UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMATICA

Departamento de Arquitectura de Computadoras y Automática

GESTIÓN DE LOS ACCESOS A MEMORIA EN SISTEMAS DE ALTAS PRESTACIONES. MEMORY ACCESSS MANAGEMENT IN HIGH PERFORMANCE COMPUTERS

MEMORIA PARA OPTAR AL GRADO DE DOCTOR
PRESENTADA POR

Rodrigo González Alberquilla

Bajo la dirección de los doctores

Francisco Tirado Fernández
Luis Piñuel Moreno

Madrid, 2013

Gestión de los Accesos a Memoria en Sistemas de Altas Prestaciones

Memory Access Management in High Performance Computers

Programa de Doctorado en Ingeniería Informática

TESIS DOCTORAL

Rodrigo González Alberquilla

Facultad de Informática
Universidad Complutense de Madrid

Madrid, Enero de 2013

Gestión de los Accesos a Memoria en Sistemas de Altas Prestaciones.

Memoria presentada por Rodrigo González Alberquilla para optar al grado de Doctor por la Universidad Complutense de Madrid, realizada bajo la dirección del Prof. D. Luis Piñuel Moreno y del Prof. D. Francisco Tirado Fernández.

Facultad de Informática
Universidad Complutense de Madrid

Madrid, Enero de 2013.

Memory Access Management in High Performance Computers.

A dissertation submitted by Rodrigo González Alberquilla to the Computer Science School of the Universidad Complutense de Madrid in partial fulfillment of the requirements for the degree of Doctor of Philosophy, done under the supervision of Prof. Luis Piñuel Moreno and Prof. Francisco Tirado Fernández.

Computer Science School
Universidad Complutense de Madrid

Madrid, January 2013.

Agradecimientos

Esta tesis es la culminación de cuatro años de trabajo, que no habría sido posible sin la guía de mis directores Paco y Luis, a los cuales quiero agradecer todos sus esfuerzos por que este trabajo llegase a buen término.

Tampoco habría sido posible sin el apoyo de mis padres, sin su paciencia y comprensión, ni sin mi hermano, que siempre han estado ahí para cualquier cosa que pudiera necesitar. Y ya que estoy con la familia, al primo Álvaro, por ser quien, de manera voluntaria o no, despertó en mi el interés por la computación, y siempre apostó por mi.

A la memoria de Ángela, porque siempre fuiste una segunda madre para mi, y en todo lo que hago, te llevo en el pensamiento.

Quiero agraceder el apoyo prestado a dos de las personas más importantes en mi vida, Estela y Dieguito, que en el día a día me han apoyado, ayudándome a desconectar, y haciendome disfrutar del mundo que hay más allá del laboratorio.

A Carolina, Edgardo, Fermín, Hugo, Rubens (y CrisPiso) y Seto, por formar equipo y compartir penurias y alegrias, y esos miércoles de parrondo, que siempre unen, sin olvidarme de Juan Carlos y Juan Carlos, porque no puedo pensar en dos ejemplos mejores con quien compartir despacho (LPA).

A todos los que son o han sido administradores de sitemas del grupo ArTeCs, porque sin maquinas a punto, el trabajo no saldría adelante, muy en especial a Tabas, por ese servicio personalizado a horas intempestivas.

Me gustaría también mencionar a Luis Ceze y Karin Strauss, por la acogida que me dieron en las estancias en Seattle, y a todo el grupo Sampa de la University of Washington por la calida acogida, en especial a Nick Hunt, por la compañía y las labores de administración realizadas. También a Robin, porque gracias a él, lo que comenzó siendo una estancia se convirtió en la construcción de otro hogar.

Al Bat Clan y a la periferia, tanto de Kenia como de Arwels, y a todos esos amigos que han sabido escucharme, hablarme y sacarme de fiesta en mejores y peores épocas, y que nombrar a unos sería injusto para otros y no hay sitio para todos.

Son demasiadas personas las que me han ayudado a seguir adelante, a centrarme y a darlo todo, y necesitaría un capitulo entero, así que espero que las personas no citadas sepan entenderme y perdonarme.

Para terminar, a Amaya, a Ana, y entregando la manzana de la discordia, en especial a Sarita, por todas las experiencias compartidas y enseñanzas impartidas.

Abstract

The memory subsystem is one of the three main components of a computer, along with the CPU and the I/O. As such, it is determinant in the performance, power consumption, robustness and any aspect of the system. This subsystem is commonly implemented in a hierarchical fashion, and the trend in the last few years is including more and more levels in this hierarchy.

The memory system is always a hot topic of research, having many efforts spread among power efficiency, performance, robustness/debuggability, and emerging memory technologies. The former two are typical issues to address, techniques use fast and/or low energy small memories to store the data that will be accessed in the near future. Robustness takes care of soft errors and security. Debuggability consists on providing help to debug hard-to-track synchronization bugs by providing feedback about the proper/not proper usage of synchronization primitives. Finally, a lot of effort is being currently invested in making a reliable and durable memory system based on emerging main memory technologies.

In this thesis, we propose three different techniques tackling three of the aforementioned issues. The first technique tackles the problem of power consumption. It consists of a semantic filter to capture the accesses to the stack in a small, direct-mapped structure to reduce power consumption in embedded systems. The second technique addresses debuggability. In that piece of work we propose an extension to the ISA and an algorithm using that extension to detect data races with low overhead. The last part of this thesis concerns reliability/durability of emerging memory technologies. In particular PCM-based memory systems. It approaches the problem from the perspective of compression to both reduce the amount of data written to the memory, thus reducing the wear, and to give a means to encode information about errors in a memory block.

Contents

Chapter 1

Introduction

In computing systems, the memory subsystem is a key component in the system. Any change in its characteristics is bound to impact the whole system. For this reason, both academia and industry have always invested a lot of effort in improving it. At first targeting performance, but gradually broadening the scope to include power efficiency, robustness, debuggability, and scalability in the integration process. In the last few years, endurance for resistive technologies that emerge in order to overcome the limits in scalability offered by the previous technology. This new technologies effectively increment the scalability, but wear and become useless rapidly, requiring techniques to elongate their life cycle.

Performance is a constant topic. A big percentage of the instructions in the dynamic stream involve the memory.Modern processors count with many mechanisms to reduce or hide the long latency of memory operations such as caching techniques, prefetching, Out-of-Order capabilities enhanced by the usage of Load-Store Queues... Still, any gain in performance in the memory subsystem leads to performance gain in the whole system.

Power has become a major design constraint in the last decade. The problem with power is twofold. On the one hand, it is energy consumption itself. Portable devices are widespread, meaning that a big part of the computing systems are powered by batteries. Reducing the power consumption is critical to elongate the autonomy of the system, and to reduce the weight of the battery, and of the whole system. Two appealing features that all vendors want for their systems. In the case of plugged computers, power is important, more power means higher power bill. On the other hand, there is the problem of density of power: even if the power a computer consumes can be afforded, the heat needs to be dissipated. Long ago, systems reached what is called "the power wall" [1], a limit in the amount of power a chip can consume. There is this much power per area unit a computer can generate not to overheat. This two problems have been addressed by many authors.

Robustness is also a constant concern. Process variation, radiation and other factors affect the reliability of the memory. Cells can be flipped upon the impact of cosmic rays or under the effect of radiation. Techniques such as ECC [2] have become popular and are a feature offered by servers.

Debuggability is desirable feature of computer systems, and the memory system can help in many ways. In a sequential program, the order of memory accesses just depends on the program and the OS. In a multithreaded program, this order also depends on the state of all the processors sharing the memory and the state of the memory itself. The probability of some interleaving not being taken into account by the programmer is high. It is not only that in multithreaded applications is easier to introduce bugs in the form of data races or atomicity violation, but also, these synchronization bugs are harder to track down than sequential bugs, because there is an implicit non-determinism in the execution that complicates the reproduction of the conditions under which the bug is triggered. The aim is to develop tools that analyze the accesses and detect the errors in the programmers reasoning. Although the memory should appear to different processors as a system that processes read/write requests, and they should not care about the underlying organization, cache memories are capable of doing low-level tracking of accesses to help detect and debug/survive errors.

Scalability is more a technological parameter, but also a feature of the memory. The technology more widespread for main memory currently, DRAM technology, has reached a limit in scalability. It doesn't scale down well beyond 30nm [3]. This has driven researchers to look for new technologies. Some of this technologies are *Phase change memory (PCM) RAM (PRAM)*, *Spin torque transfer RAM (FTT-RAM)* or *Ferroelectric RAM (FRAM)*. While DRAM is a capacitive technology, meaning that logic values are stored as a charge in a condenser, these new technologies are resistive or magnetic technologies. In these technologies, cells are made of a material that can be physically altered, changing its electrical impedance, making it possible to store logical values as different values of impedance. All this technologies scale beyond 16nm, but share common, new issue to address: the fast wearing that the state changes imply. This trade of scalability for endurance renders the technology not ready yet for commercial use, and motivates all the research done in the last few years to improve the endurance of emerging technologies.

In this thesis we propose a variety of techniques to deal with some of the aforementioned issues:

- *Power consumption:* We address power consumption leveraging the information any existing hardware has about extra semantics of memory addresses.
- *Debuggability:* Data races are the focus of our work in this topic. We target a mechanism to inspect the execution of a program *on-line* and, in case a data race report it.
- *Scalability/Endurance:* In the field of PCM, our main concern is making memory blocks usable the longest possible. This requires detecting and surviving as many failures as we are able.

Each issue is addressed from an "advantageous point" in the multilevel hierarchy that comprises the memory subsystem in high performance systems. In the following, we succinctly revisit the concept of memory as a hierarchy.

1.1. Memory subsystem as a multilevel hierarchy

The memory is organized as a multi-level hierarchy. At the bottom are placed the architected registers. At the top is placed the main memory. Since its first appearance in 1963,cache memories have been included in many systems as intermediate levels because of the many advantages they provide.

Each level in the hierarchy is defined by 4 characteristics intrinsic to the technology used: latency, power consumption, scalability and cost (money) per byte. Low levels are closer to the CPU, having a lower latency, but the closer, the more restrictive the constrais in size and latency are, which translates in the level being more expensive either in power, cost or both. The size of each level is a trade-off among cost, power and latency, because cache structures usually have the consumption and latency increased with the size. There are also other functionality aspects, which depend on the design, such as the block size, how main memory blocks map into the cache, how a block is chosen to be evicted when space is required to allocate a new one (replacement policy), inclusion, whether writes are immediately propagated to upper levels or not, and whether blocks are allocated when written to, if not present. A lot of research has been done on this design decisions, and different hardware vendors have different specifications for their products.

This hierarchy is slightly more complex in multicore machines. In such systems, there is, usually, only one main memory shared by all processors. In this scenario caching is also possible, but levels can be private to each core, shared among a number of them, of shared by all of them. If a level is shared by all cores, there is no extra complexity appart from the interconnection, all processors interface with it as if it was the main memory, and there is no further ado. If a level is private to each processors, it is desired for the hierarchy to remain coherent. A hierarchy is coherent if and only if all processors observe the modifications to a given memory location in the same order. Meaning that no caches hold an outdated copy of the data. In other words, in multicore systems, the memory hierarchy has two extra important characteristics: *coherence* and *consistency*. Consistency deals with the relative order of memory operations as observed by processors. Making a memory hierarchy coherent is "easy", but ensuring that all processors in the system observe memory operations in the same order is an overwhelming task. Therefore, there are many consistency models and each system declares which one it implements, so programmers are aware of what to and what not to expect from the system.

In this thesis we only consider the physical memory hierarchy, all the proposed techniques work in a virtual memory hierarchy, but none of them requieres any of the extra concepts or constructs present in a virtual memory hierarchy.

The different levels in the hierarchy suffer for different problems, and offer different possibilities to tackle them. Overall performance and power are common to all levels. Low levels can use the closeness to the CPU to inspect the state: registers, TLB, *etc.* and leverage that information. Higher levels, on the other hand, can take advantage of less restrictive latency constraints and more abstract view of the footprint to adapt the size and/or geometry when less flexibility is required to save

power. Debuggability is an issue not bound to any level in particular. Private levels have some information about sharing in order to guarantee coherence. This information may offer help in analyzing the accesses. Scalability is a problem of main memory. Traditionally, lower levels are made of transistors, so they scale with the technology. On the other hand RAM uses other technologies that don't scale so well. In the following we ellaborate further in which of the issues we tackle at which level.

1.2. Thesis contributions

The memory hierarchy is a vast topic, with many different aspects, all of them worthy of research. To bound the scope of this thesis, we have focused on three of the aforementioned issues, namely, power consumption, programmability/debuggability and scalability/endurance. To address these issues, we have travelled from the bottom to the top of the hierarchy, addressing at each level the problem we think is more interesting to solve or alleviate.

The first part of this thesis regards the gap between the first level cache and the registers, and how to reduce power consumption. The first contribution is the *Stack Filter*: A small and simple structure that leverages the special locality properties exhibit by stack accesses to filter them in a pseudo-level-0 cache. This structure is direct mapped and quite small, making the accesses cheap in terms of power. We show that for embedded system, a small *Stack Filter* achieves non-trivial power savings with a small cost in terms of hardware and small impact on performance.

The second part of this thesis goes a bit higher in the hierarchy, to the shared cache levels, and attempts to help the debugging process of multithreaded programs. The second contribution is minimal hardware support to detect data races, and an algorithm *Accessed Before* that using that support can detect data races on-line. We show its accuracy and we compare it both with academia tools and commercial tools.

Finally, the work in the third part of this thesis is on emerging main memory technologies. In it, we propose some techniques and insights to elongate the life cycle of a PCM-based main memory.

The contributions of this thesis can be summarized as follows:

- The *Stack Filter* is the first semantic-aware caching scheme that takes power as main focus. Also, we prove that a small Filter, such as 32 words provides good results, allowing to place the the filter under the level 1 data cache. This, along with our designed NeCK unit add up to a technique to save data cache power without affecting performance.

- Our algorithm *Access Before* is the first proposal of an on-line data-race detector from a hybrid hardware/software perspective. Using realistic hardware

extensions we are able to speedup data race detection. Our technique proofs faster than commercial tools, both industry tools and opensource tools.

- PCM is not mature enough to be incorporated as main memory technology yet. Our compressing algorithm spans the lifetime of PCM based memory systems.

1.3. Thesis organization

The remainder of this thesis is organized as follows:

- Chapter 2 introduces the Stack Filter, our proposal for reducing the power consumed by the level 1 data cache targetting embedded processors.
- Chapter 3 discusses the problem of data race detection and present our proposal of minimal hardware modifications to speed up data race detection.
- Chapter 4 Gathers a comprehensive study of the potential of compression as means to survive failures through CEPRAM, our technique proposed for extended endurance in PCM memories.
- Chapter 5 Concludes and show the future work.

Chapter 2

Power efficiency: Reducing DL1 power with a Stack Filter

The L1 data cache is one of the most frequently accessed structures in the processor.Because of this and its moderate size it is a major consumer of power. In order to reduce its power consumption, in this chapter a small filter structure that exploits the special features of the references to the stack region is proposed. This filter, which acts as a top –non-inclusive– level of the data memory hierarchy, consists of a register set that keeps the data stored in the neighborhood of the top of the stack. Our simulation results show that using a small *Stack Filter* (SF) of just a few registers, 10% to 25% data cache power savings can be achieved on average, with a negligible performance penalty.

This chapter is organized as follows: Section 2.1 motivates this piece of work. In Sections 2.2 and 2.3 we present potential studies for the idea and the design of our filter to take advantage of that potential. Next, Section 2.4 presents the related work. Section 2.5 presents the experimental environment used for the evaluation shown in Section 2.6, and Section 2.7 concludes.

