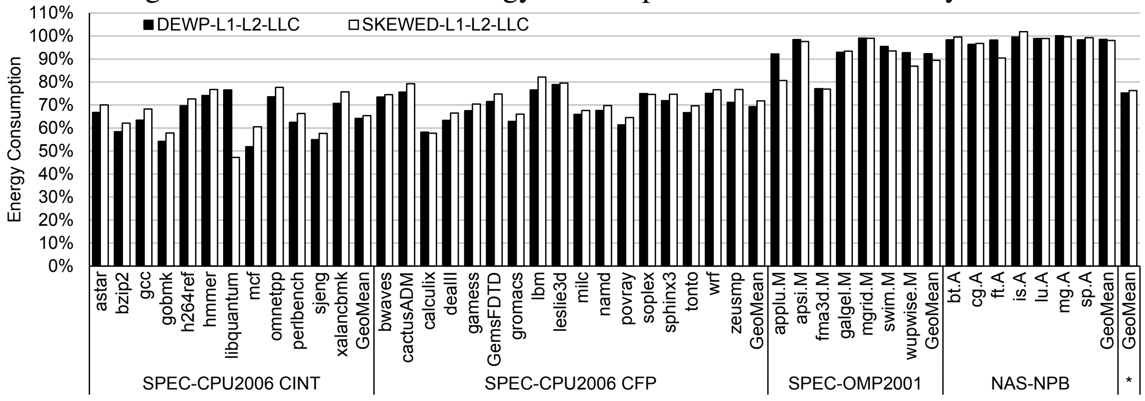
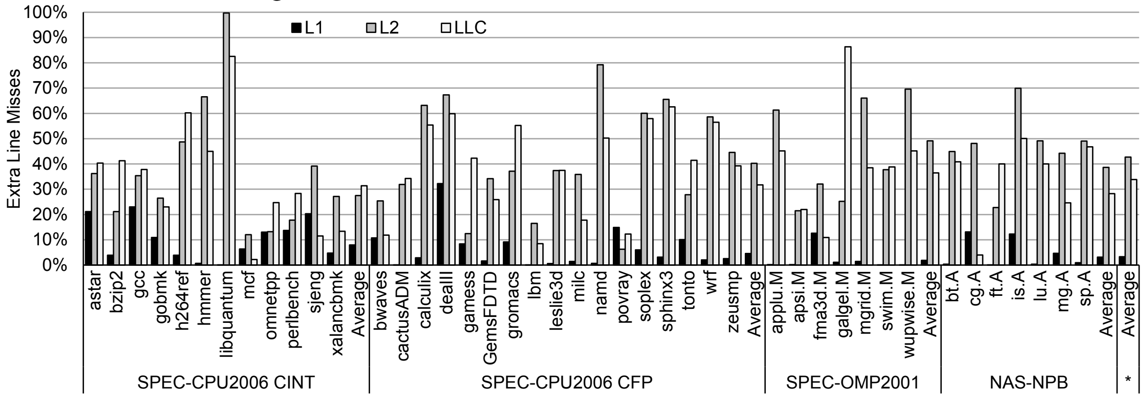


Figure 5.8: DEWP: Normalized extra cache misses.



to extra cache line misses. Figure 5.8 shows the total number of extra cache misses normalized to the number of cache misses from the baseline cache architecture.

Figure 5.9: DEWP: Normalized execution time.

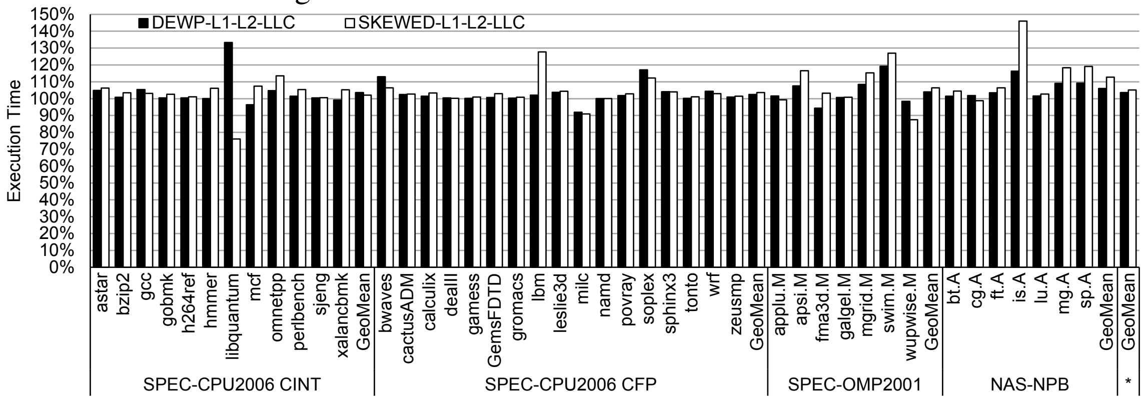


Figure 5.9 shows the execution time of our mechanism normalized to the baseline. We can observe that in most of the cases, the performance gains correlate with the sensitivity study presented in Section 5.1 and the prediction accuracy results. However, some benchmarks, such as *libquatum* and *is.A*, had a performance degradation, because our predictor failed to recognize some cache access patterns.

Table 5.1: DEWP: Total energy consumption of the cache sub-system.

		DEWP					SKEWED					ORACLE	
		L1	L2	LLC	L1-L2	L1-L2-LLC	L1	L2	LLC	L1-L2	L1-L2-LLC	L1 Line Usage	LLC Line Usage
SPEC-CPU2006 - CINT	astar	83%	92%	92%	75%	67%	85%	93%	93%	78%	70%	86%	87%
	bzip2	83%	93%	83%	76%	58%	85%	94%	84%	79%	62%	86%	82%
	gcc	82%	92%	88%	74%	63%	84%	93%	90%	77%	68%	85%	85%
	gobmk	82%	92%	80%	75%	54%	85%	93%	80%	78%	58%	86%	78%
	h264ref	88%	95%	87%	83%	70%	90%	95%	88%	85%	73%	90%	86%
	hmmer	89%	95%	89%	85%	74%	92%	96%	89%	88%	77%	92%	89%
	libquantum	78%	93%	93%	70%	77%	76%	90%	76%	67%	47%	82%	85%
	mcf	76%	90%	84%	65%	52%	80%	91%	86%	71%	61%	81%	81%
	omnetpp	83%	92%	99%	76%	74%	87%	93%	99%	80%	78%	86%	91%
	perlbench	85%	94%	84%	79%	62%	88%	94%	85%	82%	66%	88%	83%
	sjeng	83%	92%	79%	75%	55%	85%	93%	80%	78%	58%	87%	78%
	xalancbmk	86%	94%	89%	80%	71%	89%	95%	88%	83%	76%	89%	89%
	GeoMean	**83%**	**93%**	**87%**	**76%**	**64%**	**85%**	**93%**	**86%**	**79%**	**65%**	**86%**	**84%**
SPEC-CPU2006 - CFP	bwaves	90%	94%	86%	86%	73%	89%	95%	86%	84%	75%	90%	88%
	cactusADM	89%	95%	93%	83%	76%	89%	95%	93%	84%	79%	90%	90%
	calculix	81%	92%	81%	73%	58%	84%	93%	82%	76%	58%	86%	81%
	dealII	86%	94%	83%	80%	63%	88%	95%	84%	82%	66%	89%	83%
	gamess	88%	95%	85%	83%	68%	90%	95%	85%	85%	70%	90%	85%
	GemsFDTD	86%	94%	87%	80%	71%	87%	95%	89%	82%	75%	89%	90%
	gromacs	85%	94%	83%	79%	63%	87%	94%	85%	81%	66%	88%	82%
	lbm	85%	94%	89%	79%	77%	88%	98%	96%	84%	82%	89%	93%
	leslie3d	88%	95%	92%	83%	79%	88%	96%	96%	83%	80%	90%	91%
	milc	85%	94%	81%	79%	66%	87%	94%	84%	81%	68%	89%	89%
	namd	88%	95%	85%	82%	68%	90%	95%	85%	84%	70%	91%	85%
	povray	86%	94%	82%	80%	61%	88%	94%	83%	82%	65%	88%	81%
	soplex	84%	95%	84%	78%	75%	85%	95%	87%	80%	75%	87%	84%
	sphinx3	87%	95%	91%	81%	72%	88%	95%	92%	83%	75%	89%	89%
	tonto	87%	94%	85%	82%	67%	89%	95%	85%	84%	70%	90%	84%
	wrf	87%	95%	92%	82%	75%	88%	95%	94%	83%	77%	89%	91%
	zeusmp	85%	93%	88%	78%	71%	86%	94%	92%	80%	77%	88%	89%
	GeoMean	**86%**	**94%**	**86%**	**80%**	**69%**	**88%**	**95%**	**88%**	**82%**	**72%**	**89%**	**87%**
SPEC-OMP2001	applu.M	97%	98%	90%	96%	92%	92%	98%	90%	88%	81%	95%	94%
	apsi.M	100%	99%	97%	99%	98%	101%	99%	100%	100%	98%	100%	100%
	fma3d.M	92%	97%	83%	89%	77%	90%	98%	85%	88%	77%	93%	92%
	galgel.M	99%	99%	95%	98%	93%	99%	99%	95%	98%	93%	97%	95%
	mgrid.M	99%	100%	98%	99%	99%	99%	100%	101%	99%	99%	99%	98%
	swim.M	98%	99%	95%	97%	95%	94%	99%	98%	94%	93%	97%	96%
	wupwise.M	98%	99%	88%	97%	93%	95%	97%	88%	93%	87%	96%	94%
	GeoMean	**98%**	**99%**	**92%**	**96%**	**92%**	**96%**	**99%**	**94%**	**94%**	**89%**	**97%**	**96%**
NAS-NPB	bt.A	100%	100%	98%	99%	98%	100%	100%	99%	100%	100%	99%	98%
	cg.A	97%	100%	99%	97%	96%	98%	100%	95%	98%	97%	96%	94%
	ft.A	99%	100%	98%	99%	98%	92%	100%	99%	92%	90%	99%	100%
	is.A	100%	100%	98%	100%	100%	101%	100%	97%	101%	102%	99%	97%
	lu.A	100%	99%	99%	99%	99%	100%	99%	99%	99%	99%	98%	99%
	mg.A	99%	100%	98%	100%	100%	100%	100%	102%	100%	100%	99%	98%
	sp.A	99%	100%	98%	99%	98%	99%	100%	99%	99%	99%	99%	98%
	GeoMean	**99%**	**100%**	**98%**	**99%**	**99%**	**99%**	**100%**	**99%**	**98%**	**98%**	**98%**	**98%**
*	**GeoMean**	**89%**	**95%**	**89%**	**84%**	**75%**	**90%**	**96%**	**90%**	**86%**	**76%**	**91%**	**89%**

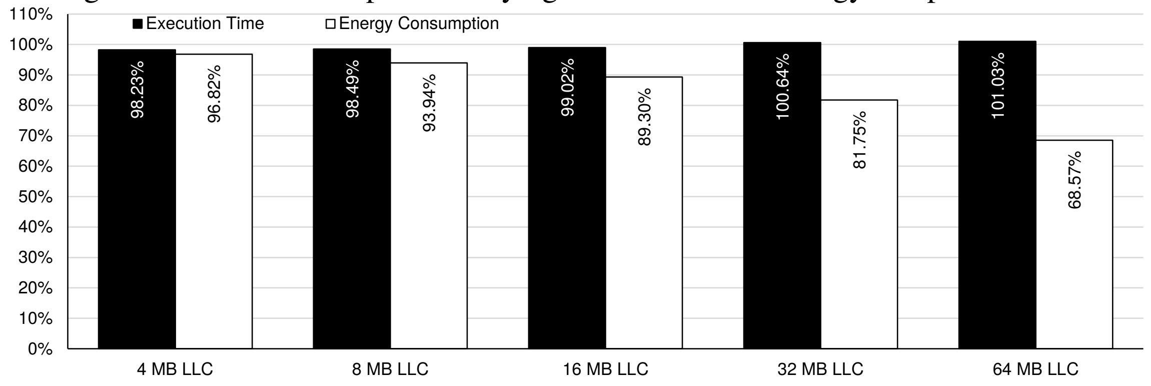
On average, the performance was harmed by less than 4%, while saving on average 25% of total cache energy.