2.1. Introduction

Continuous technical improvements in the current microprocessors field lead the trend towards more sophisticated chips. Nevertheless, this fact comes at the expense of significant increase in power consumption, and it is well-known for all architects that the main goal in current designs is to simultaneously deliver both high performance and low power consumption. This is why many researchers have focused their efforts on reducing the overall power dissipation. Power dissipation is spread across different structures including caches, register files, the branch predictor, etc. However, on-chip caches can account for 40% of the chip overall power consumption by themselves [4, 5].

One alternative to mitigate this effect is to partition caches into several smaller caches [6, 7, 8] with the implied reduction in both access time and power cost per

access. Another design, known as filter cache [9], trades performance for power consumption by filtering cache references through an unusually small L1 cache. An L2 cache, which is similar in size and structure to a typical L1 cache, comprises the upper level of the filter cache to minimize the performance loss. A different alternative named selective cache ways [10] provides the ability to disable a subset of the ways in a set associative cache during periods of modest cache activity, whereas the full cache will be operational for more cache-intensive periods. Loop caches [11] are other proposal to save power, consisting of a direct-mapped data array and a loop cache controller. The loop cache controller knows precisely whether the next data-requesting instruction will hit in the loop cache, well ahead of time. As a result, there is no performance degradation. Another different approach takes advantage of the special behavior in memory references: we can replace the conventional unified data cache with multiple specialized caches. Each handles different kinds of memory references according to their particular locality characteristics –examples of this approach are [12, 13], both exploiting the locality exhibited in stack accesses.– These alternatives show it is possible to improve caching schemes mainly in terms of performance. It is important to highlight that all of these approaches are just some of the existing proposals in the field of caches design.

In this first part we propose a different approach that also leverages the special features of references to the stack. The novelty resides in the fact that we do not employ a specialized cache for handling accesses to the stack; instead, we use a straightforward and small-sized structure that records a few words in the neighborhood of the stack pointer, and acts like a filter: if the referenced data lies in the range stored in this filter we avoid unnecessary accesses to L1 data cache. Otherwise, we perform the access as in any conventional design. This way, although the IPC remains largely unchanged, we are able to significantly reduce the power consumption of the critical data cache structure with negligible extra hardware. We target a high performance embedded processor as described in [14] as platform to evaluate our proposal, but the technique is likewise applicable to CMPs.

2.2. Stack access behavior

Typical programs employ special private memory regions as stacks to store data temporarily. Our target architecture has some call-preserved registers. After a function call ends these registers must hold the same value as when the call began. To achieve this, a function call will store (or push) those of the call-preserved registers to be written along the routine on top of the stack. Once the function ends the registers will be reloaded (or popped) from the stack, the routine will return and the normal processing shall continue.

In addition to these call-preserved registers, when the compiler doesn't manage to allocate all the local variables in registers, it has to spill some of those variables to memory to make room in the register file for others. The spilling of a variable consists in storing that variable in the stack while it is not needed to be in registers, thus making room for another variable to be allocated in that register. Later on,

when the stored variable is needed again, its value is reloaded from the stack to a register.

These operations render the stack into a structure with an elevated temporal locality. This stack is a software structure with some hardware tools to manage it.

In many architectures there is a register for storing either the address of the next available location in the stack or the address of the last word pushed on top of the stack. This register is commonly referred as stack pointer (*sp*) and identifies the lowest[1] virtual address containing valid stack data. Besides *sp*, the stack may be accessed through either the frame pointer or any other general purpose register.

In this section we present figures come from several experiments we have performed to detect and study the main features of stack references for some MiBench and MediaBench applications (see section 2.5) in our target architecture, which is an ARM based system. First of all, measurements reveal that 34.16% of all executed instructions are memory accesses. Figure 2.1 shows the percentage of load/store instructions for each benchmark. Second, stack references account for 15.58% of all memory accesses, added up across benchmarks. Details of the percentage of stack region references for each benchmark can be found in Figure 2.2. In the studied architecture-compiler pair *sp*-relative addressing is the dominant access method to the stack, accounting for 60.23% of all accesses to the stack. Also very used for stack accesses is $r_{index} + immediate$ where r_{index} is a general purpose register. In Figure 2.3 we show the percentage of memory references that use *sp*[2] as index. Four out of all the benchmarks have the *sp* usage as base register for stack references under 60%, and only 5 benchmarks reference the stack using *sp* in more than 75% of the references.

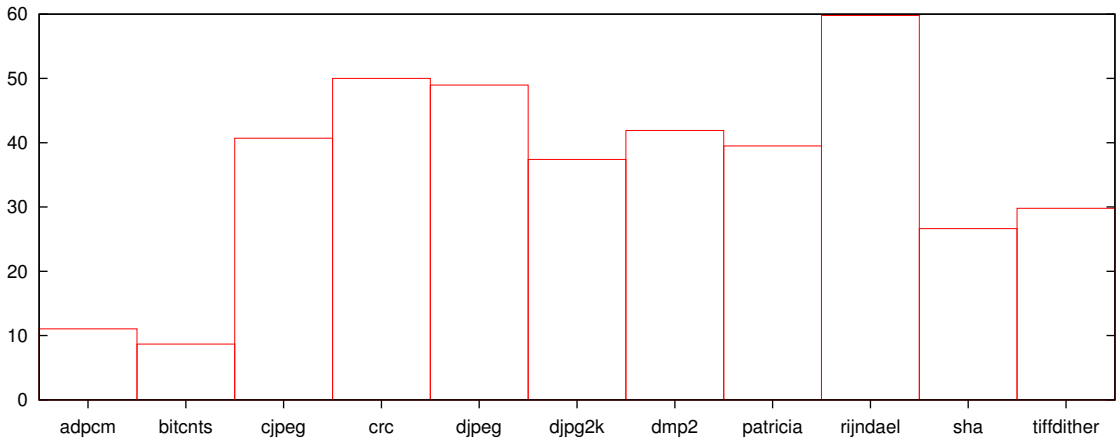


Figure 2.1: Percentage of instructions referencing memory.

A noticeable, and for most applications common, characteristic of stack references is that referenced addresses belong to a contiguous, small and locally stable memory space, as Figure 2.4 illustrates. This figure shows, for *CRC32*, the distance between

[1]In GNU-gcc ARM compiled programs the stack grows downwards.
[2]In ARM, the responsibility of holding the *sp* is assumed by the register r_{13}.

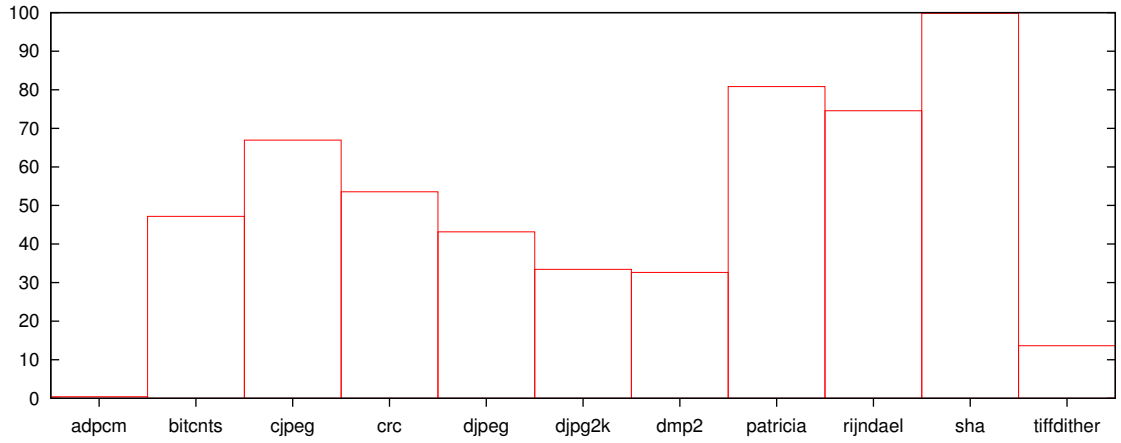


Figure 2.2: Percentage of memory references where the referenced address is in the stack region.

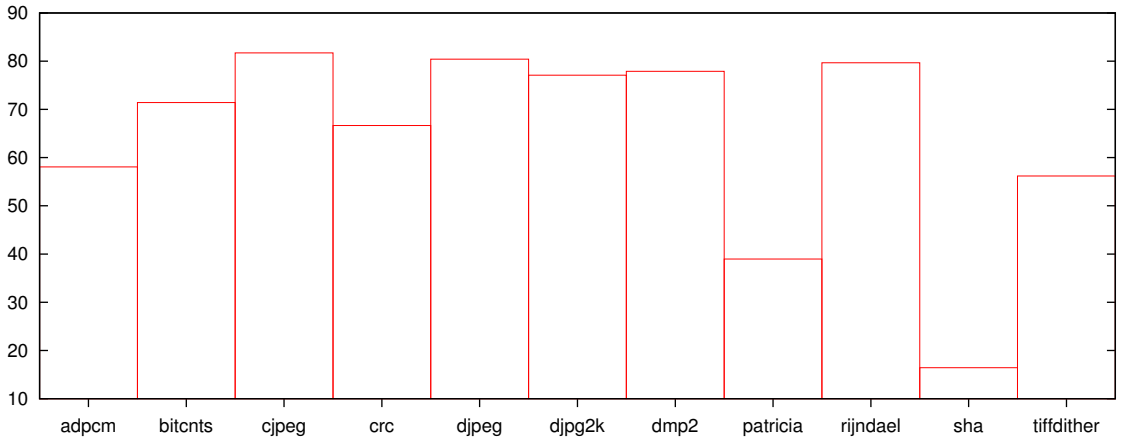


Figure 2.3: Percentage of stack references in which the addressing is done via adding an offset to the *sp* register.

the base of the stack and the *sp* value in each execution cycle. We can observe that, in a representative extract of the execution phase, this difference ranges from 137 to 170 words, meaning that it covers a space of just 33 words. In the initialization and ending phase this range widens, but the length of both phases is negligible when compared with the execution phase. If we study the *sp* value during execution we will observe that the standard deviation is, in most cases (all but *cjpeg* and *stringsearch*), lesser than 50 words. These stack references characteristics lead us to think that stack data can be efficiently recorded in a straightforward and less power hungry structure.

Also, as pointed out in [15], the stack pointer exhibits another significant feature: accesses locality in the neighborhood of the top of the stack (*TOS*). Although the *sp* value is modified at least twice each time the program calls a function upon execution (grow and shrink operations), these changes undo each one another, hence, the *TOS* remains unchanged after them. To illustrate this point Figure 2.5 shows, for *CRC32* application, an histogram which reveals that most accesses are located in the stack pointer proximity. This behavior suggests that a small structure for

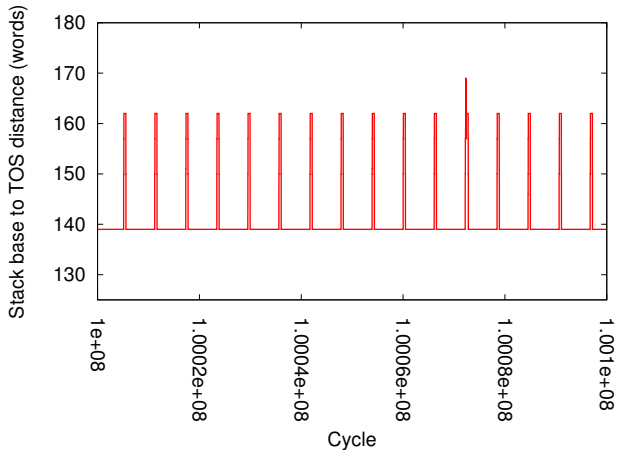


Figure 2.4: Stack behavior. *sp* evolution during execution. *time* and *distance* ranges have been cropped to show the distance in a representative interval from the execution phase, though in the short initialization and ending phasesthis distance is much greater.

keeping some words in the TOS neighborhood is enough to serve the majority of stack accesses.

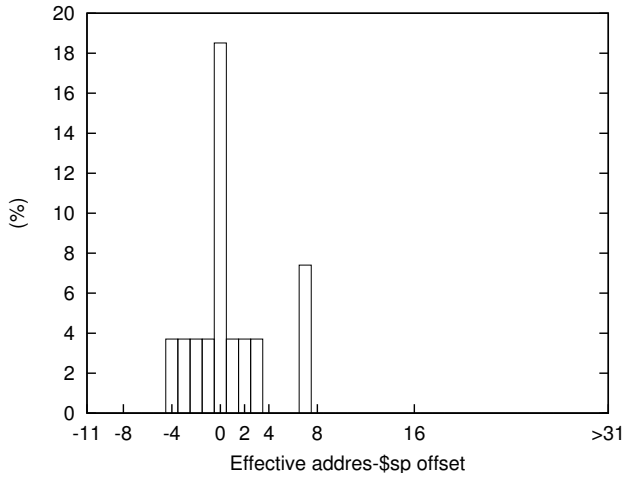


Figure 2.5: Histogram of distances between *sp* and stack addresses.

The applications employed in this study have been compiled with *GNU gcc*. We have compiled applications from the sets MiBench and MediaBench using the standard optimization flags *-O2*, which, in *gcc* implies *-fomit-frame-pointer*. When the flag *-fomit-frame-pointer* is used, the frame pointer is not used. This results in a more efficient memory management by sacrificing debuggability.

GNU gcc follows the *Stack Move Once* policy, no matter the optimization flags used. According to this policy, the stack can grow only once in each routine. This is why in Figure 2.6 stack pointer modifications –for the application *CRC32*– range from ± 4 to ± 16 words. This feature makes unprofitable the idea of speeding up the case of just one word-modifications to the stack –*i.e.* push and pop operations– because one-word modifications don't take place very often in *GNU gcc*-compiled programs.

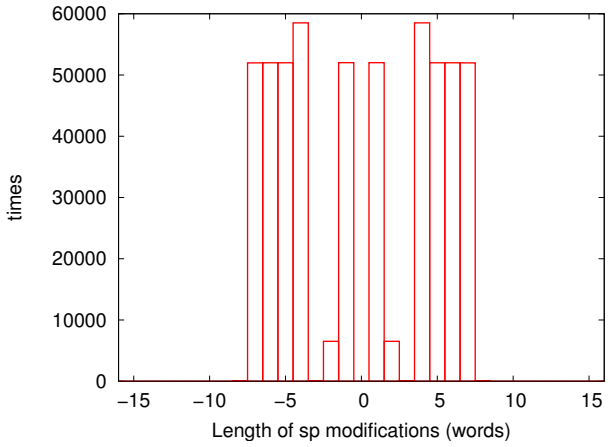


Figure 2.6: Histogram of lenghts in words of *sp* modifications for *CRC32* application.

2.3. Stack Filter Design

Many authors have realized that the locality of stack references can be taken advantage of through caching. Our proposal [14, 16] differs from the previous work on the area in three main goals:

1. We are implementing a very small filter, 8 to 32 words only.
2. Our proposal locates the filter before, in terms of hierarchy, the *dl1* cache. This has to be done under a major constrain, the access to *dl1* cannot be delayed. Therefore, we must avoid any extra penalty in misses. To deal with this constraint, we include a new functional unit in the pipeline, called NeCk –for *Ne*ighborhood *C*hec*k*– unit, to do an early detection of hits and misses, and avoid penalties.
3. Our proposal does not rely in *sp*-based addressing.

We have designed the Stack Filter (SF) using a register file to hold N words in the neighborhood of the TOS. In order to avoid extra memory operations, data is loaded into the filter on-demand, and is written back only if it has been modified. The filter has two status bits per register, "valid" and "dirty", to serve this purpose. As in cache memories, these bits are set if the register holds valid data, and if it has been modified respectively, and are unset otherwise. Memory locations are mapped to the registers according to their distance to the TOS. Changes to the *sp* lead to data-to-TOS distance variation. In order to maintain the mapping from the stack to the filter coherent, when the *sp* is modified by p words, the registers must be moved p positions to compensate the distance variation in the stack. We implement these movements by using the registers as a circular buffer: The register file is provided with a base pointer, r_{base}, which points to the register that holds the TOS. Thus, when the *sp* is modified by p words, the first p words of the filter are evicted, and written back if dirty, and then r_{base} is updated by adding p to its

value. Those p words are taken from the top of the filter if the modification is a contraction ($p < 0$), and from the bottom of the filter if it is an expansion ($p > 0$).