5.5 Design Space Exploration

This section presents a parameter sensitivity exploration for our DEWP mechanism. We study the effect of varying the LLC size when using our mechanism. For these experiments, DEWP was only enabled on the LLC.

Figure 5.10 shows the results presented in terms of energy consumption and execution time for each LLC size normalized to the same LLC sized baseline without our mechanism.

Figure 5.10: DEWP: Impact of varying the LLC size on energy and performance.



This evaluation with different LLC sizes shows that our mechanism when applied to small caches tends to produce lower gains in terms of energy consumption, and some performance improvement. This is because smaller caches tend to consume more dynamic energy, due to an increased amount of accesses. However, for these smaller caches, our mechanism benefits in terms of performance, by the early write-back and early eviction of the cache lines, after their last update or last access.

When our mechanism is applied to a bigger cache, it enables the cache to consume less energy, however, in terms of performance the under predictions can hurt the performance. Moreover, when DEWP is applied in a system using 64 MB LLC, it can keep the same energy consumption as the system with 16 MB LLC, due to its beneficial static energy savings.

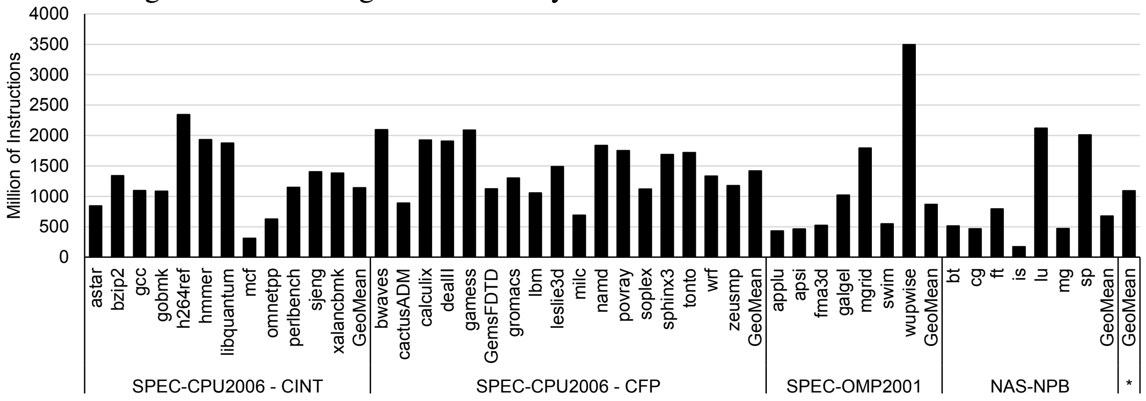
Although our mechanisms have a small train phase for each new pattern, the number of instructions between a system context-switch could influence the mechanism behavior. If the context-switch happens before the mechanism makes use of the trained patterns, the gains might be compromised. Figure 5.11 presents the average number of cycles between the OS context switch for SPEC-CPU2006, SPEC-OMP2001 and NAS-NPB suites executing on a real Intel Sandy Bridge machine.

We can observe that on average after every 1,090 million instructions happens a context switch. Our traces for SPEC-CPU2006 have 200 million instructions and 140 million instructions on average for the parallel benchmarks. It means that our mechanisms DSBP and DEWP were evaluated using a tough scenario, considering that inside a real machine, our mechanism would have more time to make use and obtain the gains from the trained patterns. However, even in this situation, our mechanisms could achieve high energy savings with a low performance degradation.

5.6 Summary

In this chapter, we introduced the DEWP mechanism to optimize the energy efficiency by keeping only alive data in the cache memories. Our mechanism achieves this by predicting access patterns of the cache lines. Using this information, DEWP is able to turn off the cache lines as soon as their data becomes dead, to write-back early dirty cache lines after their last write operation happens and also to reduce cache pollution by prioritizing the eviction of completely dead cache lines. DEWP works independently of the cache replacement algorithm and it does not modify the cache coherence protocol.

Figure 5.11: Average number of cycles between the OS context-switch.



The DEWP mechanism requires a low storage size overhead to achieve accurate predictions (73% correct predictions and 14% of underpredictions). DEWP achieves a 25% energy reduction on average compared to the baseline. The execution time was increased by 4% on average for single-threaded and multi-threaded applications. DEWP achieves close to 100% of the potential savings that a perfect (oracle) mechanism would achieve.

6 COMBINING DSBP AND DEWP

In the previous chapters, DSBP and DEWP were used separately in all the cache levels. This chapter presents the evaluation results for the integration of our two mechanisms DSBP and DEWP, executing together in a system. This integrated mechanism will be called MIXED.

6.1 Introduction

In order to choose the best integration between the predictors, we first consider the results presented in Section 1.2. It is possible to see that the three major sources of energy consumption inside the cache sub-system are the dynamic (45%) and static (17%) energy from the L1 data cache and the static energy from the LLC (19%). It means that DSBP, which saves static and dynamic energy, is suitable for the L1 cache, while the DEWP mechanism would reduce the main energy consumption from the LLC.

The second aspect to be considered is presented in Section 4.1, in which we can observe that cache levels closer to the processor (L1 and L2) tend to access less sub-blocks before the line is evicted. It means that DSBP has more potential to save energy on L1 and L2 caches because it works on the sub-block granularity. The cache lines in the LLC tend to be used completely, which is an access behavior that is suitable for DEWP. For these reasons, MIXED implements the DSBP mechanism on the L1 and L2 cache levels, while DEWP is used on the LLC.

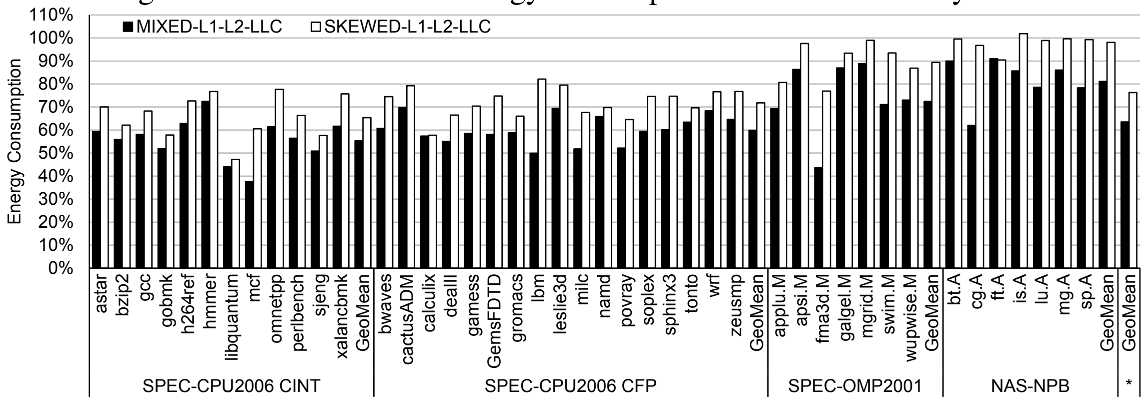
6.2 Evaluation

The baseline configuration for the processor is based on the Intel Sandy Bridge as shown in Table 3.6. The single-threaded applications from SPEC-CPU2006 and multi-threaded (8 threads) applications from SPEC-OMP2001 and NAS-NPB suites were used as workloads to evaluate MIXED.

In order to evaluate our proposed mechanism, we compare it with the SKEWED mechanism, explained in Section 2.2.6. We implemented SKEWED using three tables with 8192 entries each, with a 2 bit counter per entry to perform the prediction. Each table is indexed using a different hash function, in order to obtain three different sub-predictions, which are combined to form the final prediction.

A similar energy modeling methodology as presented in Sections 4.3 and 5.3 was used to evaluate our combination of the DSBP and DEWP mechanisms.

Figure 6.1: MIXED: Total energy consumption of the cache sub-system.



6.2.1 Energy Savings

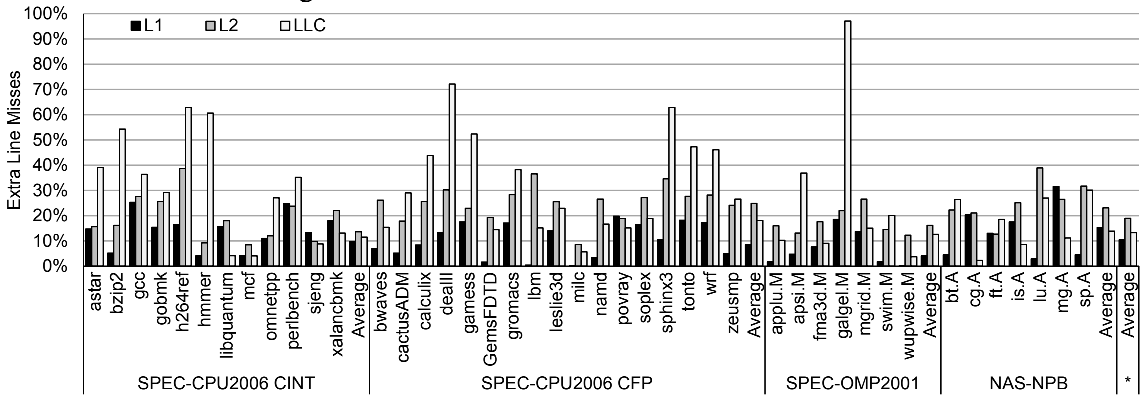
The cache energy efficiency is increased by using our two mechanisms DSBP and DEWP together. The results in Figure 6.1 are shown in terms of energy consumption for the cache memory sub-system, normalized to the baseline. MIXED achieves on average 37% of energy savings compared to the baseline, outperforming the related work by 13%.

Table 6.1 presents the results for the MIXED, DSBP, DEWP and SKEWED mechanisms when the predictors are applied to all the cache levels, also comparing with the oracle mechanisms applied on the L1 and LLC. Notice that the table shows the energy consumption considering the overall cache system.

6.2.2 Performance Impact

Our mechanism can influence the execution time of an application in different ways. MIXED can increase the performance with early evictions of dead lines, enabling more effective space in the cache memory, while early write-backs of last written lines can potentially reduce the memory controller contention. On the other hand, our mechanism can hurt the performance by causing extra cache misses because of under predictions.

Figure 6.2: MIXED: Normalized cache misses.



As shown in the previous section, successfully predicting dead cache lines can significantly reduce cache energy consumption. However, incorrect predictions may introduce a negative impact on cache performance and actually increase energy consumption due to

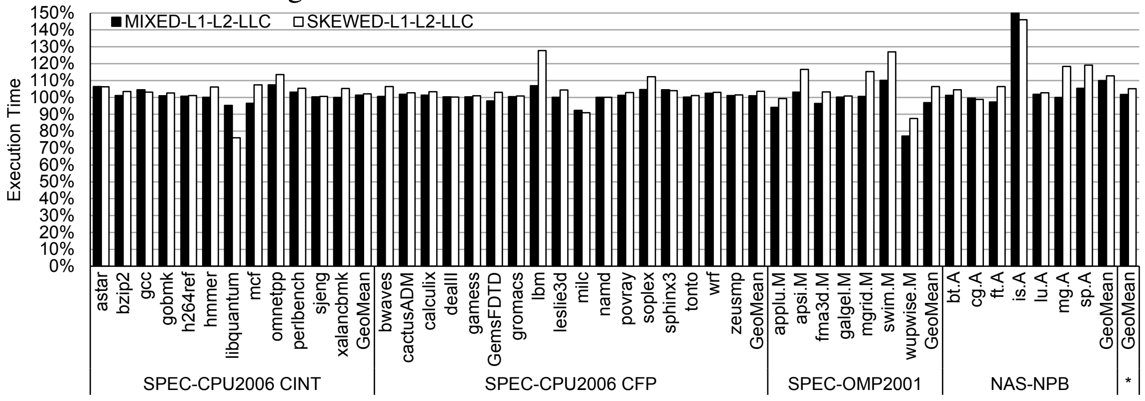
Table 6.1: MIXED: Total energy consumption of the cache sub-system.

		MIXED L1-L2-LLC	DEWP L1-L2-LLC	DSBP L1-L2-LLC	SKEWED L1-L2-LLC	L1 Oracle Line Usage	LLC Oracle Line Usage	L1 Oracle Sub-block Usage
SPEC-CPU2006 - CINT	astar	59%	67%	63%	70%	86%	87%	58%
	bzip2	56%	58%	56%	62%	86%	82%	63%
	gcc	58%	63%	59%	68%	85%	85%	65%
	gobmk	52%	54%	52%	58%	86%	78%	66%
	h264ref	63%	70%	63%	73%	90%	86%	71%
	hmmer	73%	74%	73%	77%	92%	89%	79%
	libquantum	44%	77%	43%	47%	82%	85%	56%
	mcf	38%	52%	39%	61%	81%	81%	58%
	omnetpp	61%	74%	63%	78%	86%	91%	58%
	perlbench	56%	62%	57%	66%	88%	83%	57%
	sjeng	51%	55%	52%	58%	87%	78%	67%
	xalancbmk	62%	71%	62%	76%	89%	89%	66%
	GeoMean	**55%**	**64%**	**56%**	**65%**	**86%**	**84%**	**63%**
SPEC-CPU2006 - CFP	bwaves	61%	73%	60%	75%	90%	88%	72%
	cactusADM	70%	76%	70%	79%	90%	90%	79%
	calculix	57%	58%	58%	58%	86%	81%	74%
	dealII	55%	63%	55%	66%	89%	83%	68%
	gamess	59%	68%	59%	70%	90%	85%	65%
	GemsFDTD	58%	71%	63%	75%	89%	90%	61%
	gromacs	59%	63%	59%	66%	88%	82%	71%
	lbm	50%	77%	51%	82%	89%	93%	60%
	leslie3d	69%	79%	71%	80%	90%	91%	73%
	milc	52%	66%	53%	68%	89%	89%	65%
	namd	66%	68%	66%	70%	91%	85%	61%
	povray	52%	61%	52%	65%	88%	81%	60%
	soplex	59%	75%	60%	75%	87%	84%	53%
	sphinx3	60%	72%	61%	75%	89%	89%	63%
	tonto	63%	67%	64%	70%	90%	84%	76%
	wrf	68%	75%	70%	77%	89%	91%	70%
	zeusmp	65%	71%	65%	77%	88%	89%	69%
	GeoMean	**60%**	**69%**	**61%**	**72%**	**89%**	**87%**	**67%**
SPEC OMP2001	applu.M	69%	92%	70%	81%	95%	94%	65%
	apsi.M	86%	98%	86%	98%	100%	100%	77%
	fma3d.M	44%	77%	49%	77%	93%	92%	52%
	galgel.M	87%	93%	87%	93%	97%	95%	85%
	mgrid.M	89%	99%	89%	99%	99%	98%	76%
	swim.M	71%	95%	71%	93%	97%	96%	66%
	wupwise.M	73%	93%	72%	87%	96%	94%	73%
	GeoMean	**72%**	**92%**	**74%**	**89%**	**97%**	**96%**	**70%**
NAS-NPB	bt.A	90%	98%	90%	100%	99%	98%	79%
	cg.A	62%	96%	61%	97%	96%	94%	43%
	ft.A	91%	98%	92%	90%	99%	100%	80%
	is.A	86%	100%	85%	102%	99%	97%	57%
	lu.A	79%	99%	79%	99%	98%	99%	63%
	mg.A	86%	100%	86%	100%	99%	98%	78%
	sp.A	78%	98%	79%	99%	99%	98%	63%
	GeoMean	**81%**	**99%**	**81%**	**98%**	**98%**	**98%**	**65%**
*	**GeoMean**	**63%**	**75%**	**64%**	**76%**	**91%**	**89%**	**66%**

extra cache sub-block misses. Figure 6.2 shows the total number of extra cache misses normalized to the number of cache misses from the baseline cache architecture.

Figure 6.3 shows the execution time of our mechanism normalized to the baseline. On average, the performance was degraded by less than 2%, while saving on average 37% of cache energy.

Figure 6.3: MIXED: Normalized execution time.



6.3 Summary

Results combining both proposals that aim to reduce energy consumption by predicting the cache line usage pattern showed that maximum gains can be obtained when applying both mechanisms together in a system.

Combining DSBP and DEWP can outperform in terms of energy savings the DSBP or DEWP separately by 1% and 12% (percentage points) respectively. However, this combination impacts on the execution time by 1.77%, which represents a lower overhead than DSBP (2.25%) and DEWP (3.65%).

7 CONCLUSIONS AND FUTURE WORK

Energy consumption has become an important factor in multi-core designs. Thus, mechanisms that enable energy savings while maintaining the performance are essential for keeping the power budget and an efficient operation.

Considering that cache memories consume a great portion of energy inside the chip, this thesis proposed two mechanisms, Dead Sub-Block Predictor (DSBP) and Dead Line and Early Write-Back Predictor (DEWP), to increase the energy efficiency of cache memories.

To our knowledge, the DSBP mechanism is the first proposal to exploit dead cache line prediction at the sub-block granularity. DSBP is used to reduce energy consumption in the cache sub-system by loading into the cache only those sub-blocks predicted to be useful, and turning off active sub-blocks when they become dead. In addition, the LRU replacement policy is improved by prioritizing dead lines for eviction. This modification reduces the number of dead lines that are kept in the cache, which leads to a better utilization of the cache space.

DSBP results in terms of energy consumption show significant improvements on all cache levels, with average savings of 22%, 27% and 36% when applied to the L1, L1-L2 and L1-L2-LLC respectively, compared to the baseline.

We introduced the DEWP mechanism to optimize the energy efficiency and performance of last level caches by reducing cache pollution and memory pressure. Our mechanism achieves this by predicting access patterns of the cache lines. Using this information, DEWP is able to turn off the cache lines as soon as their data becomes dead, to write-back early dirty cache lines after their last write operation happens and also to reduce cache pollution by prioritizing the eviction of completely dead cache lines.

DEWP achieves energy savings on all cache levels, with average savings of 11%, 16% and 25% when applied to the L1, L1-L2 and L1-L2-LLC respectively, compared to the baseline.

Additionally, integrating DSBP and DEWP, using DSBP on the L1 and L2 and DEWP on the LLC, produces 37% of energy savings for all the cache levels.

7.1 Future Work

Multiple future work opportunities can be explored, in order to reduce the overhead and improve the mechanism accuracy, enabling higher energy savings.

Enabling the power gating on the sub-block or cache line level can impact the design of cache memories by requiring a higher number of power lines. Moreover, the power gating operation can also generate some peak energy when turning on the cache lines.

These impacts can be evaluated in order to propose ways to reduce this impact in terms of area and energy consumption.

The prediction accuracy for the prefetched lines can be improved by including an extra bit in the cache line to indicate that a particular line was prefetched. Thus, mispredictions caused by unnecessary prefetches would not modify the prediction pattern, increasing the accuracy of our mechanisms.

Turning off sub-blocks inside the cache line can help reduce the heat from the cache memory sub-system, which in turn reduces static energy (LIU et al., 2007). However, the heat reduction benefits of our mechanism are not modeled in this work.

Dead-on-arrival lines can be filtered with bypassing algorithms using the information available from our predictor. Also, our mechanism can be used to design prefetchers that bring into the cache only those sub-blocks that are predicted to be useful, thereby decreasing the bandwidth demand of systems that employ aggressive prefetching.

Mechanisms, such as the one presented by Kim and Gratz (KIM; GRATZ, 2010), could be used together with DSBP to reduce the energy consumption on networks-on-chip by transferring only the useful data between caches. In addition, off-chip mechanisms as presented by Yoon et al. (YOON; JEONG; EREZ, 2011) can also be adopted to reduce the overall off-chip communication during accesses to the main memory, fetching only the sub-blocks predicted to be useful. This evaluation of on-chip and off-chip traffic could show the benefits of our mechanism for the interconnection system.

7.2 Published Papers

The list with published papers since 2010 (second year as PhD student) are present below:

1. CRUZ, E. H. M., DIENER, M., **ALVES, M. A. Z.**, NAVAUX, P. O. A. ***Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols.*** Journal of Parallel and Distributed Computing (JPDC), 2014

2. [1] **ALVES, M. A. Z.**, VILLAVIEJA, C., DIENER, M., NAVAUX, P. O. A. ***Energy Efficient Last Level Caches via Last Read/Write Prediction.*** Int. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD), 2013, Porto de Galinhas, PE.

3. MOREIRA, F. B., **ALVES, M. A. Z.**, DIENER, M., NAVAUX, P. O. A. ***Influência das Características de Processadores e Aplicações no Nível de Blocos Básicos.*** Symp. on Computing Systems (WSCAD-SSC), 2013, Porto de Galinhas, PE.

4. [2] **ALVES, M. A. Z.**, KHUBAIB, EBRAHIMI, E., NARASIMAN, V., VILLAVIEJA, C., NAVAUX, P. O. A., PATT, Y. N. ***Energy Savings via Dead Sub-Block Prediction***. Int. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD), 2012, New York, United States.

5. CRUZ, E. H. M., **ALVES, M. A. Z.**, CARISSIMI, A. S., NAVAUX, P. O. A., RIBEIRO, C. P., MEHAUT, J. *Using Memory Access Traces to Map Threads and Data on Hierarchical Multi-core Platforms*. Workshop on Advances in Parallel

[1]Represents the one of the most important publications related to this thesis mechanism DEWP.
[2]Represents the one of the most important publications related to this thesis mechanism DSBP.

and Distributed Computing Models. (IPDPS-APDCM), 2011, Anchorage, United States.

6. MOREIRA, F. B., MOLINA DA CRUZ, E. H., **ALVES, M. A. Z.**, NAVAUX, P. O. A. *Scratchpad Memories for Parallel Applications in Multi-core Architectures*. (Best paper award) Symp. on Computing Systems (WSCAD-SSC), 2011, Vitória, ES.