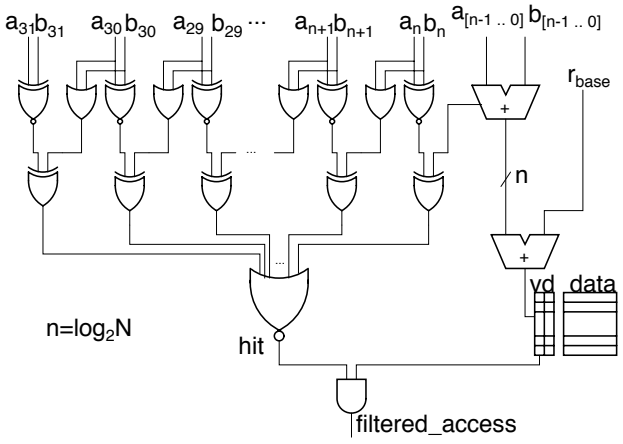


Figure 2.7: NeCk Unit detail: A carry-save adder is used to perform ($\$r_{index} + immediate) - sp$. Its output, (a, b) is then fed to this circuit. $filtered\_access$ will be 1 if the referenced word is in the N-neighborhood and is valid.

Whenever there is a memory operation, the processor must check if the address is near the top of the stack. To do that, meanwhile the ALU calculates the effective address, in parallel, a special purpose functional unit, NeCk unit, calculates whether or not the referenced word lies in the N-neighborhood of the TOS –the N-neighborhood of the TOS are those words separated less than N words from the TOS–. The "valid" bit of the would-be target register is also checked in the same execution phase. If both conditions are satisfied, in the next stage of execution the word would be forwarded from the filter, else a miss happens and the word is retrieved from the cache, and if it lies in the N-neighborhood, it is also copied to the SF. The greatest advantage of doing the offset calculation and validity checking at the same stage than the effective address calculation is that misses incur no penalty (thanks to the early detection).

2.3.1. NeCk: Neighborhood Checking unit

This special unit is responsible of deciding if the instruction will access to the filter or to the data cache. We have designed it to have a low latency, similar to that of a full 32-bit adder, to avoid elongating the critical path. Here we describe how it works

- **Offset (ofs) calculation:** To perform this operation we have added to the pipeline the NeCk unit, depicted in Figure 2.7. This unit calculates the *speaddr* offset, ofs, and checks if the data is in the N-neighborhood of the TOS, while the main pipeline calculates the effective address. This can be done in parallel because *eaddr* calculation is performed twice, once in the main

pipeline and the other in the NeCk unit, thus the result of the main pipeline is not needed to do this step. To be in the N-neighborhood, ofs, must be $0 \leq ofs < N$. As N is a power of two, all but the $\log_2 N$ less significant bits of ofs must be 0. First a $32-bit$ CSA performs $[a, b] = \$r_{index} + immediate - sp$ where $ofs = a + b$.

For $ofs_i = a_i + b_i + c_i$ to be 1, one of the following must happen: (a) there is a carry from the previous stage, and either $a_1 = b_i = 1$ or $a_i = b_i = 0$, (b) there is no carry from the previous stage and $a_i = \bar{b_i}$. In both cases the referenced word would not be in the N-neighborhood of the TOS. We can relax the checks: c_i needs not to be the carry to the i^{th} bit, $c_i = a_{i-1} \, OR \, b_{i-1}$ is enough, because if $a_{i-1} \, OR \, b_{i-1} = 1$ and $c_{out} = 0 \Rightarrow a_{i-1} \, XOR \, b_{i-1} \, XOR \, c_{i-1} = 1$ thus $ofs_{i-1} = 1$ and checking (b) for bit $i - 1$ would fail.

- **Presence determination:** Once ofs is known the presence of the data in the filter must be checked. This is done looking up the "valid" bitmap that tracks the status of the registers. If the operation is a word store, the "valid" bit can be ignored, because the operation will modify the whole register.

2.3.2. Pipeline Modifications

The inclusion of our filter implies some minor modifications to the pipeline. First of all, we must include the NeCk unit and the filter itself. Provided that *sp* modifications will need the filter to be updated, some new hardware is also needed to stop the instruction dispatching and inject in the pipeline the operations required by updates to the *sp*. The handling of modifications to the *sp* is detailed in the next section. Finally, some routing is required to communicate the execution pipeline, the stack filter and the *dl1*.

2.3.3. Stack Filter Management

Once introduced the hardware extensions and modifications, we describe the management of the stack filter: look-ups and updates.

- **Look-up:** First, the NeCk checks whether the memory reference lies in the N-neighborhood. Assuming it does, in the next stage, for a load instruction: if the "valid" bit is set, the target word is forwarded from the filter, and if it is not set, the word is loaded from data cache, copied to the filter, and the "valid" bit is set. For a store instruction, if the "valid" bit is set or the stored data is an aligned word, the word is written in the filter and then "dirty" and "valid" bits are set. If the target word is not valid and would be partially written, the filter first retrieves the word from data cache, and then performs the store in the filter. Both "valid" and "dirty" bits are set afterwards.

- **Updates to *sp*:** When the *sp* is modified the filter must be rotated. This is done in two steps:

1. *sp* **modification detection:** Whenever an operation which would modify the *sp* is dispatched, the *sp* value is backed up and instruction dispatching is stalled. Once the new *sp* value is ready, the filter calculates the amount of words the rotation will consist of, and rotates the filter.
2. **Stack filter rotation:** The filter determines which registers will be evicted, and injects write-back operations in the pipeline depending on the "dirty" bit of each register. Then resets the "valid" and "dirty" bits of the evicted registers. Finally the base register is updated by injecting an special arithmetic instruction. After the store operations are issued and the base has been modified, the instruction dispatching is resumed. We have taken into account these stalls and injection when modelling the filter in the simulator, and are properly accounted for.

Concerning exceptions/syscalls, our target architecture has different *sp* registers and stacks, for the different execution modes. *User* and *System* modes share the same *sp* and stack, and in both the filter is working. When an exception switches to any of the other five modes (*IRQ*, *FIQ*, *Supervisor*, *Undefined* or *Abort*), the *sp* switches, disabling the filter. On the other hand, the stack is a different memory region for every mode. Furthermore, the stacks are pairwise disjoint. Thus there is no chance of incoherence between filter and data cache, so there is no need to write the filter back on exceptions/syscalls.

When there is a context switch in the processor the filter also needs to be flushed as any other cache structure, but care must be taken to flush it before *dl1*, to maintain correctness.

The last issue to deal with is coherence with the rest of the memory hierarchy. At first sight, introducing a new, non-inclusive level to the memory hierarchy may seem prone to coherence errors. But a second thought on this shows that, since we are only filtering cache accesses and the stack region is private to its own thread, as long as this privacy is honored, data on the stack will never be subject to coherence errors. Therefore, coherence issues are dealt with *for free* due to the privacy of the stack. Figure 2.8 depicts how the memory space of an application and each thread of that application is divided into shared data and private data. The stack region and the saved registers (the context), are always in the private, –*i.e.* non-shared– region of each thread. Consequently, the stack is *coherence violation-free*, and no additional mechanism is needed in order to maintain coherence in the filter. Further details about the privacy of stack region can be found in [17].

2.4. Related work

Many researchers have realized that memory stack region exhibits a special locality significantly away from other data regions, and therefore, they have decided to take advantage from this behavior. In early 80's, when the trend in microprocessors design was to increase system's complexity looking for high level languages understanding, Ditzel [18] proposed to remove the register file and to include a stack

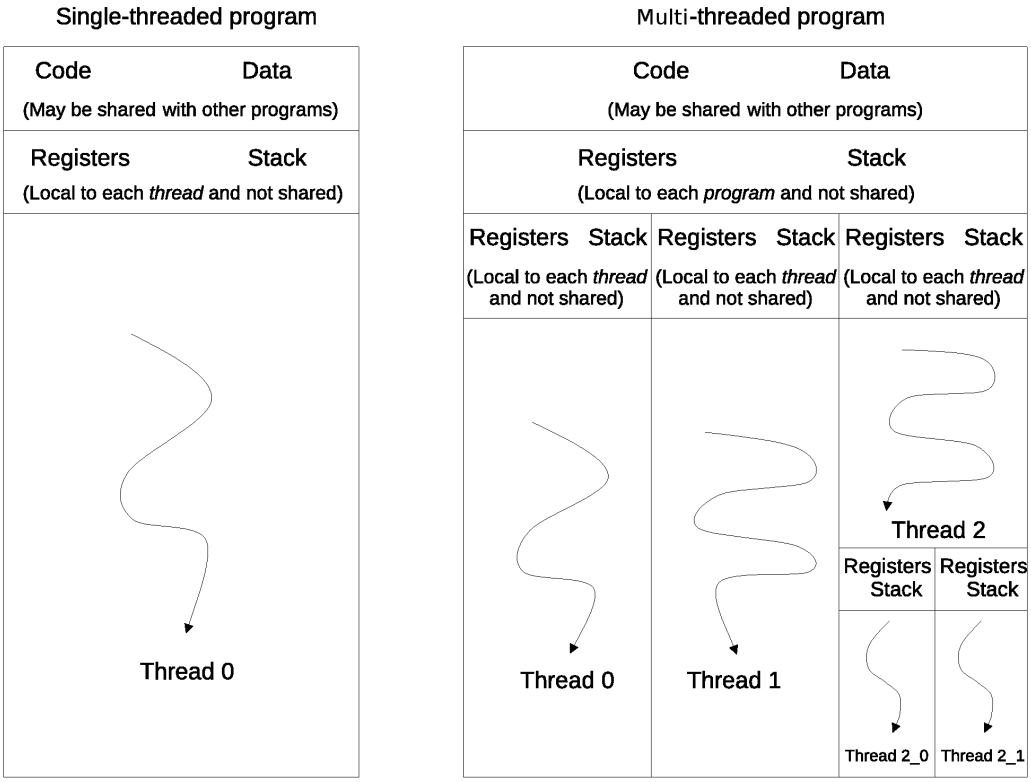


Figure 2.8: Thread Stack in single- and multithreaded programs

cache for favoring memory-memory architectures and avoiding the complex register allocation process to the compiler. Due to the shift in design trends, the mentioned technique would not be easily put into practice in current architectures.

Ward and Halstead [15] propose a TOS cache and discuss its potential benefits, relying on the fact that in architectures with *push* and *pop* operations, pushes are immediately followed by pop operations. Nevertheless it is just a theoretical proposal, without a study of the potential benefit nor a evaluation through an implementation in any real or academic processor. The main limitations of this idea are, firstly, that it requires the presence of push and pop operations in the ISA, and secondly, that it does not take into account that the *sp* register can be modified by instructions other than pops and pushes, therefore, the processor must keep track of the *sp* in order to maintain correctness and distance-to-sp in the contents of the TOS cache.

Lee et al. [12] propose a stack cache (*stack value file*) to record stack values. This cache consists of a register-based structure to store those values close to the top of stack and is located in parallel with conventional *dl1* cache. All references to the stack are morphed into movements from the stack cache instead of accessing *dl1* cache. The mechanism exploits those accesses to the stack via *sp*, penalizing other access methods, which does not occur in our design. Furthermore, our proposal

avoids instruction morphing, simplifying the design. Additionally, Lee's scheme is clearly oriented to increase performance, not to reduce energy consumption in *dl1* cache. Thus, the size of special bank employed by the authors (8KB) is significantly bigger than our small SF (64-256 B). Finally, Lee's work leaves some problems unaddressed, mainly, how data is written back to memory on TOS modifications and how the filter behaves in syscalls and exceptions.

Huang et al. [19] propose a *dl1* design that combines, all in parallel, a few new pseudo-associative caches (PSAC) and a stack cache. In the PSAC, a prediction mechanism selects which way to probe. If the probe hits, the access is satisfied with high speed and low energy consumption. Otherwise, further probing is necessary. The stack cache management implies the usage of two pointers to reduce unnecessary write-backs and line fetches. This approach needs far more storage space than our SF.

Lee and Tyson [20] propose a horizontal partitioning of the first cache level. This is, having different caches to serve accesses to different regions. In their proposal, the level 1 data cache is separated in three different caches, two smaller caches for the stack and global data, and a third bigger cache for the heap and other memory references. Similarly to our approach, this technique takes advantage of the special locality exhibited by stack accesses. The two main differences with our proposal are: first, the level at which this structure is located, level 1 for their approach as opposed to our level 0. Being in level 1 means that misses are served by the next level, therefore, if the cache is not big enough to keep the number of capacity misses low, the execution is slowed down as shown in the following. This problem is aggravated if the design does not count with a second level of cache. The second difference is that accesses are classified by address, so the stack must be located always in the same address range. This can require the re-linkage of binaries to place the stack in the chosen range. This is also a drawback if we target processors capable of executing 32 and 64-bit code, because compilers place the stack in different addresses for each ISA.

Geiger and others [21] extend the aforementioned work by proposing a *dl1* design that use large and small heap caches. Depending on application's behavior "hot" data are kept in the smaller, faster and lower-power cache. This scheme requires the compiler determining what data belongs in what cache through profile feedback. Unlike this approach, our SF works does not require any profiling for a efficient work.

Hemsath and others in a technical report [22] present the results derived from introducing a stack cache the likes of Lee's in a StrongARM SA-110 processor. This work studies the inclusion of a memory cache –the size of *dl1*– too. Furthermore, the authors propose a special structure that dynamically predicts when a replacement in the stack cache or a cache miss will occur in order to schedule the corresponding transfers with the lower levels of memory hierarchy as soon as possible and mitigate this way the long latencies associated. In addition, this study is focused on cache performance, not power efficiency.

These four last proposals locate the stack cache in the same level than 1st level data cache. This fact requires that misses must access to second level cache or

RAM memory. Thus, all three designs are more power hungry than [12] or our proposed technique, in which the extra storage is placed at a 0 level, so misses are served by 1st level cache. Kin and Gupta [9], on the other hand, propose a filter cache, which is small, directly-mapped cache placed before the level 1 data cache. The purpose of this structure is to filter out highly local accesses, serving them without reaching *dl1*, thus, reducing a big amount of power consumption. This approach and ours are both located at the same level, but ours focuses only on the stack trying to take advantage of the properties of stack accesses. The filter cache is really efficient when the data footprint is small enough to fit into the filter cache. Another difference between our approach and the filter cache is that the NeCK unit prevents the stack filter from incurring extra penalty on misses thanks to the early detection of them. In addition, a level 1 cache with one port plus a stack filter are able to serve up to two requests per cycle. These are the reasons why, when the footprint is not small enough to fit in the filter cache, the high number of capacity misses lead to many penalty cycles, extending the execution, and thus, the power dissipation.

Jang and others [23] propose so-called dynamic stack allocation where the stack pointer is shifted at run time to a memory location which is expected to cause least number of cache misses. They implement the proposed scheme using so-called dynamic stack allocator which consists of cache miss predictor to compute cache miss probability based on least recently used policy and stack pointer manager to manage multiple stack locations.

Cascaval and others [24] propose a method to estimate the number of cache misses, at compile time, using a machine independent model based on stack algorithms. The proposed algorithm computes the stack histograms symbolically, using data dependence distance vectors and is totally accurate when dependence distances are uniformly generated. The stack histogram models accurately fully associative caches with LRU replacement policy, and provides a very good approximation for set-associative caches and programs with non-constant dependence distances.The stack histogram is an accurate, machine-independent metric of locality. Compilers using this metric can evaluate optimizations with respect to memory behavior.

2.5. Experimental Environment

We have simulated our proposed SF over the studied target platform using Sim-Panalyzer [25], a simulation tool built on top of Simplescalar/ARM [26]. By default, it is configured to faithfully model the SA-110 StrongARM processor. To evaluate the efficiency of our data cache filtering, we use Sim-Panalyzer, with the parameters listed in Table 2.1.

For the simulations of our proposed filter in this ARM system, we have used several applications taken from the suite MiBench [27]: *adpcm*, *bitcount*, *CRC32*, *cjpeg*, *djpeg*, *patricia*, *rijndael*, *sha*, *tiffdither*, and a couple taken from the suite MediaBench2 video [28]: *mpeg2dec* and *jpeg2000dec*. All the applications have been

Clock	500 MHz
Branch predictor	Bimodal, 2K entries
IF queue size	8
Issue kind	In order
RUU	4
LSQ size	4
Decode/Issue/Commit width	2/2/2
Functional Units (INT)	ALU:2; Mult.:1
Functional Units (FP)	ALU:1; Mult.:1
L1 Instruction Cache	16K (32 way)
L1 Data Cache	16K (32 way)
L1 Data Cache latency	1 cycle
Memory access (first block)	64 cycles

Table 2.1: Simulation parameters for ARM studied processor.

simulated to completion using the reference workloads distributed with MiBench and MediaBench2. The compiler used is *GNU arm-linux-gcc-4.1* with the standard optimization flag *"-O2"*. Given that in the *CRC32* implementation delivered in MiBench the file is read in a character basis, we have added a modified version that reads the file in a 512 byte block basis, what is more efficient. This modified version has been labeled *crc** as opposed to *crc*.

The power model employed in this platform is the one supplied by Sim-Panalyzer [29]. The simulator has been modified to incorporate our filter in the micro-architectural simulation as well as in the power model. Provided that PAnalyzer has functions to model register files, the proposed filter is modeled similarly as the original register file. The NeCk unit has a cost of about 250 2-input gates, which is less than 1% of the logic in the processor. Each memory access decides whether it accesses the filter or the cache. Each time the filter is read from or written to a call to the PAnalyzer interface accounts the power consumption. When the filter rotates, the instruction dispatch is stalled. For each dirty word that has to be written back, we account for one read operation in the filter, one write operation in *dl1* and it incurs in a 1-cycle penalty. After the injection, the issue is resumed.

The technology parameters corresponding to 90nm have been extracted from CACTI 5.3 [30]. For those parameters without correspondence in CACTI, like $[p|n]mobility$, $Cgate$, $Cgatepass$, $C[p|n]diff[area|side|ovlp]$, $C[p|n]oxideovlp$, $Cpolywire$ and $R[p|n]channelstatic$ we have applied linear interpolation. The values of these and the rest of parameters of the used power model are shown in Table 2.2.

In Table 2.3 we show the percentage of chip area occupied by different parts of the ARM processor using a 32 word filter.

V_{dd}	1.2	nmobility	$409.16 \cdot 1.1$
L_{tech}	0.0902	β	$nmobility/pmobility$
L_d	0.0266	nR_c	11.3
L_{eff}	$L_{tech} - 2 \cdot Ld$	pR_c	11.8
nV_{th}	0.237	RK_p	0.82
pV_{th}	0.237	R_p	0.45
V_{th}	$\frac{nV_{th}}{2} + \frac{pV_{th}}{2}$	$packageC_{eff}$	$3.548 \cdot 10^{-12}$
t_{ox}	$1.2 \cdot 10^{-3}$	$C_{ndiffarea}$	$0.137 \cdot 10^{-15} \cdot \frac{1.2}{1.5} \frac{F}{\mu m2}$@1.5V
C_{ox}	$1.79 \cdot 10^{-14}$	$C_{pdiffarea}$	$0.343 \cdot 10^{-15} \cdot \frac{1.2}{1.5} \frac{F}{\mu m2}$@1.5V
e_{ox}	$C_{ox} \cdot t_{ox}$	$C_{ndiffside}$	$0.275 \cdot 10^{-15} \cdot \frac{1.2}{1.5} \frac{F}{\mu m}$@1.5V
nI_{dss}	$1076.9 \cdot 10^{-6}$	$C_{pdiffside}$	$0.275 \cdot 10^{-15} \cdot \frac{1.2}{1.5} \frac{F}{\mu m}$@1.5V
pI_{dss}	$712.6 \cdot 10^{-6}$	$C_{ndiffovlp}$	$0.138 \cdot 10^{-15} \cdot \frac{1.2}{1.5} \frac{F}{\mu m}$@1.5V
nV_{bi}	0.7420424	$C_{pdiffovlp}$	$0.138 \cdot 10^{-15} \cdot \frac{1.2}{1.5} \frac{F}{\mu m}$@1.5V
pV_{bi}	0.8637545	$C_{noxideovlp}$	$0.263 \cdot 10^{-15} \cdot \frac{1.2}{1.5} \frac{F}{\mu m}$
nC_{ja0}	$0.9868856 \cdot 10^{-15}$	$C_{poxideovlp}$	$0.338 \cdot 10^{-15} \cdot \frac{1.2}{1.5} \frac{F}{\mu m}$
pC_{ja0}	$1.181916 \cdot 10^{-15}$	C_{gate}	$1.95 \cdot 10^{-15} \cdot \frac{1.2}{1.5} \frac{F}{\mu m^2}$
nC_{jp0}	$0.08 \cdot 10^{-15}$	$C_{gatepass}$	$1.45 \cdot 10^{-15} \cdot \frac{1.2}{1.5} \frac{F}{\mu m^2}$
pC_{jp0}	$0.08 \cdot 10^{-15}$	$C_{polywire}$	$0.25 \cdot 10^{-15} \cdot \frac{1.2}{1.5} \frac{F}{\mu m}$
aspect ratio	2.4	$R_{nchannelon}$	$1.54 \frac{V_{dd}}{1076.9} \cdot 10^{-6} \Omega \cdot \mu m$
$Wtech$	$aspect\,ratio \cdot Ltech$	$R_{pchannelon}$	$2.45 \cdot R_{nchannelon} \Omega \cdot \mu m$
Wd	0.000899357	$R_{nchannelstatic}$	$\frac{25800}{8751} R_{nchannelon} \Omega \cdot \mu m$
$Weff$	$W_{tech} - 2Wd$	$R_{pchannelstatic}$	$\frac{61200}{20160} R_{pchannelon} \Omega \cdot \mu m$
pmobility	$84.78 \cdot 1.5$		

Table 2.2: Technology parameters for simulated ARM processor.

2.5.1. x86 configuration for comparative study

For the sake of completeness we include the configuration used for the evaluation of our technique in an $x86$ machine discussed in Section 2.6.2. We have used the same applications, compiled with the same compiler version, x86-linux-gcc-4.1, using the same optimization flags: *"-O2"*.

The simulation has been carried out using PIN [31] a dynamic binary instrumentation tool, that allows us to instrument memory references and *sp* modifications to simulate the stack filter in this architecture. The filter management, and hit/miss detection is done in the same manner than for ARM.

In the following, we perform some quantitative analysis to further understand the

Logic	Bimod	RAS	*dl1*+tlb	*dl1*+tlb	BTB	Filter	Regs	
							Int	FP
$< 1e^{-6}$	$3e^{-2}$	$2e^{-3}$	31.1+13.7	31.1+13.7	10.1	$1e^{-2}$	$6e^{-2}$	$8e^{-2}$

Table 2.3: Chip area breakdown for the branch predictor (Bimod), return address stack (RAS), data cache and data tlb, instruction cache and instruction tlb, branch target buffer (BTB), our proposed filter and the register files (%).

Filter hit/dl1 hit	0.0094
Filter hit/(dl1 hit+dtlb hit)	0.0064

Table 2.4: Relative Power Consumption.

effectiveness of the proposed design.

2.6. Evaluation

The purpose of our evaluation is to show that just by using a stack filter, a reasonable amount of power can be saved, having a negligible impact on the performance. We start with a comparison among three different filter sizes for ARM architecture. Afterwards, we compare the filter usage for two different target architectures, namely ARM and x86. Finally, we compare the stack filter with filter cache [9], and region-based caching [20] to show that for small sizes it achieves greater power savings with lower performance overhead.

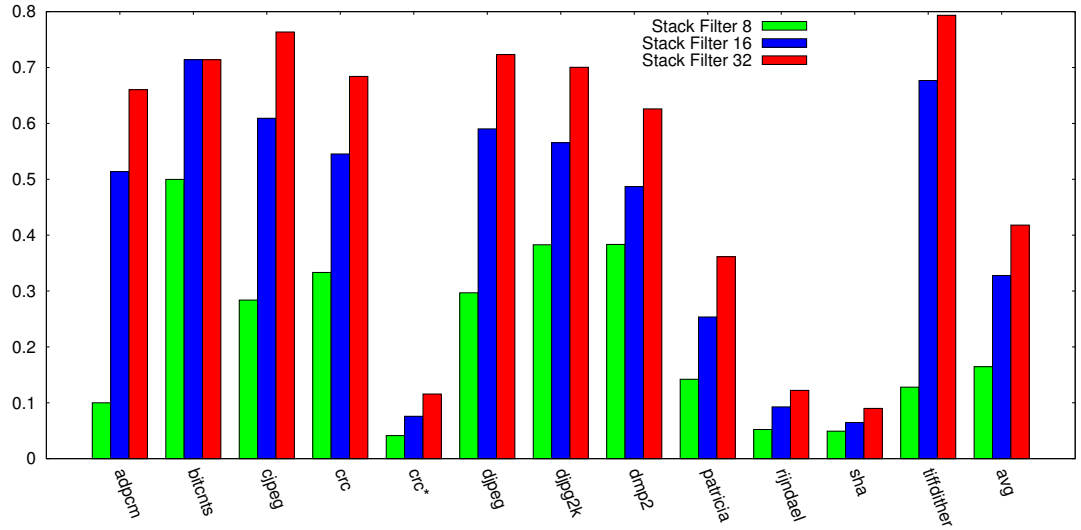


Figure 2.9: Stack filter hit rate over stack references for different sizes.

2.6.1. Filter Analysis

In order to choose what filter size to use, we have carried out runs for three different filter sizes: 8 words (32 bytes), 16 words (64 bytes) and 32 words (128 bytes). For the three sizes, half the registers are mapped to registers over the TOS and the other half are mapped to registers under the TOS. The measures have been averaged using the geometric mean.

To support our claims that a SF is a useful structure, we present its hit rate in Figures 2.9 and 2.10. The first figure shows the hit rate considering just references

to the stack, the second one shows the hit rate taking into account all memory references (total hit rate). If we focus on a 32-word filter, *cjpeg* and *tiffdither* achieve over 75% hit rate, and on average (geometric mean) applications achieve over 40% hitrate. For these benchmarks, the hit rate achieved by a 16-word filter is over 60%, and the average across benchmarks is slightly over 32%. These numbers scale down when all memory accesses are considered. As an illustration, *adpcm* uses the stack rarely, therefore, the elevated hit rate (65%) turns into a poor total hit rate. Nevertheless, 6 out of the 13 benchmarks manage to achieve over 25% hit rates with a 32-word filter, which translates directly into 25% less accesses to *dl1*.

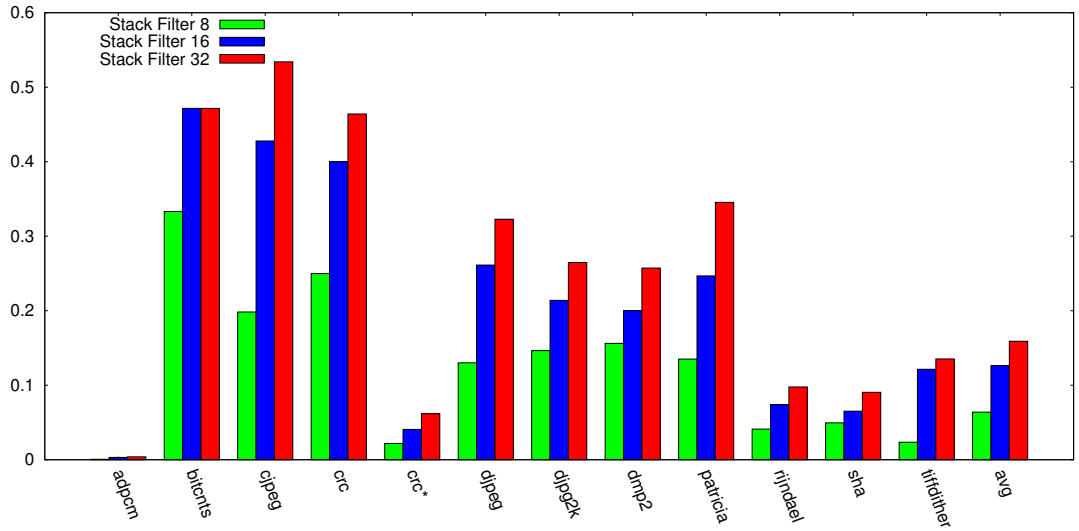


Figure 2.10: Stack filter hit rate over memory references for different sizes.

As expected, the reduction in the number of accesses to *dl1* implies a reduction in the power consumption of that structure. In Figure 2.11 we show the power consumption ratio in *dl1* with respect to a baseline machine (same processor with no stack filter). The maximum savings are achieved for *cjpeg* with more than half of the power consumption saved, and almost 25% power saving on average. Likewise, power consumption in data TLB is reduced. This happens because a virtually addressed cache only looks up the TLB when the access misses in the filter and goes to *dl1*. For all those benchmarks with few stack usage the power savings achieved are negligible.

The SF is conceived as power-oriented and not performance-oriented. Its latency is the same as that of *dl1*, so, at first glance no performance gain is expected. No performance loss is expected either, because the filter hit is detected early meaning no extra penalty in SF misses. This is not completely true, there are some side effects that slightly affect performance. Figure 2.12 shows the slowdown/speedup for different applications. It varies from a x1.1 slowdown for *crc* with a 8-word filter to a x1.01 speedup for *sha*. The main cause for the slowdown especially for *crc* is that the number of function calls produced in a small lapse of time is big. When that happens, activation records are written into the stack, but shortly after data is written, it is displaced out of the filter due to SF rotations caused by the stack

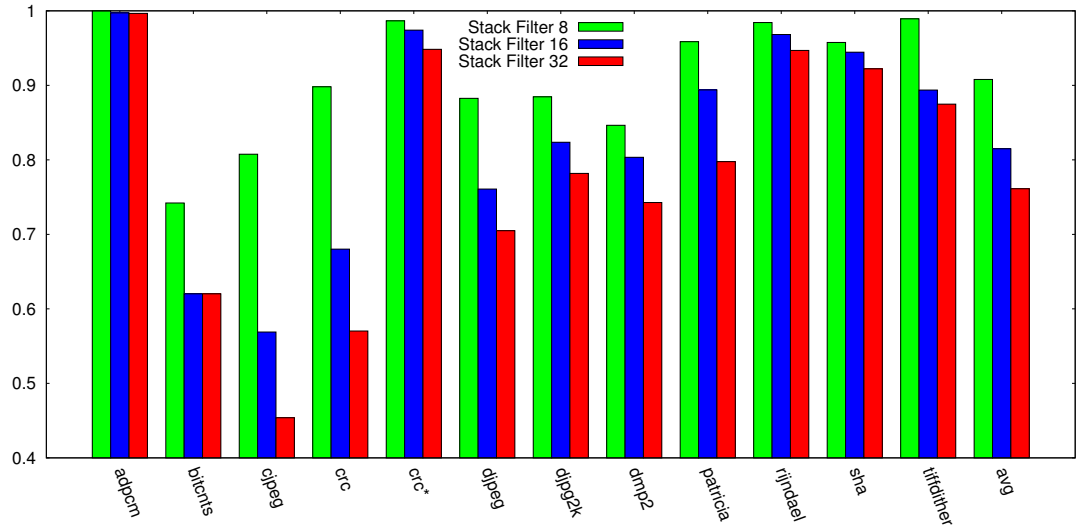


Figure 2.11: Stack filter *dl1* power ratio (filter/baseline).

movements caused by procedure calls and returns. When data is displaced, the SF must write it back to *dl1*, and that takes time. In the more reasonable (block based) version labeled *crc**, the stack pointer is modified less, so the performance loss is negligible. On the other hand, when using a SF, we are increasing the execution resources. We are adding 32-128 bytes of non-inclusive pseudo-dl0 storage. This extra storage prevents conflict misses between data in the cache and data in other memory regions. In addition, the SF has its own ports, what makes possible to serve two memory accesses in the same cycle as long as one of them hits in the SF and the other one targets an address outside the SF.