7. FREITAS, H. C., **ALVES, M. A. Z.**, SCHNORR, L. M., NAVAUX, P. O. A. *Impact of Parallel Workloads on NoC Architecture Design*. Euromicro Int. Conference on Parallel, Distributed and Network-Based Computing. 2010, Pisa, Italy. p. 551-555.

8. RUTZIG, M. B., MADRUGA, F. L., **ALVES, M. A. Z.**, FREITAS, H. C., BECK FILHO, A. C. S., MAILLARD, N., NAVAUX, P. O. A., CARRO, L. *TLP and ILP exploitation through a Reconfigurable Multiprocessor System*. IEEE Int. Parallel And Distributed Processing Symp. (IPDPS), 2010, Atlanta, United States, p. 1-8.

9. DIENER, M., MADRUGA, F. L., RODRIGUES, E. R., **ALVES, M. A. Z.**, SCHNEIDER, J., NAVAUX, P. O. A., HEIß, H. *Evaluating Thread Placement Based on Memory Access Patterns for Shared Cache Multi-core Processors*. Int. Symp. On Advances Of High Performance Computing And Networking (AHPCN), 2010 Melbourne, Australia.

10. MOR, S. D. K., **ALVES, M. A. Z.**, LIMA, J. V. F., MAILLARD, N., NAVAUX, P. O. A. *Eficiência Energética em Computação de Alto Desempenho: Uma Abordagem em Arquitetura e Programação para Green Computing*. Seminário Integrado de Software e Hardware (SEMISH), 2010, Belo Horizonte, MG.

11. **ALVES, M. A. Z.**, CERA, M. C., LIMA, J. V. F., MAILLARD, N., NAVAUX, P. O. A. *Enhancing Energy Efficiency using Efficient Parallel Programming Techniques*. Latin-American Conference on High Performance Computing (CLCAR), 2010, Gramado, RS.

12. **ALVES, M. A. Z.**, NAVAUX, P. O. A. *Intermediary Cache Memory Level for Data Exchange and Interactions of Parallel Scientific Applications*. Latin-American Conference on High Performance Computing (CLCAR), 2010, Gramado, RS.

13. CRUZ, E. H. M., **ALVES, M. A. Z.**, NAVAUX, P. O. A. *Process Mapping Based on Memory Access Traces*. Symp. on Computing Systems (WSCAD-SCC), 2010, Petrópolis, RJ, p. 72-79

REFERENCES

ABELLA, J. et al. IATAC: a smart predictor to turn-off l 2 cache lines. **ACM Transactions on Architecture and Code Optimization (TACO)**, [S.l.], v.2, n.1, p.55–77, 2005.

AGARWAL, V. et al. Clock rate versus IPC: the end of the road for conventional microarchitectures. **ACM SIGARCH Computer Architecture News**, [S.l.], v.28, n.2, p.248–259, 2000.

ALVES, M. et al. Energy Savings via Dead Sub-Block Prediction. In: IEEE INT. SYMP. ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING. **Proceedings...** IEEE, 2012. p.51–58. (SBAC-PAD'12).

ALVES, M. et al. Energy Efficient Last Level Caches via Last Read/Write Prediction. In: IEEE INT. SYMP. ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING. **Proceedings...** IEEE, 2013. p.73–80. (SBAC-PAD'13).

ARGOLLO, E. et al. COTSon: infrastructure for full system simulation. **ACM SIGOPS Operating Systems Review**, [S.l.], v.43, n.1, p.52–61, 2009.

AUSTIN, T.; LARSON, E.; ERNST, D. SimpleScalar: an infrastructure for computer system modeling. **IEEE Micro**, [S.l.], v.35, n.2, p.59–67, 2002.

BAER, J.-L.; CHEN, T.-F. An effective on-chip preloading scheme to reduce data access penalty. In: ACM/IEEE CONF. ON SUPERCOMPUTING. **Proceedings...** ACM, 1991. p.176–186. (SC'91).

BAILEY, D. H. et al. **The nas parallel benchmarks**. [S.l.]: The International Journal of Supercomputer Applications, 1991.

BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In: USENIX ANNUAL TECHNICAL CONFERENCE, FREENIX TRACK. **Anais...** [S.l.: s.n.], 2005. p.41–46.

BERTOZZI, D. et al. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.16, n.2, p.113–129, 2005.

BINKERT, N. et al. The M5 simulator: modeling networked systems. **IEEE Micro**, [S.l.], v.26, n.4, p.52–60, 2006.

BINKERT, N. et al. The gem5 simulator. **ACM SIGARCH Computer Architecture News**, [S.l.], v.39, n.2, p.1–7, 2011.

BJERREGAARD, T.; MAHADEVAN, S. A survey of research and practices of network-on-chip. **ACM Computing Surveys (CSUR)**, [S.l.], v.38, n.1, p.1, 2006.

BOJAN, T. et al. Functional coverage measurements and results in post-Silicon validation of Core 2 duo family. In: INT. HIGH LEVEL DESIGN VALIDATION AND TEST WORKSHOP. **Anais...** IEEE, 2007. p.145–150. (HLVDT'07).

BORKAR, S. Design challenges of technology scaling. **IEEE Micro**, [S.l.], v.19, n.4, p.23–29, 1999.

BUTKO, A. et al. Accuracy evaluation of GEM5 simulator system. In: RECONFIGURABLE COMMUNICATION-CENTRIC SYSTEMS-ON-CHIP (RECOSOC), 2012 7TH INTERNATIONAL WORKSHOP ON. **Anais...** [S.l.: s.n.], 2012. p.1–7.

CHEN, C. F. et al. Accurate and complexity-effective spatial pattern prediction. In: IEEE INT. SYMP. ON HIGH PERFORMANCE COMPUTER ARCHITECTURE. **Proceedings...** IEEE, 2004. p.276–287. (HPCA'04).

DESIKAN, R.; BURGER, D.; KECKLER, S. Measuring experimental error in microprocessor simulation. In: IEEE/ACM INT. SYMP. ON COMPUTER ARCHITECTURE. **Proceedings...** IEEE, 2001. p.266–277. (ISCA'01).

DOWECK, J. White Paper Inside Intel® Core® Microarchitecture and Smart Memory Access. **Intel Corporation**, [S.l.], 2006.

FLEMING, P. J.; WALLACE, J. J. How Not to Lie with Statistics: the correct way to summarize benchmark results. **Commun. ACM**, New York, NY, USA, v.29, n.3, p.218–221, Mar. 1986.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture**: a quantitative approach. 4th.ed. USA: Elsevier, 2007.

HENNING, J. L. SPEC CPU2006 benchmark descriptions. **ACM SIGARCH Computer Architecture News**, [S.l.], v.34, n.4, p.1–17, 2006.

HUANG, M. et al. L1 data cache decomposition for energy efficiency. In: ACM INT. SYMP. ON LOW POWER ELECTRONICS AND DESIGN. **Proceedings...** [S.l.: s.n.], 2001. p.10–15. (ISPLED'01).

IBM. **System/370 model 155 theory of operation/diagrams manual v. 5**: buffer control unit. Poughkeepsie, N.Y.: IBM System Products Division, 1974.

INTEL. **Intel Performance Counter Monitor - A better way to measure CPU utilization**. 2012.

INTEL. **Intel 64 and IA-32 Architectures Software Developer's Manual**. 2013. n.September.

JAIN, R. **The art of computer systems performance analysis**: techniques for experimental design, measurement, simulation, and modeling. USA: J. Wiley, 1991.

JOUPPI, N. P. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In: ACM SIGARCH COMPUTER ARCHITECTURE NEWS. **Proceedings...** ACM, 1990. n.3a, p.364–373. (SIGARCH'90, v.18).

KAXIRAS, S.; HU, Z.; MARTONOSI, M. Cache decay: exploiting generational behavior to reduce cache leakage power. In: IEEE/ACM INT. SYMP. ON COMPUTER ARCHITECTURE. **Proceedings...** IEEE, 2001. p.240–251. (ISCA'01).

KHAN, S. et al. Using dead blocks as a virtual victim cache. In: IEEE/ACM INT. CONF. ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES. **Proceedings...** ACM, 2010. p.489–500. (PACT'10).

KHARBUTLI, M.; SOLIHIN, Y. Counter-based cache replacement and bypassing algorithms. **IEEE Transactions on Computers (TC)**, [S.l.], v.57, n.4, p.433–447, 2008.

KIM, C.; BURGER, D.; KECKLER, S. Q. An adaptive, non-uniform cache structure for wire-delay dominated on-chip-caches. In: INT. CONF. ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS. **Proceedings...** IEEE, 2002. p.211–222.

KIM, C.; BURGER, D.; KECKLER, S. W. Non-Uniform Cache Architetures for Wire-Delay Dominated On-Chip Caches. **IEEE Computer**, [S.l.], 2003.

KIM, H.; GRATZ, P. V. Leveraging Unused Cache Block Words to Reduce Power in CMP Interconnect. **IEEE Computer Architecture Letters (CAL)**, [S.l.], v.9, n.1, p.33–36, 2010.

KUMAR, S.; WILKERSON, C. Exploiting spatial locality in data caches using spatial footprints. In: ACM SIGARCH COMPUTER ARCHITECTURE NEWS. **Proceedings...** ACM, 1998. v.26, p.357–368.

LAI, A.-C.; FALSAFI, B. Selective, accurate, and timely self-invalidation using last-touch prediction. In: IEEE/ACM INT. SYMP. ON COMPUTER ARCHITECTURE. **Proceedings...** IEEE, 2000. p.139–148. (ISCA'00).

LAI, A.-C.; FIDE, C.; FALSAFI, B. Dead-block prediction & dead-block correlating prefetchers. In: IEEE/ACM INT. SYMP. ON COMPUTER ARCHITECTURE. **Proceedings...** IEEE, 2001. p.144–154. (ISCA'01).

LEE, H.; TYSON, G.; FARRENS, M. Eager writeback-a technique for improving bandwidth utilization. In: IEEE/ACM INT. SYMP. ON MICROARCHITECTURE. **Proceedings...** IEEE, 2000. p.11–21. (MICRO'00).

LI, S. et al. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In: IEEE/ACM INT. SYMP. ON MICROARCHITECTURE. **Proceedings...** IEEE, 2009. p.469–480. (MICRO'09).

LI, S. et al. The McPAT Framework for Multicore and Manycore Architectures: simultaneously modeling power, area, and timing. **ACM Transactions on Architecture and Code Optimization (TACO)**, [S.l.], v.10, n.1, p.5, 2013.

LINDNER, M. A. **libconfig - C/C++ configuration file library**. Available at: http://www.hyperrealm.com/libconfig/. Accessed on: 14-January-2013.

Linux Kernel Developers. **Performance analysis tools for Linux**. Available at: https://perf.wiki.kernel.org. Accessed on: 14-January-2013.

LIU, Y. et al. Accurate temperature-dependent integrated circuit leakage power estimation is easy. In: EDA CONF. ON DESIGN, AUTOMATION AND TEST IN EUROPE. **Proceedings...** EDA Consortium, 2007. p.1526–1531. (DATE'07).

MAGNUSSON, P. et al. Simics: a full system simulation platform. **IEEE Micro**, [S.l.], v.35, n.2, p.50–58, Feb 2002.