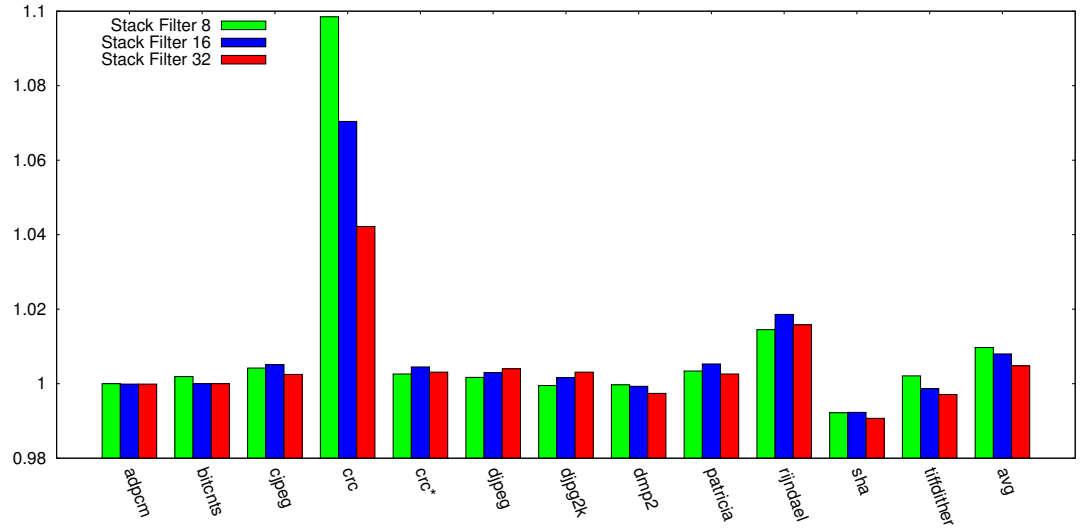


Figure 2.12: Execution time variation for different sizes (filter/baseline)

To conclude the comparison among the different filter sizes we discuss overall power

consumption. Table 2.4 shows that the restrictive, direct-mapped stack filter is far less power hungry than a set-associative, flexible *dl1* cache. Consequently, the power consumed by the accessing the filter is much less than the savings achieved in *dl1*. Figure 2.13 collects the variation in power consumption for different filter sizes and benchmarks. It is important to notice that, no matter how well or bad the filter performs, if it is at least 16-word in size, power consumption is never higher than the baseline. Power savings in the whole processor are heavily determined by *dl1* power savings, depicted in Figure 2.11. Saved power in *dl1* is divided by how much of the overall power is consumed by *dl1* cache and data TLB. Power consumption is also affected by the variations in execution time. The longer (respectively the shorter) a program is executing, the more (respect. less) power is consumed due to leakage. A 32-word SF is able to reduce almost 10% power in the execution of *cjpeg*, over 6% for *crc*, 5% for *djpeg* and between 1.5% and 4% for *bitcnts*, *djpeg2k*, *dmpeg2*, *sha* and *tiffdither*. Power reduction for a 16-word bit is not so good, even though, 5 benchmarks manage to save more than 2% power.

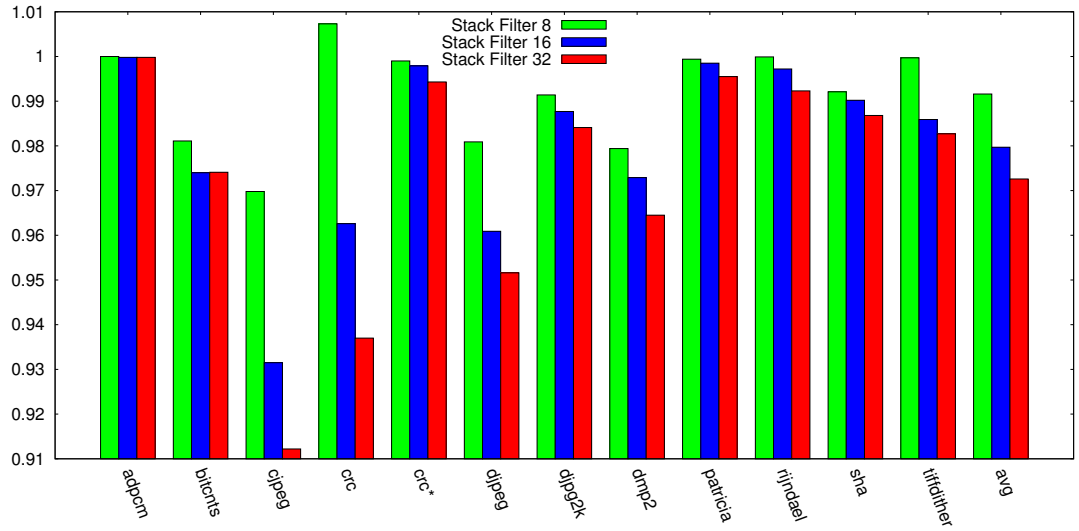


Figure 2.13: Stack filter micro-architecture power ratio (filter/baseline).

For the rest of the evaluation section we just consider a filter with 32 word capacity (128 bytes plus 64 bit metadata, and 5-bit base register), out of which 16 words map over the TOS and 16 words map under the TOS, because, as shown in the previous analysis, this configuration achieves important savings with really few extra storage required.

2.6.2. ISA Impact

This part of our study focuses on detecting the impact of the architecture over the usage and profitability of a stack filter. The filter configuration used for the comparison is the same for ARM and x86: 32-word filter, 16 words over the TOS and 16 under the TOS.

Figures 2.14 and 2.15 show the hit rate and total hit rate for the same applications when running on ARM and on x86. Both numbers are far bigger for x86 than for ARM. This is due to two factors. The first one is that an x86 machine has less architected registers than an ARM machine, 8 as opposed to 16. With so few registers, a register allocator has a harder time allocating variables into registers, so variables are spilled to the stack and reloaded more often, resulting in more references to the activation record and a higher locality, what translates into a higher hit rate. The second reason is the calling convention. Due to the small number of architectural registers, the x86 calling conventions mostly pass arguments on the stack; only the return value, or a pointer to it, is passed in a register. This, again, translates into hitting accesses to the SF. To illustrate this point, Figure 2.15 shows that for 8 out of 11 applications the total hit rate is over 50% in the x86 machine. This means that half the accesses to *dl1* cache are filtered. In the light of this results, we can state that x86 low power processors such as Intel Atom would take more advantage of a SF.

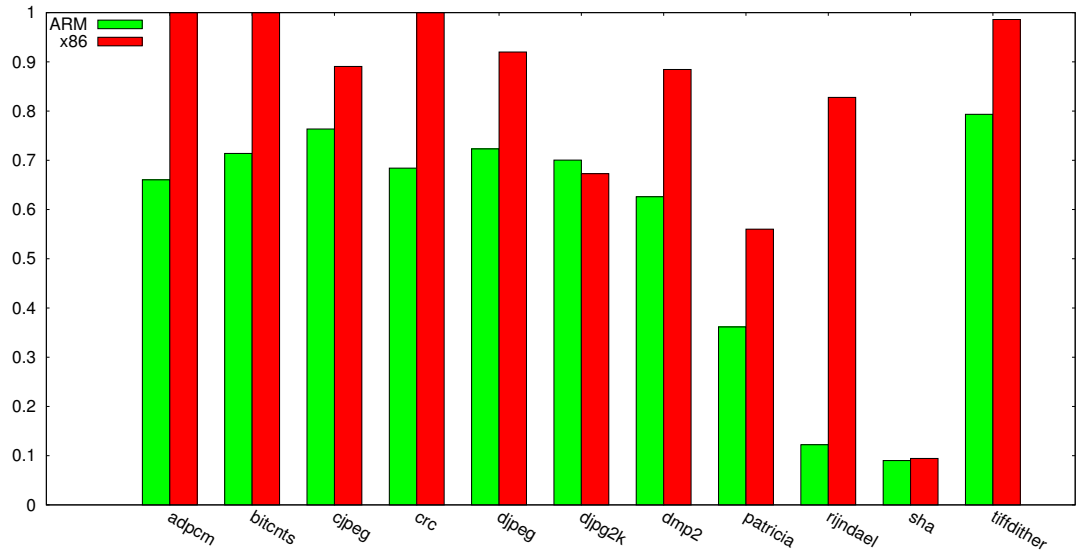


Figure 2.14: Stack filter hit rate over stack references for ARM and x86.

2.6.3. Comparative study with previous proposals

The last part of this evaluation addresses the comparison of our SF with Lee and Tyson's region-based caching architecture [20], and Kin *et al.*'s filter cache [9]. The purpose of this comparison is to point out the weaknesses of these approaches and show how a SF overcomes these weaknesses.

In the following, we compare a 32-word SF, a 32-word filter cache (fc), and a specialized cache for the stack region for different sizes measured in words (r), all of them normalized to a baseline that is an ARM processor without any of the techniques.

In Table 2.5 we show the slowdown/speedup for different approaches.

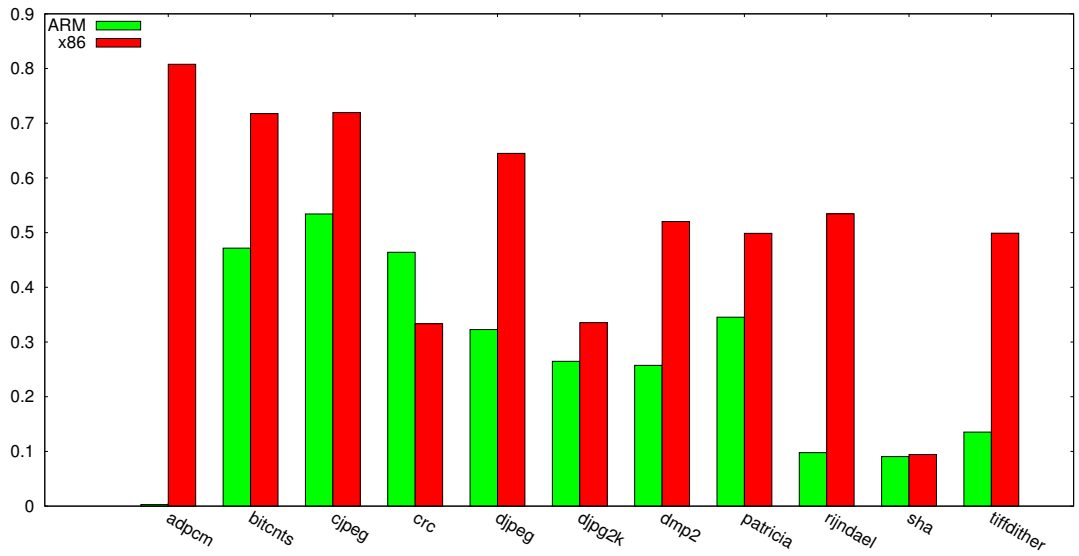


Figure 2.15: Stack filter hit rate over memory references for ARM and x86.

When the proposed size is as small as 128 bytes extra storage (32 words) the filter cache is unable to outperform a stack filter for any of the benchmarks. For region-based caching, it takes, in many applications, a 512-word cache to match or outperform a 32-word stack filter. The main reason why region-based caching does not work so well, is because we are targeting a small processor, without even L2 cache. In the absence of a L2 cache, stack-cache misses have main-memory-access latency. A killing scenario is locating an array bigger than the stack-cache in the stack region, and iterating over it. It is bound to generate stack-caches misses. In region based caching all this misses are served by the next level in the hierarchy, which in our target processor is main memory, dramatically slowing down the execution. On the other hand, a SF will just hold some of the elements of the array, and the rest will be retrieved from *dl1*, thus, the execution is not slowed down. This high latency misses cause a big slowdown in benchmarks with lower hit rate like *rijndael* for all simulated sizes, or *crc* and *sha* for under 128-word stack-region caches. In order to overcome this high latency, the size of the stack cache needs to be quite big (about 2KB) to achieve a miss rate low enough to be beneficial. This reason makes region based caching weaker than our proposal in designs without L2 cache in which the extra storage size needs to be as small as 32 or 64 words.

The reason behind the bad performance of the filter cache is that it takes no advantage of the special locality exhibited by stack accesses, or any accesses at all. Therefore, when two disjoint local regions are being accessed, they are subject to conflict misses. In addition, for benchmarks in which the data footprint is bigger than the filter cache, capacity misses make the miss rate grow, slowing the execution down, because misses incur in a one cycle penalty, plus one more cycle if a write-back is required. For example, in the execution of *djpg2k* the miss rate is 0.282 and the write-back rate is 0.185. The high miss rate and the high write-back rate require a lot of *level 0-level 1* traffic, slowing down the execution.

	adpcm	*bcnt*	*cjpg*	*crc*	*djpg*	*djpg2k*	*dmp2*	*patri*	*rjndl*	*sha*	*tiffdith*
sf	0.99	1.00	1.00	1.04	1.00	1.00	0.99	1.00	1.01	0.99	0.99
fc	1.03	1.00	1.08	1.07	1.13	1.12	1.08	1.00	1.06	1.01	1.04
r32	1.00	1.19	5.28	5.47	7.45	1.12	2.01	1.71	10.9	6.57	1.00
r64	1.00	1.10	2.85	5.24	4.41	1.07	1.47	1.33	9.21	5.76	1.00
r128	1.00	1.10	1.44	1.00	2.39	1.53	1.27	1.10	7.88	4.30	0.99
r256	1.00	1.00	1.14	1.00	2.08	1.52	1.17	1.06	6.79	1.15	0.99
r512	1.00	1.00	1.08	1.00	1.06	1.51	1.14	1.02	5.78	1.10	0.99

Table 2.5: Execution time variation (technique/baseline) for each technique and benchmark

To wrap up the evaluation of our approach, we show the energy-delay product (EDP) of the different approaches for every benchmark in Table 2.6. EDP follows the line of slowdown, but emphasizing the differences. This means that power consumption differences are closely related to execution time differences. A SF is able to keep the product under 1 for all benchmarks but *rijndael*. The filter cache is outperformed in all benchmarks and only manages to have the product under 1 in two benchmarks, *sha*, and *bitcnts*. Concerning region based caching, it is not able to outperform SF with the same size for any of the benchmarks. Moreover, even with a 2KB cache size, the EDP is higher than that of the stack filter for most benchmarks. Finally, it is worth noting that in both Table 2.5 and Table 2.6 we are using *crc* instead *crc**, which would be better for our technique.

	adpcm	*bcnt*	*cjpg*	*crc*	*djpg*	*djpg2k*	*dmp2*	*patri*	*rjndl*	*sha*	*tiffdith*
sf	0.98	0.97	0.91	0.97	0.95	0.98	0.95	0.99	1.00	0.97	0.97
fc	1.01	0.98	1.06	1.06	1.14	1.18	1.10	0.99	1.07	0.97	1.02
r32	1.00	1.48	21.3	72.8	40.2	2.35	5.18	2.99	213	42.5	0.99
r64	1.00	1.24	7.10	54.3	17.9	1.28	2.46	1.78	150	32.9	0.99
r128	1.00	1.23	1.98	0.94	6.73	1.99	1.67	1.19	109	16.4	0.97
r256	1.00	0.98	1.26	0.94	5.23	1.93	1.40	1.11	85.7	1.35	0.97
r512	1.00	0.99	1.13	0.94	1.09	1.90	1.32	1.03	60.0	1.21	0.97

Table 2.6: EDP for each technique and benchmark

2.7. Conclusions

All programs have a software stack to hold, among other things, the state of the machine through function calls. Among the special features of this stack there is locality, both spatial and temporal. The closer a word is to the TOS the higher the locality is. Caches naturally exploit this locality. While there are several proposals to increase performance through stack cache, we believe that using a stack filter, having power consumption savings as goal, is an easy way to deal with the power-hunger of cache memories, because the stack doesn't need all the power and flexibility of the caches.