MARTY, M. et al. General Execution-driven Multiprocessor Simulator. In: ISCA TUTORIAL. **Proceedings...** IEEE, 2005. (ISCA'05).

MURALIMANOHAR, N.; BALASUBRAMONIAN, R.; JOUPPI, N. Architecting efficient interconnects for large caches with CACTI 6.0. **IEEE Micro**, [S.l.], v.28, n.1, p.69–79, 2008.

OLUKOTUN, K. et al. The Case for a Single-Chip Multiprocessor. In: IEEE INT. SYMP. ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS. **Proceedings...** IEEE, 1996. p.2–11. (ASPLOS'96).

PATEL, A. et al. MARSSx86: a full system simulator for x86 cpus. In: DESIGN AUTOMATION CONFERENCE. **Proceedings...** ACM, 2011. p.261–305. (DAC'11).

PATIL, H. et al. Pinpointing Representative Portions of Large Intel® Itanium® Programs with Dynamic Instrumentation. In: IEEE/ACM INT. SYMP. ON MICROARCHITECTURE. **Proceedings...** IEEE, 2004. p.81–92. (MICRO'04).

POWELL, M. et al. Gated-V_{DD}: a circuit technique to reduce leakage in deep-submicron cache memories. In: ACM INT. SYMP. ON LOW POWER ELECTRONICS AND DESIGN. **Proceedings...** ACM, 2000. p.90–95. (ISPLED'00).

PUJARA, P.; AGGARWAL, A. Cache noise prediction. **IEEE Transactions on Computers (TC)**, [S.l.], v.57, n.10, p.1372–1386, 2008.

RENAU, J. et al. **SESC simulator**. Available at: http://sesc.sourceforge.net. Accessed on: 10-January-2013.

SAITO, H. et al. Large system performance of SPEC OMP2001 benchmarks. In: INT. SYMP. ON HIGH PERFORMANCE COMPUTING. **Anais...** [S.l.: s.n.], 2006. p.370–379. (ISHPC'06).

Semiconductor Industry Association. Model for assessment of CMOS technologies and roadmaps (MASTAR). **Semiconductor Industry Association**, [S.l.], 2007.

SHERWOOD, T. et al. Automatically characterizing large scale program behavior. In: ACM SIGARCH COMPUTER ARCHITECTURE NEWS. **Proceedings...** ACM, 2002. n.5, p.45–57. (SIGARCH'02, v.30).

SHERWOOD, T.; PERELMAN, E.; CALDER, B. Basic block distribution analysis to find periodic behavior and simulation points in applications. In: PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 2001. PROCEEDINGS. 2001 INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2001. p.3–14.

SMITH, J. E.; SOHI, G. S. The Microarchitecture of Superscalar Processors. **IEEE Micro**, [S.l.], v.83, n.12, p.1609–1624, 1995.

STUECHELI, J. et al. The virtual write queue: coordinating dram and last-level cache policies. In: IEEE/ACM INT. SYMP. ON COMPUTER ARCHITECTURE. **Proceedings...** IEEE, 2010. p.72–82. (ISCA'10).

UBAL, R. et al. Multi2Sim: a simulation framework to evaluate multicore-multithread processors. In: IEEE 19TH INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, PAGE (S). **Anais...** [S.l.: s.n.], 2007. p.62–68.

UBAL, R. et al. Multi2Sim: a simulation framework for cpu-gpu computing. In: INT. CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES. **Proceedings...** [S.l.: s.n.], 2012. p.335–344.

UNGERER, T.; ROBIC, B.; SILC, J. Multithreaded Processors. **The Computer Journal**, [S.l.], v.45, n.3, p.320–348, 2002.

WANG, Z.; KHAN, S. M.; JIMÉNEZ, D. A. Improving writeback efficiency with decoupled last-write prediction. In: IEEE/ACM INT. SYMP. ON COMPUTER ARCHITECTURE. **Proceedings...** IEEE, 2012. p.309–320. (ISCA'12).

WANG, Z.; KHAN, S. M.; JIMÉNEZ, D. A. Rank idle time prediction driven last-level cache writeback. In: ACM SIGPLAN WORKSHOP ON MEMORY SYSTEMS PERFORMANCE AND CORRECTNESS. **Proceedings...** ACM, 2012. p.21–29. (MSPC'12).

WEAVER, V.; MCKEE, S. Are cycle accurate simulations a waste of time. In: WORKSHOP ON DUPLICATING, DECONSTRUCTING, AND DEBUNKING. **Proceedings...** [S.l.: s.n.], 2008. p.40–53.

YEH, T.-Y.; PATT, Y. N. Two-level adaptive training branch prediction. In: IEEE/ACM INT. SYMP. ON MICROARCHITECTURE. **Proceedings...** [S.l.: s.n.], 1991. p.51–61. (MICRO'91).

YEH, T.-Y.; PATT, Y. N. Alternative implementations of two-level adaptive branch prediction. In: ACM SIGARCH COMPUTER ARCHITECTURE NEWS. **Proceedings...** [S.l.: s.n.], 1992. n.2, p.124–134. (SIGARCH'92, v.20).

YOON, D. H.; JEONG, M. K.; EREZ, M. Adaptive granularity memory systems: a tradeoff between storage efficiency and throughput. In: IEEE/ACM INT. SYMP. ON COMPUTER ARCHITECTURE. **Proceedings...** IEEE, 2011. p.295–306. (ISCA'11).

YOURST, M. PTLsim: a cycle accurate full system x86-64 microarchitectural simulator. In: IEEE INT. SYMP. ON PERFORMANCE ANALYSIS OF SYSTEMS & SOFTWARE. **Proceedings...** IEEE, 2007. p.23–34. (ISPASS'07).

YUFFE, M. et al. A fully integrated multi-CPU, GPU and memory controller 32nm processor. In: INT. SOLID-STATE CIRCUITS CONFERENCE DIGEST OF TECHNICAL PAPERS. **Anais...** IEEE, 2011. p.264–266. (ISSCC'11).

APPENDIX A ADDITIONAL VALIDATION RESULTS

This appendix presents the absolute numbers comparing the real machine execution and the simulation of SPEC-CPU2006, SPEC-OMP2001 and NAS-NPB benchmark suites.

A.1 SPEC-CPU2006 Results

Table A.1: SPEC-CPU2006 results for the Core 2 Duo machine and SiNUCA.

		IPC			Branch Miss		L1 DATA MPKI	
		Real	**Sim**	**Diff**	**Real**	**Sim**	**Real**	**Sim**
SPEC-CPU2006 - CINT	astar	0.79	0.49	38%	3.63%	5.83%	-	20
	bzip2	1.13	0.77	32%	7.93%	13.49%	19	16
	gcc	0.71	0.60	16%	2.69%	5.34%	92	14
	gobmk	0.99	0.84	15%	10.69%	10.70%	16	11
	h264ref	1.79	1.40	22%	1.63%	2.97%	10	4
	hmmer	1.88	1.52	19%	0.60%	0.58%	6	12
	libquantum	1.07	0.42	61%	3.06%	2.60%	77	24
	mcf	0.22	0.14	39%	5.97%	6.13%	258	137
	omnetpp	0.81	0.39	52%	3.82%	2.96%	-	35
	perlbench	1.13	1.02	10%	3.77%	2.80%	26	14
	sjeng	1.28	1.07	17%	6.00%	6.65%	5	2
	xalancbmk	0.81	0.93	15%	3.31%	1.42%	-	31
	GeoMean	**0.94**	**0.68**	**24%**	**3.57%**	**3.82%**	**26**	**16**
SPEC-CPU2006 - CFP	bwaves	1.51	1.04	31%	4.33%	5.55%	14	35
	cactusADM	0.58	0.67	16%	1.10%	1.81%	40	20
	calculix	1.79	1.01	44%	1.93%	7.58%	6	5
	dealII	0.77	1.15	48%	3.62%	3.00%	-	11
	gamess	1.40	1.53	9%	1.74%	2.15%	8	3
	GemsFDTD	0.69	0.66	4%	0.67%	0.27%	34	51
	gromacs	0.84	1.11	31%	4.86%	4.83%	24	9
	lbm	0.59	0.51	14%	0.87%	0.45%	42	53
	leslie3d	0.84	0.67	20%	0.20%	1.54%	95	63
	milc	0.63	0.65	3%	0.22%	0.00%	99	22
	namd	0.82	1.96	138%	3.46%	0.20%	-	1
	povray	0.84	1.13	34%	3.46%	4.32%	-	23
	soplex	0.81	0.56	31%	3.54%	4.11%	-	30
	sphinx3	1.04	1.01	3%	2.66%	4.09%	15	16
	tonto	1.02	1.53	49%	1.44%	1.76%	31	5
	wrf	1.01	0.91	10%	0.72%	2.48%	23	24
	zeusmp	0.94	0.96	2%	0.73%	1.54%	33	13
	GeoMean	**0.90**	**0.94**	**16%**	**1.46%**	**1.31%**	**27**	**14**
*	**GeoMean**	**0.92**	**0.82**	**19%**	**2.11%**	**2.04%**	**26**	**15**
	StDev			**27%**				

Table A.2: SPEC-CPU2006 results for the Sandy Bridge machine and SiNUCA.

		IPC			**Branch Miss**		**L1 DATA MPKI**	
		Real	**Sim**	**Diff**	**Real**	**Sim**	**Real**	**Sim**
SPEC-CPU2006 - CINT	astar	0.85	0.84	1%	8.21%	5.42%	46	22
	bzip2	1.34	0.89	33%	7.05%	13.50%	18	17
	gcc	1.27	0.79	38%	0.78%	4.96%	49	19
	gobmk	1.17	0.86	26%	7.70%	10.35%	9	12
	h264ref	2.38	1.55	35%	1.35%	2.72%	5	7
	hmmer	1.92	1.54	20%	0.45%	0.59%	10	13
	libquantum	1.86	0.70	62%	0.19%	2.11%	33	90
	mcf	0.32	0.33	5%	5.16%	5.96%	201	201
	omnetpp	0.73	0.81	12%	1.51%	2.99%	50	40
	perlbench	1.34	1.06	21%	2.56%	2.60%	18	16
	sjeng	1.47	1.14	23%	3.96%	6.37%	4	2
	xalancbmk	1.52	1.24	19%	0.49%	1.34%	33	64
	GeoMean	**1.21**	**0.92**	**17%**	**1.85%**	**3.62%**	**22**	**22**
SPEC-CPU2006 - CFP	bwaves	2.09	1.41	33%	1.48%	5.40%	16	35
	cactusADM	1.04	1.03	1%	1.00%	1.76%	33	18
	calculix	1.91	1.11	42%	1.51%	7.62%	7	6
	dealII	1.94	1.13	42%	1.07%	2.95%	18	20
	gamess	2.07	1.59	23%	1.12%	1.95%	7	4
	GemsFDTD	1.17	1.11	5%	0.60%	0.27%	55	55
	gromacs	1.29	1.09	15%	3.76%	4.87%	15	12
	lbm	1.41	1.44	2%	0.28%	0.45%	96	71
	leslie3d	1.49	1.44	4%	1.21%	1.54%	55	109
	milc	0.87	1.28	46%	0.08%	0.00%	36	60
	namd	1.81	1.94	7%	3.44%	0.19%	12	1
	povray	1.75	1.08	38%	1.65%	3.81%	24	30
	soplex	1.24	0.97	22%	2.68%	3.84%	36	73
	sphinx3	1.91	2.10	10%	2.24%	4.03%	19	22
	tonto	1.74	1.55	11%	0.56%	1.71%	18	11
	wrf	1.48	1.55	5%	0.45%	2.47%	25	37
	zeusmp	1.30	1.29	1%	0.64%	1.41%	39	14
	GeoMean	**1.51**	**1.33**	**10%**	**0.99%**	**1.27%**	**24**	**20**
*	**GeoMean**	**1.38**	**1.14**	**12%**	**1.29%**	**1.96%**	**23**	**21**
	StDev			**16%**				

A.2 SPEC-OMP2001 Results

Table A.3: SPEC-OMP2001 results for the Core 2 Duo machine and SiNUCA.

		Average IPC			**Branch Miss**		**L1 DATA MPKI**		**Speedup**		
		Real	**Sim**	**Diff**	**Real**	**Sim**	**Real**	**Sim**	**Real**	**Sim**	**Diff**
1 Thread	applu.M	0.88	0.29	67%	0.00	0.00	73.92	86.82	1.00	1.00	0%
	apsi.M	1.12	0.94	16%	0.01	0.01	25.08	5.63	1.00	1.00	0%
	fma3d.M	0.64	0.15	77%	0.01	0.00	12.98	125.93	1.00	1.00	0%
	galgel.M	1.44	1.16	20%	0.00	0.01	73.27	20.38	1.00	1.00	0%
	mgrid.M	0.90	0.94	5%	0.00	0.01	11.01	18.85	1.00	1.00	0%
	swim.M	0.74	0.52	29%	0.00	0.00	56.21	51.73	1.00	1.00	0%
	wupwise.M	1.96	0.47	76%	0.01	0.00	16.44	28.35	1.00	1.00	0%
	GeoMean	**1.02**	**0.52**	**29%**	**0.00**	**0.00**	**29.16**	**32.04**	**1.00**	**1.00**	**0%**
	StDev			**31%**							**0%**
2 Threads	applu.M	0.59	0.20	66%	0.00	0.00	75.65	165.36	1.34	1.39	4%
	apsi.M	1.05	0.91	14%	0.01	0.01	25.83	10.68	1.88	1.93	3%
	fma3d.M	0.58	0.10	83%	0.01	0.00	13.94	244.41	1.78	1.36	24%
	galgel.M	1.04	0.89	15%	0.00	0.01	87.62	48.45	1.42	1.53	7%
	mgrid.M	0.81	0.75	7%	0.00	0.01	14.39	36.46	1.82	1.61	11%
	swim.M	0.37	0.33	12%	0.00	0.00	72.26	103.90	1.00	1.25	25%
	wupwise.M	1.78	0.42	76%	0.01	0.00	17.49	53.84	1.81	1.82	0%
	GeoMean	**0.79**	**0.40**	**26%**	**0.00**	**0.00**	**33.09**	**63.72**	**1.55**	**1.54**	**6%**
	StDev			**34%**							**10%**

Table A.4: SPEC-OMP2001 results for the Sandy Bridge machine and SiNUCA.

		Average IPC			**Branch Miss**		**L1 DATA MPKI**		**Speedup**		
		Real	**Sim**	**Diff**	**Real**	**Sim**	**Real**	**Sim**	**Real**	**Sim**	**Diff**
1 Thread	applu.M	1.40	0.78	44%	0.00	0.00	43.57	301.62	1.00	1.00	0%
	apsi.M	1.53	1.09	29%	0.00	0.01	19.68	12.30	1.00	1.00	0%
	fma3d.M	0.96	0.40	58%	0.01	0.00	10.68	221.38	1.00	1.00	0%
	galgel.M	1.95	2.10	8%	0.00	0.01	29.67	51.74	1.00	1.00	0%
	mgrid.M	1.35	1.20	11%	0.01	0.01	20.36	28.34	1.00	1.00	0%
	swim.M	1.26	1.16	8%	0.00	0.00	64.57	110.90	1.00	1.00	0%
	wupwise.M	2.49	0.89	64%	0.00	0.00	5.25	166.81	1.00	1.00	0%
	GeoMean	**1.50**	**0.98**	**23%**	**0.00**	**0.00**	**21.12**	**80.70**	**1.00**	**1.00**	**0%**
	StDev			**24%**							**0%**
2 Threads	applu.M	1.10	0.66	40%	0.00	0.00	43.54	295.52	1.57	1.69	8%
	apsi.M	1.48	1.08	27%	0.00	0.01	19.54	12.26	1.93	1.99	3%
	fma3d.M	0.87	0.33	62%	0.01	0.00	10.53	219.40	1.81	1.63	10%
	galgel.M	1.88	1.62	14%	0.00	0.01	28.47	51.40	1.91	1.54	19%
	mgrid.M	1.30	1.17	10%	0.01	0.01	20.33	28.40	1.92	1.95	1%
	swim.M	0.68	0.94	38%	0.00	0.00	64.39	108.82	1.07	1.63	51%
	wupwise.M	2.36	0.66	72%	0.00	0.00	5.26	161.37	1.89	1.49	21%
	GeoMean	**1.28**	**0.83**	**31%**	**0.00**	**0.00**	**20.92**	**79.67**	**1.70**	**1.69**	**10%**
	StDev			**23%**							**17%**
4 Threads	applu.M	0.63	0.48	23%	0.00	0.00	43.52	291.94	1.79	2.51	40%
	apsi.M	1.31	1.07	19%	0.00	0.01	19.94	12.32	3.42	3.93	15%
	fma3d.M	0.73	0.24	67%	0.01	0.00	10.26	217.66	3.03	2.40	21%
	galgel.M	1.58	1.10	30%	0.00	0.01	25.92	50.74	3.16	2.09	34%
	mgrid.M	1.14	1.12	2%	0.01	0.01	20.30	28.55	3.39	3.74	10%
	swim.M	0.34	0.81	141%	0.00	0.00	64.07	107.58	1.06	2.81	165%
	wupwise.M	2.09	0.64	69%	0.00	0.00	5.24	164.48	3.35	2.85	15%
	GeoMean	**0.96**	**0.70**	**28%**	**0.00**	**0.00**	**20.60**	**79.50**	**2.55**	**2.84**	**27%**
	StDev			**47%**							**55%**
8 Threads	applu.M	0.33	0.31	4%	0.00	0.00	42.46	288.66	1.84	3.27	78%
	apsi.M	1.05	1.04	0%	0.00	0.01	19.91	11.95	5.46	7.75	42%
	fma3d.M	0.55	0.15	73%	0.01	0.00	9.70	215.34	4.55	2.97	35%
	galgel.M	0.67	0.72	6%	0.00	0.01	21.74	51.28	2.62	2.46	6%
	mgrid.M	0.59	0.89	50%	0.00	0.01	20.28	30.15	3.45	5.88	70%
	swim.M	0.17	0.47	175%	0.00	0.00	61.29	107.42	1.04	3.27	215%
	wupwise.M	1.63	0.47	71%	0.00	0.00	5.22	163.32	5.22	4.20	20%
	GeoMean	**0.57**	**0.49**	**17%**	**0.00**	**0.00**	**19.71**	**79.55**	**3.01**	**3.95**	**41%**
	StDev			**62%**							**70%**

A.3 NAS-NPB Results

Table A.5: NAS-NPB results for the Core 2 Duo machine and SiNUCA.

		Average IPC			Branch Miss		L1 DATA MPKI		Speedup		
		Real	Sim	Diff	Real	Sim	Real	Sim	Real	Sim	Diff
1 Thread	bt.A	1.63	1.03	37%	0.00	0.02	28.46	27.96	1.00	1.00	0%
	cg.A	1.57	0.84	46%	0.01	0.01	121.01	129.95	1.00	1.00	0%
	ft.A	1.53	1.25	18%	0.00	0.03	148.41	87.31	1.00	1.00	0%
	is.A	0.70	1.58	127%	0.00	0.00	28.22	20.52	1.00	1.00	0%
	lu.A	0.90	0.65	28%	0.00	0.01	55.66	55.00	1.00	1.00	0%
	mg.A	1.41	0.89	37%	0.00	0.01	35.74	20.28	1.00	1.00	0%
	sp.A	1.24	0.81	35%	0.00	0.01	46.79	36.67	1.00	1.00	0%
	GeoMean	**1.23**	**0.97**	**39%**	**0.00**	**0.01**	**54.02**	**42.87**	**1.00**	**1.00**	**0%**
	StDev			**36%**							**0%**
2 Threads	bt.A	1.44	0.98	32%	0.00	0.02	30.54	55.69	1.76	1.90	8%
	cg.A	1.14	0.48	58%	0.01	0.01	125.92	227.65	1.44	1.13	21%
	ft.A	1.21	1.15	5%	0.00	0.03	147.86	165.43	1.59	1.84	16%
	is.A	0.65	1.27	96%	0.00	0.00	30.55	37.33	1.89	1.61	15%
	lu.A	0.67	0.51	24%	0.00	0.01	58.02	102.25	1.48	1.49	1%
	mg.A	0.92	0.60	35%	0.00	0.01	64.41	41.70	1.33	1.36	2%
	sp.A	0.82	0.61	26%	0.00	0.01	57.51	71.88	1.33	1.52	14%
	GeoMean	**0.94**	**0.75**	**30%**	**0.00**	**0.01**	**62.52**	**81.55**	**1.53**	**1.53**	**7%**
	StDev			**29%**							**8%**

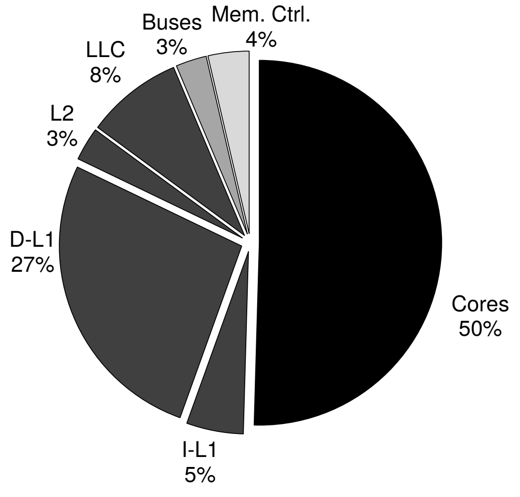
Table A.6: NAS-NPB results for the Sandy Bridge machine and SiNUCA.