In this chapter we have shown that a small register-based storage is enough to filter a lot of stack references, significantly reducing the number of accesses to *dl1*. This reduction implies less power consumption in the data cache. In addition, the filter

is register-based and smaller than the cache, therefore, the power per access is much less than that of the data cache. Concerning performance, our filter has the same latency than *dl1*, and filter misses incur no penalty, so little performance variation is expected, and as the experimental results show, little is obtained.

The evaluation of this technique in both ARM and x86 machines confirms that our approach reports satisfactory results in such different target platforms. Just a small filter is enough to significantly reduce the power dissipation of the *dl1*, and the processor overall power consumption. Including the stack filter leaves performance almost unvaried. The little variations observed are brought about by the non-inclusiveness of the filter, that means a little increase in the cache size, which increases performance, and the overhead incurred by *sp* updates, which slightly decreases performance. All of this leads us to think that a register-based stack filter is an easy-to-implement way to reduce power consumption.

Chapter 3

Debuggability: Data Race Detection with Minimal Hardware Support

This chapter takes care of debuggability, at a shared cache level. In this second part we present Accessed Before, a simple and light on-line data race detection algorithm, and the small hardware extensions required for it to work. The intuition behind Accessed Before is using the coherence protocol to avoid extra explicit and expensive inter-thread communication.

We show a complete evaluation of Access Before. First we evaluate the accuracy comparing it with Happened Before-based race detection. Next, we show high level overhead characterization, space and performance. And a comparison with other academia algorithms. The last part of the evaluation gathers a comparison with two commercial tools for data race detection.

As a final contribution, we show the complete proof that Accesses Before is complete in that for every data race there exists an instruction interleaving that would expose the data race such that Accessed Before can detect it.

This chapter is arranged as follows. Section 3.2 provides the necessary background to understand our work, revisiting key concepts about data race detection and cache coherence protocols. Section 3.3 presents the main idea in the algorithm and its hardware support, how it works and why it works. In Section 3.4, we show more details Section 3.5 discusses related work, Section 3.6 briefly describes our experimental environment, and Section 3.7 presents our evaluation and respective analysis. Section 3.8 concludes. Appendix A presents a proof that our approach eventually finds all static data races in a program, given sufficient execution variability.

3.1. Introduction

Developing much-needed parallel and concurrent software for multicores is an even harder task than developing sequential code. Programmers have to deal with subtle interaction between threads and often hit complex concurrency bugs. Among these bugs, data races are the most common type. A data race occurs when two memory operations in different threads, at least one of which is a write operation, access the same memory location, and are not ordered by synchronization. Non-synchronized accesses to shared data could lead to crashes or silent data corruption, so current languages including Java [32] and the new C++ standard [33] completely disallow or discourage data races.

For the reasons above, researchers have proposed a variety of algorithms to detect and avoid data races, including many hardware-only [34, 35, 36, 37, 38] and software-only [39, 40, 41, 42] implementations.

Hardware-only implementations are typically complex. They require extensive hardware support, like changes to the cache hierarchy, including augmenting cache blocks with significant additional state, extending cache coherence messages to carry additional information, and modifying the cache coherence protocol state machine to check for events of interest. Also, the storage requirements, many times close to key processor structures, are quite prohibitive. Software-only implementations, on the other hand, can be used without modifying the architecture, but are typically too slow to be an always-on feature. The analysis operations performed in software are slower, and moreover, these algorithms require a significant amount of meta-data and frequent inter-thread communication.

In this chapter, we propose a hybrid solution called Accessed Before: hardware support is boiled down to the bare minimum, reducing complexity, and making detection of inter-thread communication much faster than prior approaches. We augment the ISA with one simple instruction that takes an address as input and returns the coherence state of the cache block containing that address. We also propose a new algorithm that uses this support to effectively detect data races. Our solution leverages two key insights: (1) the dynamic information we need can be extracted from coherence state already tracked by the hardware; (2) there is a well-defined category of dynamic data races that are much cheaper to detect and yet can be proved to include all static data races given sufficient executions. We also show how to perturb execution schedules to speed up the exposure of data races to the detection mechanism, achieving high accuracy compared to traditional happened-before data race detection, but at significantly lower space and time overheads.

In this piece of work, we present Accessed Before with its associated hardware support and use widely known benchmarks to evaluate its performance, comparing it with state-of-the-art commercial applications: Valgrind's Helgrind, and Intel Thread Checker. We evaluate the algorithms in a shared-memory 8-core machine, and then study how the performance overhead of each tool scales with the number of threads and workload input set size.

Given that the common case is the absence of races and the required hardware support is as fast or faster than a regular load instruction.

We estimate performance by implementing Accessed Before using the Intel Pin framework and running all benchmarks native instead of on a simulator. This allows for a more realistic performance comparison.

3.2. Background

Terminology We refer to instructions using the standard terminology: *static instructions* refer to the static object code and are identified by their instruction address; *dynamic instructions* refer to all instances of static instructions executed by threads. Similarly, A *static race* refers to a pair of static instructions that, when executed, may be involved in a data race, and a *dynamic race* refers to one manifestation of a data race at execution time. When we refer to threads, we may refer to *the local thread*, *i.e.*, a specific thread where data race detection is taking place, or to *remote threads*, *i.e.*, threads that may be interacting via shared-memory operations with the local thread. If a dynamic race has taken place, we may refer to a local thread as *the detecting thread* and to a remote thread as *the offending thread*. An *epoch* is the set of dynamic instructions in a thread executed between two consecutive synchronization operations. To simplify our discussion, we assume a static one-to-one mapping of a thread to a core and its associated private cache. We discuss how we handle scenarios in which this assumption does not hold in Sections 3.4.3 and 3.4.4.

We focus on standard POSIX threads, *pthreads*, and associated synchronization mechanisms, such as thread creation and joining, mutex creation, acquisition and release, and conditional variable creation, waiting and signaling. We define as *source* of a synchronization the thread initiating the synchronization, *e.g.*, by unlocking a mutex or sending a signal. The *destination* of this synchronization is the next thread to synchronize through the same construct as the source, *e.g.*, by locking the mutex or waiting on the signal. When we refer to epoch ordering, we mean Lamport's happened-before partial order [43].

3.2.1. Data race detection

When writing shared memory concurrent programs, programmers must use synchronization to restrict the order in which different threads perform certain operations and thus control access to shared variables. A data race occurs when programmers omit synchronization operations, and allow more than one thread to access a variable in an non-synchronized fashion, where at least one of these accesses modifies this variable.

Figure 3.1(a) shows a scenario with two properly synchronized (race-free) accesses to variable v, *i.e., a race-free scenario.* First, thread 1 writes v. Then, thread 1 releases

lock L and synchronizes with thread 0 when thread 0 acquires lock L. Finally, thread 0 reads v. Figure 3.1(b) illustrates a data race. As before, thread 1 writes v. However, in this scenario, thread 0 proceeds to read v without synchronizing, which characterizes a data race. Note that this would not be considered a data race if thread 0 only had initially read, not written, variable v.

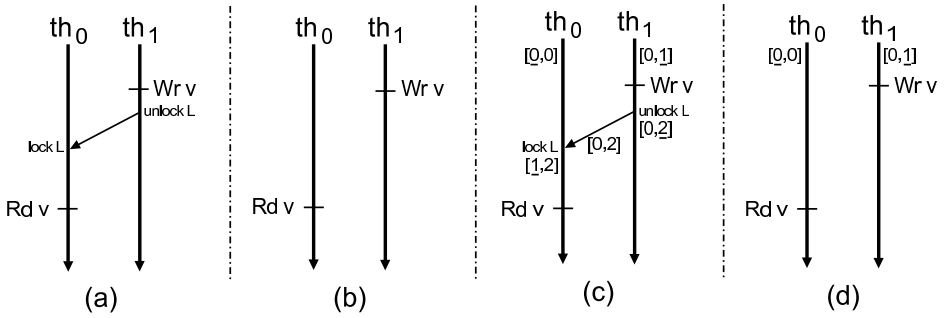


Figure 3.1: Multithreaded executions: (a) no data race occurs; (b) data race occurs; (c) happened-before relation and vector clocks when no race occurs; (d) happened-before relation and vector clocks when race occurs.

There are two basic approaches to data race detection: happened-before based algorithms [44, 45, 37] and lock-set based algorithms [41, 38]. Lock-set based algorithms check that the locking discipline is being followed by monitoring all lock acquires and releases, as well as all memory operations. For any one memory access to a given shared variable, there must be at least one lock that is consistently held on all other accesses to that same variable. In this chapter, we focus on happened-before algorithms, which rely on Lamport's happened-before relation [43] to partially order memory accesses based on observed synchronization operations and program order. If there is no ordering between any two accesses (at least one of them being a write access) to the same address, a data race is detected. The next section shows that this idea can be applied to epochs to implement a happened-before data race detection algorithm.

Set-based data race detection works recording for each thread two sets. One holds the variables read, and the other the variables written. When the thread performs a synchronization operation, the sets are copied to the history, intersected with those in the history and cleared. If, when we intersect the sets of concurrent epochs, any intersection is non-void, i.e. there exists a variable v such as $v \in (W_{i,e} \cap R_{j,e'}) \cup (R_{i,e} \cap W_{j,e'}) \cup (R_{i,e} \cap W_{j,e'})$ where i is the identifier of a given thread, $j \neq i$ is the identifier of the would-be racing thread, and e and e' are concurrent epochs. Under these conditions we can state that there is a race in variable v between threads i and j. Doing this requires a lot of communication between threads. Whenever an epoch ends for a given thread, that thread must ask every other thread for the read and write sets of their epochs such that there is no order between them and the ending epoch.

3.2.2. Happened-before race detection

Lamport's happened-before relation is a binary relation defined between events of a concurrent system. We apply its definition to a multithreaded execution where we define each epoch as a single event. This implies that the same ordering constraints that apply to an epoch also apply to all instructions within that epoch. The definition of the happened-before relation ($\rightarrow$) is as follows: (1) If epochs e_1 and e_2 occur in the same thread and e_1 precedes e_2 in program order, then $e_1 \rightarrow e_2$, *i.e.*, epoch e_1 happened before epoch e_2; (2) if the last operation of e_1 is the source of a synchronization, and e_2 begins at the destination of that synchronization, then $e_1 \rightarrow e_2$, *i.e.*, epoch e_1 happened before epoch e_2; and (3) the definition is closed by transitivity: $e_1 \rightarrow e_2, e_2 \rightarrow e_3 \Rightarrow e_1 \rightarrow e_3$

Race detection algorithms based on this relation associate logical clocks with each thread to encode the ordering established by the happened-before relation. Local clocks are incremented every time threads are involved in a synchronization operation. Each thread also records the latest logical clock value it observes from each of the other threads. Thus, each thread manages its own vector clock, consisting of one local component and as many remote components as remote threads. When two threads synchronize, the source sends its vector clock to the destination, which in turn performs an element-wise maximum operation between its own vector clock and the received one.

Figure 3.1(c) shows an example. There are two threads, each vector clock has two elements: a local (underlined) and a remote component. When thread 1 releases lock L, it associates its own vector clock with L. When thread 0 acquires lock L and observes thread 1's vector clock, it selects the maximum of each pair of components to calculate its new vector clock, and then increments its local component. When a non-synchronizing access happens, the accessed variable is tagged with the vector clock corresponding to the current epoch in the thread performing the access. For example, when thread 1 writes v, v is tagged with [0,1]. In addition, the new clock value is compared to the previous clock value associated with v. If the old clock value represents a time that is not earlier than the new one, the two accesses are unordered and a data race is detected. Figure 3.1(d) shows such a case. When thread 0 reads v, v's old clock is [0,1] and the new clock is [0,0]. This indicates that thread 1 has not synchronized with thread 0 since the last time thread 1 wrote v, and the data race is detected.

Note that the algorithm can be implemented by recording the address of all variables read and written within each epoch into read and write sets, along with the epoch's vector clock, and by checking for overlaps of these sets each time an epoch ends. We compare our proposed algorithm with this implementation (HapB). Also, note that HapB is complete, *i.e.*, any race that takes place during execution will be detected, but it requires a significant amount of communication among threads: the vector clock must be communicated on every synchronization and the set comparison requires expensive cache transfers.

3.2.3. Cache coherence

We now briefly review concepts related to the cache coherence protocol upon which we build.

Without loss of generality, we assume an invalidate-based MESI protocol.

A cache block is always in one of the following states: M (modified) if it is only cached in the local cache and its contents are different from main memory — this cache has the responsibility of supplying an updated copy of the block to other caches if they request it; E (exclusive) if it is cached only in the local cache and its contents have not been modified; S (shared) if the block is in the local cache without having been modified. It may be present in other caches as well; or I (invalid) if the contents must to be fetched again due to a remote invalidation, that rendered the local information out of date.

Before a cache attempts to modify a block, it needs to set that block in M state. This is achieved by sending invalidate messages to all sharers of the block and waiting for the corresponding acknowledgments. The exception to this is if the cache block is in E state, in which case the state can immediately and silently transition to M. Before a cache can read contents of a block, it must have that block in any of the stable states other than I. This is achieved by sending a request for a copy of the block. When the data is received, the block is installed in state E if no other cache has a copy or in S if any other cache has a copy.

In the following, we show how to use this to speed-up race detection.

3.3. Minimal hardware support for data race detection

Our proposed minimal hardware support for data race detection consists of simply exposing the coherence state of a cache block to the software layer via one additional instruction. A software layer then records and uses the state information to detect data races. To leverage this support, we propose a new race detection algorithm, "AccessedBefore", or simply AccB. The key idea is to track the last observed state of cache blocks and detect if they have been downgraded within the boundaries of an epoch. A downgraded block within an epoch indicates a potential data race: a remote cache must have requested a downgrade in the local cache. Note that all state necessary to the analysis is local to a thread, so no inter-thread communication is required for race detection; HapB, in contrast, requires substantial inter-thread communication.

3.3.1. AccessedBefore (AccB) Algorithm

The idea behind AccB is using the implicitly generated coherence traffic to avoid inserting extra operations to detect races. Whenever a thread writes a memory

location, all caches are notified to invalidate local copies. Also, if a memory location is written in a local cache and a remote thread reads this location, the local cache receives a coherence request to send the corresponding value and downgrade the cache line to shared state. It is possible to detect data races by leveraging state transitions and tracking information about epochs.

Figure 3.2 illustrates how race detection works in AccB. First, thread 0 performs a synchronization operation and starts an epoch (1). When thread 0 performs a store to variable v (2), the already existing coherence protocol transitions the corresponding cache block to M state and the software layer records the pair of *address* and *state*: $< v, M >$ in a local table. When thread 1 subsequently reads variable v (3), a coherence action is triggered, causing the block cached by thread 0 to downgrade to S state. At this point, the software layer is unaware of the downgrade. Finally, right before thread 0 writes v again (4), the software layer examines the current state of v's block in the local cache (S) and the state recorded in its local table (M), observes that a downgrade has happened, detecting the data race. For those cases in which the second access does not happen, a downgrade check is performed when the epoch ends (5).