		Average IPC			Branch Miss		L1 DATA MPKI		Speedup		
		Real	Sim	Diff	Real	Sim	Real	Sim	Real	Sim	Diff
1 Thread	bt.A	2.16	1.27	41%	0.00	0.02	28.81	35.20	1.00	1.00	0%
	cg.A	1.89	1.54	19%	0.01	0.01	71.23	216.59	1.00	1.00	0%
	ft.A	2.33	1.96	16%	0.00	0.03	54.02	153.26	1.00	1.00	0%
	is.A	0.82	1.94	135%	0.00	0.00	35.87	22.76	1.00	1.00	0%
	lu.A	1.69	1.57	7%	0.01	0.01	33.83	55.69	1.00	1.00	0%
	mg.A	2.29	1.40	39%	0.00	0.01	28.08	29.53	1.00	1.00	0%
	sp.A	1.76	1.45	18%	0.00	0.01	43.57	58.96	1.00	1.00	0%
	GeoMean	**1.77**	**1.57**	**26%**	**0.00**	**0.01**	**40.03**	**59.30**	**1.00**	**1.00**	**0%**
	StDev			**44%**							**0%**
2 Threads	bt.A	2.11	1.26	40%	0.00	0.02	28.83	35.08	1.95	1.98	2%
	cg.A	2.11	1.45	31%	0.01	0.01	70.94	216.31	2.22	1.89	15%
	ft.A	2.13	1.93	9%	0.00	0.03	53.84	153.17	1.82	1.97	8%
	is.A	0.79	1.62	105%	0.00	0.00	36.90	26.42	1.92	1.67	13%
	lu.A	1.63	1.47	9%	0.01	0.01	33.77	53.77	1.93	1.80	6%
	mg.A	1.97	1.30	34%	0.00	0.01	27.92	31.28	1.71	1.86	8%
	sp.A	1.60	1.33	17%	0.00	0.01	43.57	59.57	1.81	1.83	1%
	GeoMean	**1.68**	**1.47**	**25%**	**0.00**	**0.01**	**40.11**	**60.81**	**1.90**	**1.86**	**6%**
	StDev			**33%**							**5%**
4 Threads	bt.A	1.89	1.20	37%	0.00	0.02	27.92	35.11	3.49	3.77	8%
	cg.A	1.78	1.11	38%	0.01	0.01	70.49	217.95	3.74	2.89	23%
	ft.A	1.82	1.78	2%	0.00	0.03	53.58	153.61	3.11	3.63	17%
	is.A	0.77	1.64	113%	0.00	0.00	36.78	24.36	3.73	3.38	9%
	lu.A	1.47	1.28	13%	0.01	0.00	32.57	51.14	3.44	2.96	14%
	mg.A	1.17	1.24	6%	0.00	0.01	27.54	34.27	2.04	3.54	74%
	sp.A	1.16	1.21	4%	0.00	0.01	42.18	59.87	2.60	3.33	28%
	GeoMean	**1.38**	**1.33**	**14%**	**0.00**	**0.01**	**39.37**	**60.61**	**3.10**	**3.34**	**19%**
	StDev			**39%**							**23%**
8 Threads	bt.A	1.59	1.00	37%	0.00	0.02	27.85	35.08	5.84	6.28	7%
	cg.A	1.08	0.78	28%	0.01	0.01	69.62	218.73	4.48	4.05	10%
	ft.A	1.29	1.37	6%	0.00	0.03	53.01	154.30	4.41	5.60	27%
	is.A	0.63	1.36	116%	0.00	0.00	36.07	25.40	6.07	5.61	8%
	lu.A	1.24	1.13	9%	0.00	0.00	32.23	45.74	5.79	4.68	19%
	mg.A	0.51	0.83	62%	0.00	0.01	26.71	49.54	1.77	4.74	167%
	sp.A	0.71	0.93	31%	0.00	0.01	41.69	60.90	3.15	5.13	63%
	GeoMean	**0.94**	**1.03**	**28%**	**0.00**	**0.00**	**38.83**	**63.47**	**4.19**	**5.11**	**23%**
	StDev			**38%**							**58%**

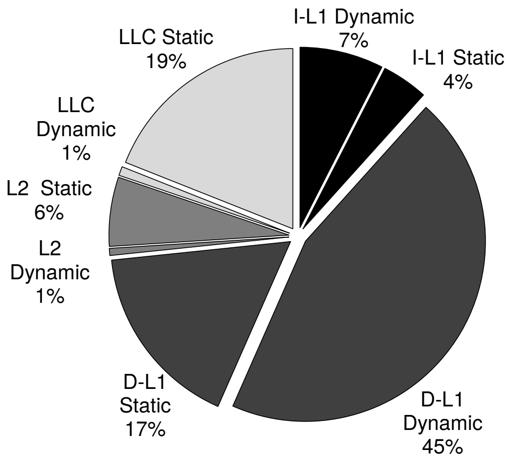
APPENDIX B RESUMO EXPANDIDO EM PORTUGUÊS

O projeto de processadores energeticamente eficientes, começa com um design eficiente de todos os componentes que possuem alto consumo de energia. Embora as memórias *cache* sejam fundamentais para se atingir alto desempenho computacional, estas memórias consomem uma significativa fração da energia total do *chip* (LI et al., 2009).

Figure B.1: Divisão do consumo de energia para o processador Sandy Bridge, executando os conjuntos de aplicações SPEC-CPU2006, SPEC-OMP2001 e NAS-NPB.



(a) Consumo total de energia no *chip*.



(b) Consumo de energia da memória cache.

A Figura B.1 mostra o consumo de energia para o processador Intel Xeon (microarquitetura Sandy Bridge (YUFFE et al., 2011)). O consumo médio de potência corresponde a 37 W. Podemos observar que 43% do consumo de energia dentro do *chip* é gasta pelas memórias *cache*, enquanto cada núcleo de processamento (8 núcleos ao total) é responsável por 6% da energia gasta pelo *chip* (50% no total).

Devido ao crescimento no consumo energético das memórias *cache* e o limite máximo de consumo de potência pelo *chip*, mecanismos para melhorar a eficiência destas memórias estão se tornando mais importantes. Dentro do subsistema de memória *cache*, as principais fontes de consumo de energia estão relacionadas com a energia dinâmica e estática da *cache* L1, e a energia estática da Last-Level Cache (LLC), o que pode ser explicado pelo número elevado de operações na *cache* L1 e pela grande área ocupada pela *LLC*.

Para os processadores atuais com um tamanho de linha de *cache* fixo, a ineficiência energética pode ocorrer em dois níveis: 1) no nível da linha de *cache*, onde uma linha é mantida ligada (ativa) por muito mais tempo do que o necessário, e 2) no nível de sub-bloco, quando partes de uma linha de *cache* que não serão usadas são trazidas para a *cache*, e também quando sub-blocos vivos (isto é, ainda serão acessados) tornam-se mortos (ou seja, não serão mais acessados) depois de alguns acessos, mas são mantidos ativos até a linha ser removida da memória *cache*.

Além do impacto no consumo de energia, ao manter linhas que não serão mais usadas (linhas mortas) dentro da *cache*, aumenta-se a poluição da *cache* e a contenção no controlador de memória. A poluição da *cache* aumenta quando a política de substituição toma decisões erradas, removendo linhas vivas em vez de linhas já mortas. Esta poluição também pode aumentar o número de faltas de dados na *cache*, gerando assim impacto negativo sobre o desempenho do sistema. O problema de contenção no controlador de memória acontece quando a *cache* mantém linhas modificadas que já receberam a última escrita (não serão mais modificadas). Ao fazer isso, estas linhas sofrerão *write-back* para o próximo nível de memória apenas quando a linha for removida da *cache*. No entanto, considerando que os acessos a memória ocorrerem em rajadas (WANG; KHAN; JIMÉNEZ, 2012a), as operações de *write-back* podem aumentar a pressão na memória nos momentos em que um grande conjunto de dados está sendo solicitados em um curto espaço de tempo (rajadas de acessos).

O objetivo principal desta tese é introduzir mecanismos que possibilitem o aumento da eficiência energética das memórias *cache*. Este objetivo será alcançado através dos seguintes passos:

- Propomos o Dead Sub-Block Predictor (DSBP) para prever em tempo de execução a utilização da linha de *cache* em granularidade de sub-blocos. Este mecanismo será usado para armazenar apenas o sub-blocos úteis dentro da linha de *cache*, desligando os sub-blocos ao se tornarem mortos.

- Propomos o Dead Line and Early Write-Back Predictor (DEWP) para detectar quando uma linha de *cache* recebe a sua última leitura e última escrita. Este preditor será utilizado para adiantar o *write-back* de linhas modificadas para assim que elas recebem sua última escrita, desligando também as linhas assim que receberem sua última leitura.

- Para avaliar os novos mecanismos propostos, desenvolvemos o Simulator of Non-Uniform Cache Architectures (SiNUCA), um novo simulador com precisão de ciclos, orientado a traço de execução, composto pelos principais componentes: processador, memórias *cache*, interconexões e sistema de memória. Este simulador é capaz de simular sistemas *multi-core*, com *multi-banked cache* e interconexões Network-on-Chip (NoC). Este simulador é validado comparando com uma máquina real em termos de desempenho e consumo de energia.

O objetivo geral desta tese é a concepção de mecanismos que permitam a economia de energia na memória *cache*, mantendo o desempenho do sistema no mesmo nível. Os custos adicionais de tais mecanismos também serão avaliados, a fim de mostrar seus benefícios e possíveis desvantagens.

B.1 Simulator of Non-Uniform Cache Architecture (SiNUCA)

O simulador SiNUCA foi desenvolvido e validado em termos de desempenho e energia. Alimentado por traços de execução e com precisão de ciclos, o simulador é capaz de executar aplicações *single-threaded* e *multi-threaded* com alto nível de detalhes de todos os componentes do *pipeline*. Além disso, foi proposto um grande conjunto de *microbenchmarks* a fim de correlacionar os resultados do simulador (desempenho e outras estatísticas), com dados reais de duas plataformas x86.

O SiNUCA tem as seguintes características principais:

Alta precisão: O SiNUCA implementa componentes de arquitetura com um elevado grau precisão, não apenas no *pipeline* de execução, mas também na hierarquia de memória e interconexão. O simulador também modela com precisão arquiteturas paralelas, tais como sistemas *multi-core* e sistemas multi-processados. A implementação foi feita utilizando informações disponíveis publicamente. Para as informações não disponíveis, *microbenchmarks* foram utilizados para observar o comportamento dos componentes de uma máquina real.

O SiNUCA foi validado com aplicações *single-threaded* e *multi-threaded* de nossos *microbenchmarks* e também de cargas de trabalho comerciais. Os resultados da simulação foram comparados com as estatísticas obtidas de máquinas reais.

Modelo energético: O consumo de energia está se tornando cada vez mais importante para arquiteturas de processadores. No entanto, a maioria dos simuladores atuais não modelam o consumo de energia, apenas o desempenho. Para avaliar o consumo de energia, integramos a ferramenta Multi-core Power, Area, and Timing (McPAT) (LI et al., 2009, 2013), que utiliza as estatísticas geradas pelo SiNUCA. Estes resultados foram validados utilizando contadores de hardware de energia existentes na máquina real.

Suporte a técnicas emergentes: O SiNUCA é capaz de modelar várias tecnologias de ponta, como Non-Uniform Cache Architecture (NUCA), Non-Uniform Memory Access (NUMA), NoC e controladores de memória Double Data Rate (DDR) 3. Além das técnicas tradicionais, tais como *prefetchers* de *cache*, preditores de desvio e outros. O suporte para novas tecnologias é importante para a simulação de novo sistemas com precisão.

Flexibilidade: Outra característica importante para apoiar a pesquisa em arquitetura de computadores é a facilidade de implementação ou extensão de funcionalidades no simulador. Este aspecto é fornecido pelo SiNUCA com uma arquitetura modular, escrita em C++, que fornece um acesso direto aos detalhes de funcionamento de todos os componentes simulados. Outros simuladores são limitados por metalinguagens que não expõem todas as funcionalidades da microarquitetura, tornando-se mais difícil para modelar novos mecanismos ou modificar os já existentes.

A validação do simulador mostrou uma diferença média no desempenho de 12% usando os *microbenchmarks single-threaded* e 26% para os *multi-threaded* ao simular uma máquina Core 2 Duo. Uma diferença média no desempenho de 6% para *single-threaded* e 29% para cargas de trabalho *multi-threaded*, quando simulando uma arquitetura Sandy Bridge.

Considerando a modelagem de energia usando McPAT utilizando as estatísticas de execução fornecidas pelo SiNUCA, obtemos uma diferença média de 18% para os *microbenchmarks single-threaded* e 46% para *multi-threaded* ao modelar a arquitetura Sandy Bridge.

Os resultados referentes ao desempenho do simulador mostram que o SiNUCA permite arquitetos de computador avaliarem novas técnicas em um tempo razoável, simulando em média 270 kHz, equivalente a 250 Kilo Instructions per Second (KIPS).

B.2 Dead Sub-Block Predictor (DSBP)

Propomos o DSBP (ALVES et al., 2012) para melhorar a eficiência energética das memórias cache. O DSBP usa o histórico recente para prever quais sub-blocos serão úteis e quantos acessos cada sub-bloco receberá antes que se torne morto. O principal objetivo do DSBP é reduzir o consumo de energia estática e dinâmica, trazendo apenas os sub-blocos úteis para o cache, e também desligando os sub-blocos ativo após o número previsto de acessos. O DSBP também é usado para melhorar a política de substituição de cache existente, priorizando linhas mortas (isto é, quando todos os sub-blocos estão desligados) para serem removidas. Os resultados mostram que esta política compensa efetivamente a falta de dados adicional que o DSBP pode causar durante previsões errôneas de uso de uma linha de cache.

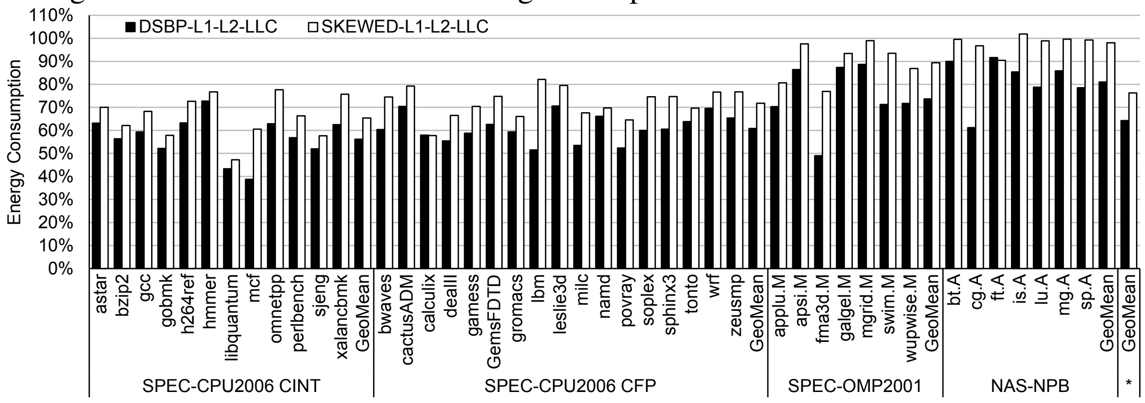
As principais contribuições do mecanismo DSBP são:

Preditor de uso de sub-blocos: Um mecanismo é apresentado para prever e alocar apenas os sub-blocos úteis de cada linha de cache. Ao contrário de trabalhos anteriores, que requerem um acesso a tabela de previsão após cada acesso a linha de cache, o acesso a estrutura preditora é feita apenas quando o mecanismo está aprendendo um novo padrão ou quando uma nova linha de dados está sendo trazida para dentro da cache. Em média, o nosso mecanismo acessa sua estrutura interna em apenas 60% dos acessos a memória cache.

Preditor de sub-blocos mortos: Nosso mecanismo também prevê quando cada sub-bloco dentro de uma linha de cache se tornará morto. Para o nosso conhecimento, este é o primeiro preditor de uso que atua em um nível de sub-blocos, desligando sub-blocos mortos e economizando em média 36% de energia, em comparação com uma cache tradicional.

Remoção adiantada: Nosso mecanismo melhora o algoritmo de substituição linhas da cache. O preditor de sub-blocos fornece informações para o algoritmo de substituição, marcando linhas mortas como possíveis vítimas para serem removidas.

Figure B.2: DSBP: Consumo de energia total para o subsistema de memória cache.



Nosso mecanismo que prevê quais sub-blocos serão úteis e quando cada sub-bloco se tornará morto aumenta a eficiência energética das memórias cache. A Figura B.2 apresenta o consumo total de energia para cada preditor avaliado. Os resultados são mostrados aplicando cada mecanismo em todos os níveis de memória cache. O DSBP atinge uma economia média de energia de 36% para todos os níveis de memória cache, superando em 27% os melhores resultados do trabalho correlato SKEWED.

Em geral, o DSBP requer um baixo tamanho de armazenamento para alcançar previsões precisas (61% previsões corretas e 10% dos previsões subestimadas). O DSBP consegue uma redução média no consumo de energia de 36%, em comparação com a arquitetura base. O tempo de execução foi aumentado em 2,25% em média para aplicações single-threaded e multi-threaded. O DSBP alcança 64% do potencial de economia que um mecanismo perfeito (oráculo) é capaz de alcançar.

B.3 Dead Line and Early Write-Back Predictor (DEWP)

O mecanismo proposto DEWP (ALVES et al., 2013) é composto por um preditor de última leitura/escrita que trabalha na granularidade de linha de cache. O preditor de última leitura que visa economizar energia desligando linhas de cache mortas ou inválidas. O preditor de última escrita que visa adiantar as operações de *write-back* em linhas de cache modificadas, uma vez que estas linhas não serão mais modificadas. Utilizando ambos preditores de última leitura e última escrita o mecanismo é capaz de detectar quando uma linha recebe o seu último acesso, priorizando essas linhas para serem removidas da cache.

O preditor de última leitura usa o histórico de acessos para prever quando uma linha de cache se tornará morta, podendo assim ser desligada. A linha de cache é considerada *morta* sempre que ela receber a sua última leitura antes de ser removida ou invalidada.

A previsão de última escrita permite adiantar o *write-back* das linhas de cache modificadas, reduzindo a pressão sobre o controlador de memória entre leituras e escritas durante rajadas de acessos. Além disso, a realização adiantada do *write-back* também permite que essas linhas possam ser desligadas quando a última leitura for prevista.

Ambos preditores reduzem a poluição da cache, priorizando a remoção de linhas mortas. Todas as linhas de cache que normalmente seriam expulsas da memória cache pela política de substituição são considerados mortas desde seu último acesso. Ao expulsar estas linhas, outras linhas de cache que ainda estão vivas podem permanecer por mais tempo dentro da memória cache.

As principais contribuições do mecanismo DEWP são:

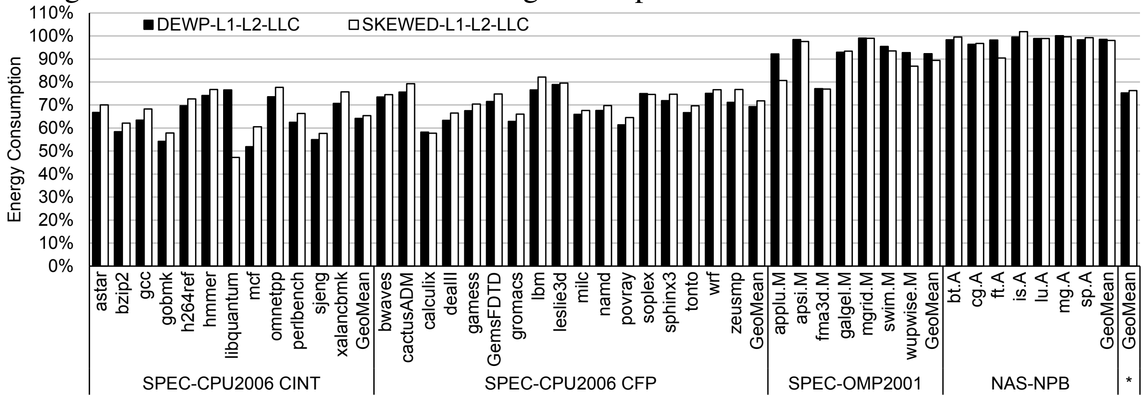
Preditor de última leitura: Capaz de desligar linhas de cache depois de receberem a última leitura antes que sejam removidas da cache. Isto se traduz em 25% de economia de energia da cache.

Preditor de última escrita: Nosso mecanismo pode adiantar as operações de *write-back* de linhas modificadas depois de receberem a última escrita. Aumentando a janela de tempo para o *write-back* das linhas de cache, reduzindo assim a pressão no controlador de memória.

Preditor de último acesso: Combinando os resultados das duas previsões, o nosso mecanismo detecta o último acesso para cada linha, priorizando as linhas mortas para serem removidas da cache, melhorando assim a utilização da cache. O preditor atinge 73% de previsões de corretas com 14 % de falsos positivos em média para toda os níveis de cache.

A eficiência energética é aumentada usando nosso mecanismo que desliga linhas de cache mortas ou invalidas. Os resultados da Figura B.3 apresenta o consumo total de energia para cada preditor avaliado. Os resultados são obtidos ao aplicar cada mecanismo em todos os níveis de memória cache. O DEWP atinge uma economia média de energia

Figure B.3: DEWP: Consumo de energia total para o subsistema de memória cache.



de 25% para todos os níveis de memória cache, superando em 4% os melhores resultados do trabalho correlato SKEWED.

Em geral, o DEWP requer pequenas tabelas para alcançar previsões precisas (73% previsões corretas e 14% de previsões subestimadas). O DEWP consegue atingir uma economia média de energia de 25%, em comparação com a arquitetura base. O tempo de execução foi aumentado em 3,75% em média para aplicações *single-threaded* e *multi-threaded*. O DEWP alcança próximo de 100% do potencial de economia que um mecanismo perfeito (oráculo) é capaz de alcançar.

B.4 Combinando os mecanismos DSBP e DEWP

Os resultados combinando as duas propostas que visam reduzir o consumo de energia através da previsão do padrão de uso da linha de cache, mostrou que os ganhos máximos podem ser obtidos ao aplicar os dois mecanismos acoplados em um sistema.

Combinando o DSBP nos níveis L1 e L2 e o DEWP na LLC pode-se superar em termos de economia de energia o DSBP ou o DEWP separadamente por 1% e 12% (pontos percentuais), respectivamente. Além disso, esta combinação gera um menor impacto sobre o tempo de execução, com acréscimo médio de 1,77%, o que representa um sobrecusto menor do que o DSBP (2,25%) e o DEWP (3,65%).

B.5 Conclusões

O consumo de energia tem se tornado um fator importante em projetos *multi-core*. Assim, mecanismos que permitam a economia de energia, mantendo o desempenho em um mesmo nível são essenciais para manter o consumo de energia e uma operação eficiente. Considerando que as memórias cache consomem uma grande parcela de energia dentro dos *chips*, esta tese apresenta dois mecanismos, o DSBP e o DEWP, capazes de aumentar a eficiência energética das memórias cache.

O mecanismo DSBP trabalhando em granularidade de sub-blocos de dados atinge 36% de economia de energia em média, o que equivale a 15% do consumo total do *chip*. Em granularidade de linha de memória cache, o mecanismo DEWP atinge 25% de economia de energia em média, o que equivale a 10% do consumo total do *chip*. A implementação mista fornece os melhores ganhos energéticos, economizando 37% em média, o que equivale a 16% do consumo total do *chip*.