Table 3.1 shows the different types of downgrade and the kind of race that causes them. For example, the first row corresponds to the example in Figure 3.2. The downgrade detected is a $M \rightarrow S$ transition and the first access in the local thread is a write. From this information, we can conclude that the conflicting access is a read and that the race consists of a read after write (W→R).

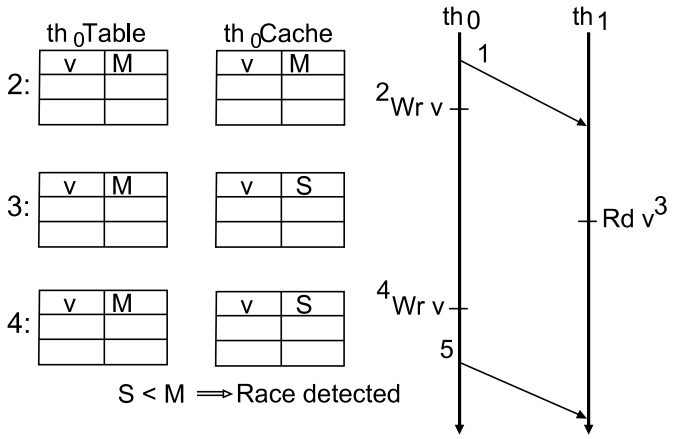


Figure 3.2: Using coherence state to detect a race.

Table 3.2 summarizes our algorithm's operation by explaining the actions taken by the software layer on each relevant event. Again, note that all analysis actions are local to a thread: the only communication between threads happens through the cache coherence protocol (which would be present even in the absence of our technique). Also, the information collected into the local table only pertains to a *single epoch*, as we are not interested in downgrades that happen across synchronization operations. These are two important advantages of our approach when compared to HapB, which has additional storage and communication requirements.

Transition at local thread	Access at local thread	Access at remote thread	Race type
$M \rightarrow S$	write	read	W→R
$M \rightarrow I$	write	write	W→W
$M \rightarrow I$ $E \rightarrow I$ $S \rightarrow I$	read	write	R→W

Table 3.1: Types of downgrades and races.

Event of interest	Algorithm action
Beginning of epoch	Clear local table.
Before memory access	Check the current state of the corres–ponding cache block against the entry in its local table (if any) to detect downgrades.
After memory access	Record the state of the corresponding cache block in its local table.
End of epoch	Check every entry in its local table and their corresponding state in the cache to detect remaining downgrades.

Table 3.2: Events of interest and related algorithm actions.

At the beginning of an epoch: the software layer clears its local table.

Immediately before a memory access: the software layer checks the current state of the corresponding cache block against the entry in its local table (if any) to detect downgrades.

We provide simple hardware support for data-race detection by exposing information intrinsic to the cache coherence state of a cache line.

We leverage the implicit communication already done by the cache coherence protocol to avoid further communication between threads. The insight is that, if a variable is shared by a set of caches, then these threads will produce coherence traffic among their caches. Furthermore, we can define an order between the four different MESI states: $M > E = S > I$. This order quantifies the thread ownership level of the block. Whenever a thread owns a block and shares it, the level of ownership over the block downgrades, *i.e.*, if a thread has a block in M state in its local cache, and then a remote thread reads the block, or even worse, writes the block, the state in the local cache is downgraded to S or I, respectively. It is easy to infer that state downgrades are related with sharing, and thus, these downgrades can only happen safely, *i.e.* in a data-race free way, if the operation that set the block in the initial state and the remote operation that triggered the state downgrade are properly ordered by synchronization operations.

As a result, data-races can be detected by monitoring accesses both local and remote within epoch boundaries.

To detect sharing within the boundaries of an epoch –between synchronization

operations–, we just need to focus on the evolution of the coherence state of variables in the local cache. From the first time the local thread accesses a variable on, it must "keep an eye" on the state of that variable until the epoch ends. If a race takes places, is because another thread accesses to a variable, v, that has been accessed in the ongoing epoch, in other words, a variable the local thread is "keeping an eye on". For the sharing to actually be a race, at least one of the operations must be a write. First, let us consider the scenario in which the local thread reads v, therefore, it is on S (or E) state. When the offending thread writes v, it triggers a $S \rightarrow I$ transition in the cache of the local thread, which is a downgrade, and as such, deemed a race. Regarding the other possible scenario, if the local thread writes v, then v is set in M state in the local cache. When the remote thread reads (respectively writes) v, it triggers a $M \rightarrow S$ (respect. $M \rightarrow I$) transition in the local cache, which is, again, a downgrade and a race is reported.

If we take a look from the other perspective, this is, instead of considering the downgrades implied by a race, let us analyze the meaning of downgrades on variable v within the boundaries of an epoch:

- $M \rightarrow S$ This means that the local thread wrote v and later a remote thread read v, meaning a W→R race.
- $M \rightarrow I$ This means that the local thread wrote v and later a remote thread wrote v again, meaning a W→W race.
- $S \rightarrow I$,$E \rightarrow I$ This means that the local thread read v and later a remote thread wrote v, meaning a R→ race.

Under the light of this insight, we can assert that detecting races inside epoch boundaries is equivalent to detecting downgrades. This means that the safe way for downgrades to happen is across epoch boundaries, *i.e.*, with synchronization operations between the first and the second access.

To monitor the state of each variable, the thread just needs some extra (local) storage to record the state in which the block was set after the last (local) access. Then, it will need to compare this recorded state with the current state in the cache. This comparison only needs to be performed right before any subsequent access to the same variable and at the end of the epoch to detect any downgrade that could have happened during the epoch. Note that only shared variables need to be monitored, since non-shared variables lie in the thread stack, which is private to each thread. Besides, once the epoch ends, there is no point in keeping the previous state of variables accessed in that epoch. As a result, we only need to keep information about the current epoch.

We are considering that, in the event of a race, the local thread, i, always accesses the variable before the offending thread, j. We can do this without any loss of generality, because, inasmuch as every thread monitors its variables, we can consider j as the local thread and i as the offending thread. If that is the case, thread j would be the one detecting the race.

It is important to note that the structure needed for this mechanism is local, *i.e.* non-shared, and there is no explicit communication among threads, unlike in HapB. All the communication needed is implicitly done by the underlying hardware implementing the coherence protocol, and takes place regardless of the fact that we are taking advantage of it. These two facts dramatically reduce the overhead of our detection algorithm when compared to HapB.

3.3.2. Sources of Inaccuracy

Our algorithm has three sources of inaccuracy: (1) false sharing, (2) cache block evictions, and (3) early epoch ending. We discuss their impact and mitigation below. Section 3.7 quantifies the impact they have.

False sharing: We chose to keep the hardware support required by our proposal to a minimum, so we do not extend the memory access information to granularity finer than what is already provided by coherence protocols: a cache block. False sharing of the block may result in false positives. Any other race detection approach using the same granularity would have the same limitation (*e.g.*, HapB). Moreover, our approach can be easily extended to finer granularity if necessary (at an extra cost) and is orthogonal to software techniques to mitigate false positives due to false sharing.

Cache block eviction: Once a block is evicted from a cache, its associated coherence state is lost. As a result, the evicting cache loses its ability to detect downgrades, so it may miss data races (false negatives) as a result of a remote access that would downgrad a block if it was in the cache, but since the block was evicted, it goes unadverted. Evictions can never introduce false positives because after the loss of the coherence information, any new access will "reset" the state of the variable in the local table, not introducing any false downgrades. We have again chosen a very simple approach, as saving and restoring state information when a block is evicted would be quite complex and would result in significant storage requirements.

Early epoch ending: Once an epoch ends, the access monitoring of variables in the epoch also stops. Therefore, if there are later downgrades to the corresponding cache block resulting from non-synchronized memory accesses, they are not detected and a race might be missed (false negative). Figure 3.3 illustrates this issue in more detail. Thread 1 reads variable v and starts monitoring it (1). Next, thread 1 ends its epoch by acquiring a lock unrelated to v (2), at which point it stops monitoring variable v. Sometime later, thread 0 performs a write operation to v (3). Even though the read and the write accesses are unordered, the data race is not detected at thread 1 due to its early epoch ending.

In essence, our algorithm looks for access conflicts between concurrently running epochs. Any such conflict is necessarily not ordered by synchronization operations and therefore must be the result of a race (or false sharing as described above). This implies that a race will be detected if the epochs in which their memory accesses

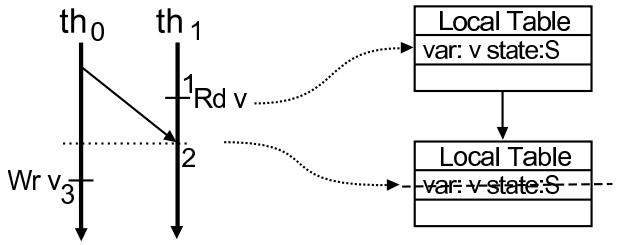


Figure 3.3: Source of inaccuracy: epoch ends before race takes place.

are contained overlap in time, or if they are "close by" in time. Interestingly, for every possible race in a program there must exist an execution in which the epochs of the racy instructions overlap in time. We provide a proof of this statement in Appendix A. As a result, given enough executions, AccB will not have any false negatives. We propose two optimizations to quickly expose races to AccB: choosing epoch boundaries more judiciously; and carefully perturbing the execution schedule to increase the probability of overlapping epochs with races. We explain these optimizations in detail in Section 3.4.3.

3.4. Implementation

3.4.1. Hardware support

We extend the ISA of an off-the-shelf multiprocessor with a StateCheck (`StChk off(basereg),reg`) instruction, which returns the state of `off(basereg)`'s cache block through register `reg`. If the block is not present in the cache, the instruction returns a special NotPresent (NP) state to distinguish this case from a present block in Invalid state. The last valid state is returned if the cache block is in a transient state.

To implement the StateCheck instruction, we need to make minor modifications to (1) processor control logic, (2) cache data paths, and (3) cache controllers. We modify the processor control logic to recognize the StateCheck instruction and communicate to the cache that the information requested is not the data associated to `off(basereg)`, but the state. We modify the cache with a multiplexer that creates a path for coherence state into the processor via the existing data path. Cache controllers require two modifications: (1) if the requested block is not currently cached, the cache controller returns the NP state without triggering a miss request, and (2) when it receives a request for the state, the cache controller retrieves the state of the corresponding block and asserts the multiplexer select bit to allow the state to flow to the processor.

So far we assumed L1 caches to be the point of coherence, but other configurations are possible. We can divide these configurations into two categories: (1) coherence is maintained among caches private to a hardware thread (*e.g.* private non-inclusive

L1 and L2 caches), and (2) coherence is maintained in caches shared by more than one hardware thread (*e.g.* SMT processor with a single L1 data cache). In the first case, the mechanism we propose works seamlessly: the state is obtained from the private cache where a hit happens (NP in case of a miss in all private caches). In the second case, accesses and resulting changes of state by different processors need to be distinguished by replicating the state for each thread.

3.4.2. Software Layer

Data structures: We associate a local table with each thread to record information about accesses performed during an epoch. These tables are hash tables indexed by data address and stored in main memory. Each entry contains the expected state (based on the type of the last access to the cache block) for the corresponding address. Entries also contain the address of the instruction that performed the last local access to the address. We want this tables to be in main memory because, although it is slower than having them in hardware, we want to avoid the bound in size that hardware resources imply.

Instrumentation points: We use dynamic binary rewriting to instrument every synchronization operation (thread creation and joining, mutex creation, acquisition and release, and conditional variable creation, waiting and signaling) and every memory operation not involved in a synchronization operation. Synchronization operations delimit epochs. We insert code right before the synchronization operation to search the local table for any downgraded variables in the ending epoch. We subsequently clear the table in preparation for the next epoch.

The instrumentation of memory accesses checks the current state of the corresponding address in the cache using a StateCheck instruction, and compares it with the state recorded in the table. If it detects a downgrade, it reports the race, along with the corresponding address and the instruction address of the previous access. Then, it updates the state in the table with the maximum (following the order $M > E = S > I$) of the recorded state and the current state. Note that using the maximum is safer than executing StateCheck again after the instrumented memory access executes because downgrades could be missed in the window between the instrumented instruction executes and the StateCheck instruction executes. Algorithms 3.1, 3.2, and 3.3 present the instrumentation code in more detail.

3.4.3. Optimizations

We introduce three optimizations: two to mitigate the early epoch ending problem discussed in Section 3.3.2, and one to cut down on the instrumentation overhead.

Algorithm 3.1 pre_sync()

```
for all ea ∈ Table do
    CurSt ⇐ StateCheck(ea)
    PrvSt ⇐ CurSt
    if CurSt ∈ {S, I} then
        PrvSt ⇐ Table[ea].St
    end if
    if PrvSt > CurSt then
        ReportRace(ea)
    end if
end for
Clear(Table)
```

Algorithm 3.2 pre_load(ea)

```
PrvSt ⇐ Table[ea].St
CurSt ⇐ StateCheck(ea)
if PrvSt > CurSt then
    ReportRace(ea)
else
    Table[ea].St ⇐ max{CurSt, PrvSt}
end if
```

Algorithm 3.3 pre_store(ea)

```
CurSt ⇐ StateCheck(ea)
if CurSt ∈ {S, I} then
    PrvSt ⇐ Table[ea].St
else[No downgrade, avoid table access]
    PrvSt ⇐ CurSt
end if
if PrvSt > CurSt then
    ReportRace(ea)
else
    Table[ea].St ⇐ M
end if
```

Coverage improvement: Redefining epoch boundaries

To mitigate the early epoch ending problem, we change how we define the boundaries of an epoch for AccB and only end an epoch when the local thread is the *source* of a synchronization operation (but not when it is the destination). To understand why this does not affect the correctness of our algorithm, consider how synchronization operations are used: when a thread is the source of a synchronization operation, it is typically communicating to other threads that it has modified shared data that can now be safely accessed; when a thread is the destination of a synchronization operation, it is being notified by another thread that it is now safe to access shared data that this other thread modified. However, unlike HapB, which needs to keep track of synchronization operations at the source and at the destination for proper ordering, AccB does not use the latter in any way. Thus, there is no advantage in ending an epoch when a thread is the destination of a synchronization. Moreover, extending the epoch further makes larger epochs, which mitigates the early epoch ending problem. As an illustration, the race missed in Figure 3.3 can be detected if thread 1 does not end the epoch upon receiving the synchronization. However, this optimization cannot be applied to HapB because HapB requires proper ordering of such epochs (vector clock updates).

HapB considers information about all threads. Therefore, when there is a synchronization operation between two threads, HapB records that synchronization in all involved threads. This is because HapB establishes that instructions previous to the sync operation in the *source* happen before those after the synchronization operation in the *destination*. Concerning Owned Before, we do not take into consideration what is happening in remote threads. The algorithm does not rely in knowing which epochs have "happened before" which ones, but it focuses on knowing when a remote thread requests data owned by the local thread. Thus, ending an epoch when the local thread is *destination* of a synchronization operation is not useful, because the meaning of it is that something has happened before, which is a piece of information our algorithm has no use for. On the other hand, being the *source* of a synchronization tells us that subsequent accesses to shared variables are likely to be synchronized, and thus ending the epoch is appropriate. By doing this we achieve two goals. The first one is reducing the overhead, because we will do less checks. The second and more important it to stretch the epochs, which alleviates the early epoch ending problem.

Coverage improvement: Schedule perturbation

To address the issue that AccB can only detect races between epochs that overlap in time, we randomly perturb program execution to encourage an increased variety of overlapping epoch sets. When an epoch starts, the instrumentation code randomly chooses one of two actions: (1) to continue executing normally, or (2) to join its thread to a rescheduling barrier. The thread waits at this barrier until a bounded random timeout occurs. At this point, all threads that joined this first barrier start executing their respective epochs. Once a thread finishes executing its epoch,

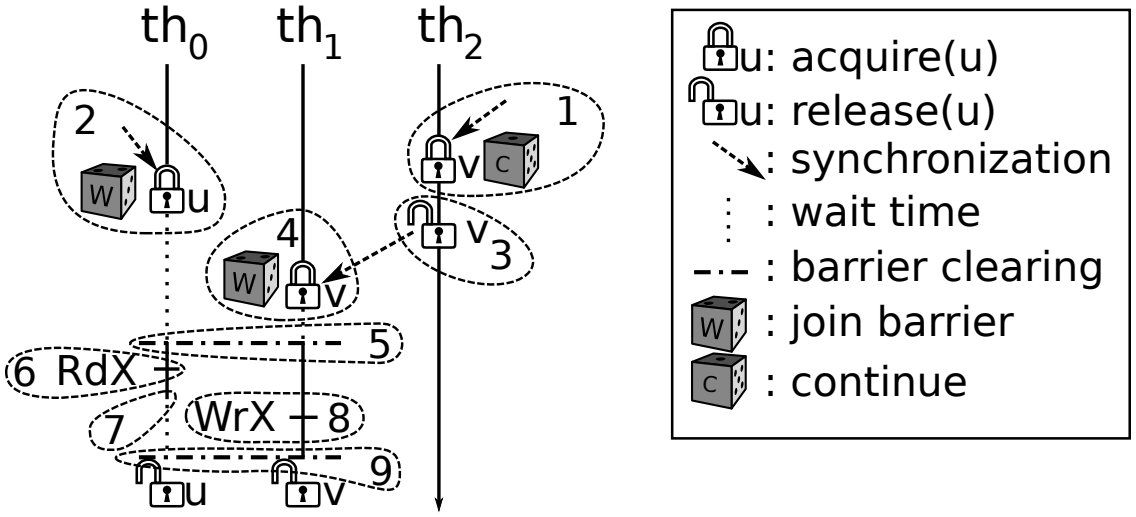


Figure 3.4: Schedule perturbation example

it joins a checking barrier. When all threads that joined the first barrier join this second barrier, or it times out, epoch checks are completed and all threads continue. We refer to this technique as probabilistic barriers.

Figure 3.4 illustrates the process with an example that builds on top of Figure 3.3, an execution without any scheduling disturbances that suffers from the early epoch ending problem. In Figure 3.4, thread 2 starts a new epoch by acquiring lock v (1). At this point, it is randomly decided that it will not join the wait barrier, so its execution continues normally. Thread 1 then starts an epoch by acquiring lock u (2), and it is again randomly decided that thread 1 will join the wait barrier. Eventually, thread 2 releases lock v (3), which is subsequently acquired by thread 0 (4). At this point, thread 0 also joins the wait barrier. Eventually a timer associated with this wait barrier triggers the beginning of all epochs in threads that joined the barrier (5). Thread 1 then reads variable X and completes its epoch execution (7). If no scheduling perturbation were being applied, thread 1 would perform all its checks and conclude the epoch then. With scheduling perturbations, it instead joins a second wait barrier. Thread 0 then writes variable X (8), so this write occurs in (logical) overlap with the read of X. Thread 0 ends its epoch, joins the second barrier, and both threads perform their checks (9). During its check, thread 1 observes the downgrade on X caused by thread 0 and finally detects the data race.

Unlike CHESS [46], which systematically explores all interleavings at the granularity of critical sections and runs race detection on every possible interleaving to achieve full coverage, AccB disturbs execution by randomly extending some of the epochs such that they fully overlap to address the early epoch ending problem, a problem specific to how AccB detects races, on a best-effort basis.

Further reducing overheads with extra hardware support (AccB++)

So far, we have presented how AccB works with minimal hardware support, *i.e.*, by only exposing the raw coherence state to the software layer. This enables us to implement an efficient data race detection mechanism with extremely simple hardware support. We can further accelerate this detection with a few still very simple modifications: we add a few bits to the state of each cache block and use them to reduce the number of accesses to the local table.

In addition to the regular coherence state, we store three extra bits per cache block, namely, locally read bit (*lrd*), locally written bit (*lwr*), and downgraded bit (*dgd*). These bits are used to record the nature of the last local access within an epoch (*lrd* or *lwr*) and downgrades (*dgd*) to a cache block touched by the local thread within that epoch. We also use an additional instruction that gang-clears these three bits in all local cache blocks, which is used by the software layer in the beginning of every epoch. We modify the cache controller to set *lrd* or *lwr* on a local read or local write access, respectively, and to set *dgd* on downgrades due to remote requests (but only when either *lrd* or *lwr* have been previously set). We extend the StateCheck instruction to return these three bits together with the regular coherence state.

The software layer uses these additional bits to detect accesses followed by downgrades within an epoch. A StateCheck instruction is inserted immediately before each memory access and the *dgd* bit is checked. If the *dgd* bit is set, a data race is detected. These bits optimize how the local table is used: we leverage the *lrd* and *lwr* bits to reduce the number of accesses to the table, since we only need to record information about the first read and write accesses to a variable in each epoch (this is sufficient to report one data race – others may be detected once the first is eliminated). As such, on every memory access, instead of checking the presence of the corresponding address in the table, we check the *lrd* and *lwr* bits. If none are set, this is the first access to this variable within the current epoch, so the address and the corresponding instruction address are added to the table. If only the *lrd* bit is set and the access being instrumented is a write, this is the first write access to the variable, so the instruction address of the table entry is updated. If the *lwr* bit is already set, no new updates are needed. Note this does not completely eliminate the use of the local table because it is still necessary for end-of-epoch checks and for recording the instruction address of accesses.

An alternative to adding bits to each cache block is to add a *global downgrade* register per cache, which can be combined with the per-block *dgd* bit or may substitute it entirely. This can be implemented either as a single-bit flag or as a counter. This register indicates whether any downgrades have happened since the beginning of an epoch (or how many downgrades have taken place, if a counter is used). We can use this information to avoid scanning the entire local table at the end of epochs if no downgrades have happened. We can use the counter to detect when to stop scanning the local table. This optimization is quite profitable, and is very easy to implement. It only requires one bit (or a k-bit saturating counter, where k is a small number, *e.g.* 4 bits – if the count saturates, the instrumentation detects it and scans the entire table).

If in the future, the footprint of programs grow making cache block evictions a source of inaccuracy, we can add a small victim cache next to the data cache. Whenever a block that has its *dgd* bit set is evicted from the data cache, it is cached in the victim cache. This allows AccB to report the race even if the block involved in the race has been evicted from the data cache.

3.4.4. System Issues

Thread migration. AccB relies in the coherence state of cache blocks stored within physical caches. When the system changes the thread-to-core mapping, the involved caches are flushed and the threads relocated, losing all the local coherence information. Since AccB only monitors information corresponding to the current epoch, this loss only affects race detection in that epoch. In addition, since the destination cache is also flushed, no false positives are produced due to migrating a thread with a recorded state "greater" than the state in the destination cache prior to the migration and flush. If flushes on every thread migration are considered too expensive or not done automatically, an alternative is to check, via an instruction like x86's CPUID, in which core the thread is running at every epoch end and compare it with the core identification number recorded in the previous epoch. If they are different, a migration has taken place and the instrumentation ignores any races detected for that epoch. Yet another alternative is to always return the NP state to StateCheck on epochs that follow context switches.

Speculation. Speculation can interfere with the proposed support and cause additional false positives in a few scenarios: (1) StateCheck is executed speculatively in a local core, (2) a load is executed speculatively in a remote core, and (3) a pre-fetch request is issued in a remote core. The simplest solution is to allow false positives, which are likely to be low. Other solutions to the first problem are to either reuse mechanisms traditionally used for load speculation (*e.g.*, replay or snooping) or to only set the access bit when the load retire. Solutions to (2) consist of limiting speculation to when it is safe. For example, allowing a speculative load to proceed only when it reaches the point-of-no-return in designs like CHERRY [47] or if the cache block is in the local cache in a valid state. (3) can be easily mitigated by turning pre-fetching off during test runs.

Shared caches. A more complex situation is when a cache is shared by more than a core. Our approach leverages the coherence actions taken when there is data sharing among caches. If a cache is shared by different cores, we cannot assume that there is a direct correspondence between data sharing and coherence actions. As an illustration, if two cores sharing the cache are accessing the same location, there will be no coherence action. Another problem is that downgrades can be *masked*. For instance, let threads th_0 and th_1 share the same cache, and th_2 use a different cache. If th_0 writes a variable v, then th_2 reads it, exposing a race condition, and downgrading the state of v in th_0 cache. Later on, th_2 synchronizes with th_1 which,

in turn writes v, setting its state in the shared cache to modified. The next time th_0 accesses v or ends an epoch it will check for races on v discovering none because the downgraded state has been re-upgraded by th_1.

Our algorithm is not designed for architectures in which the cache is shared among different cores, but it is able to overcome these two situations, with additional support. For AccB to detect races in threads executing in cores which are sharing the cache, the extra hardware support described in Section 3.4.3 must be available. Furthermore, it requires that the three status bits are replicated, one triple per sharing core. Also, to maintain the semantics of these bits, the cache is augmented to identify from which core each request originates. With this support, when a core probes the cache to inspect the current state it also gets the bits corresponding to the other threads, and thus, it can tell if the current access races with any of the accesses recorded in the status bits of the other cores. This replication is also enough to overcome the downgrade masking problem. When a the cache receives a coherence message that requires a downgrade, the cache controller inspects the status bits of every core, and sets the downgrade bit of all cores that should suffer a downgrade taking into account the value of the status bits instead of the current access. This way each core will know that there has been a downgrade when it probes the cache by looking at its own downgrade status bit.

Multi-level hierarchy The last issue we discuss is proving the support needed by AccB on a more complex cache hierarchy. If there are multiple levels of cache, AccB only monitors the coherence state in the lower level, because coherence actions in the lower level are enough to detect sharing. The only potential problem is that if coherence actions between upper levels are taken at a bigger granularity than the block size in lower levels, there can be downgrades resulting from false sharing, which can incur in false positives.

3.5. Related Work

As discussed earlier, there is a large body of work on data race detection, ranging from software-only tools to hardware mechanisms. Some of the notable examples of software-only tools are RecPlay [40], which does happened-before race detection; Eraser [41], which does lock-set violation detection; and more recently, RaceTrack [42], which is a simplified form of happened-before that reduces space-time overheads and FastTrack [48], which proposes an adaptive representation of vector clocks to reduce time overheads. FastTrack works well for managed languages. While much progress has been made, software-based race detectors still have high performance overheads and often very high space overheads.

Given the typical high cost of race detection in software, recent work developed hardware mechanisms to reduce performance overheads. Past work in this area focused on mostly-hardware solutions. For examples, HARD [38] is a hardware

implementation of the lock-set algorithm; SigRace [35] is a signature-based implementation of happened-before that uses speculative execution to reduce false positives; ReEnact [37] leverages support for thread-level speculation to detect and potentially dynamically correct data races. AVIO [49] is an atomicity violation detector that also leverages coherence state and several extensions to cache lines to determine undesirable interleaving. Note that AVIO only detects atomicity violations, not necessarily data races (atomicity violations do not necessarily imply data races). While some of this work reduces the overhead of bug detection significantly, they require a substantial amount of non-trivial hardware. Our focus is on absolute minimal hardware support to *accelerate* race detection, as opposed to implementing it in hardware. Atom-Aid [50] is also related to atomicity violations, but instead of only finding and reporting them, it also minimizes the probability that they manifest themselves by creating implicitly atomic blocks to prevent some interleaving.

There is another piece of work that relies in the coherence protocol to detect data races: Schimmel and Pankratius [51] propose TachoRace, which also leverages the implicit communication performed by the coherence protocol. The main difference between TachoRace and AccB is that AccB is more general, not focusing just on maintaining a locking discipline. Their proposal also requires some modifications to the coherence protocol, adding coherence messages, which AccB intentionally avoids.

Conflict exceptions [52] is related to our work in the category of races it detects. In this proposal, the system detects when synchronization-free regions (epochs) conflict with other concurrently running synchronization-free regions. Such conflicts can only happen when a data race exists. This is in essence the same type of event AccB detects. However, the goal of conflict exceptions is to detect these events in a fully precise manner and throw exceptions during runs in the field, which requires significantly more hardware (50% cache overhead for access bits) and complexity. We sacrifice some accuracy in order to require minimal hardware extensions. We also prove that AccessedBefore is guaranteed to find a race for some schedule, and show a scheduling manipulation heuristic that finds virtually all races with a reasonable number of executions.

To our knowledge, the works most related to ours are Min and Choi [34], and Nagarajan and Gupta [53]. Both pieces of work propose to expose certain cache coherence events in the form of software traps to enable analysis of parallel program behavior. Nagarajan and Gupta [53] showcased their mechanism with deterministic replay and barrier speculation. Min and Choi [34] developed a limited form of happened-before detection using coherence events for a subclass of programs with structured parallelism. In contrast, our proposal does not rely on software traps; it is essentially a *load* operation. Software traps are arguably more flexible, but are also more costly to implement and potentially harder to use due to the possibility of deadlocks. Finally, these proposals focus on other applications of tracking coherence events. We propose a new race detection algorithm that not only uses our novel hardware support to reduce performance overheads, but also significantly reduces space overhead compared to happened-before.

	Size	32 KB
	Ways	8
	Coherence Protocol	MESI
	Replacement Policy	LRU
Block Size	**Main results**	64 Bytes
	Accuracy tests	4 Bytes
	Sensitivity tests	16, 32 and 64 Bytes

Table 3.3: Cache configuration. *Sensitivity tests* refers to the evolution of false positives when the granularity at which the coherence information is kept grows.

Overall, our proposal is much lighter-weight than previous hardware mechanisms and it could be used to complement past software-only approaches when used as a filter.

3.6. Experimental Setup

We evaluate AccB using POSIX threads and the Pin [31] dynamic binary instrumentation framework.

3.6.1. Accuracy, space overhead and speedup characterization

In addition to the software layer we describe in Section 3.4, we model a detailed memory hierarchy, including cache coherence. Our model also includes the StateCheck ISA extension, as well as the optional hardware support described in Section 3.4.3. Table 3.3 summarizes the cache parameters used for our experiments.

We compare AccB with an implementation of HapB using the same instrumentation framework. For accuracy comparisons, both algorithms are applied simultaneously to the same run such that both observe the same interleaving of memory accesses. Since HapB is complete, the data races identified by AccB should be a subset of the ones detected by HapB. We verified this is the case for every run.

For performance measurements, we assume the latency of a StateCheck instruction to be the same as a load instruction that hits the cache, due to their similarity. When measuring performance of AccB++, we collect lrd/lwr/dgd bit distributions observed in our simulations that use a cache model (accuracy experiments) and use them to determine whether a table access is needed.

HapB Implementation. We have implemented a carefully optimized version of HapB using hash-sets for read- and write-sets. We also implemented a filter for the hash-set contents to speedup intersections using a 1024-bit Bloom-filter. Epochs belonging to the same thread are stored in an ordered linked list. Epoch information is pruned as soon as an old epoch has been ordered before the current epochs of all threads, so we have a space-optimal implementation of HapB for fair space overhead comparison. Vector clocks are implemented as regular arrays.

FastT Implementation. We have implemented a version of FastT according to the specification in [48]. As C and C++ don't have the flexibility provided by Java, that allows to embed extra information per-object we use the same tables used in our implementations of HapB and AccB to hold the metadata coupled to variables.

Benchmarks. To evaluate the effectiveness and performance of AccB we use a variety of workloads. First, we use the SPLASH-2 benchmark suite [54]. We run these benchmarks on 8 threads in an 8-core machine with the test inputs. We also use other applications that have bugs reported in the literature: AGet, PBZip [55], Apache httpd server and MySQL database server [49]. We run AGet with 8 threads that each download a file. We run PBZip with the *-dp8kf* option and two 1910-byte files as input, the same setup as in [55]. We do not report performance improvements for PBZip because its bugs lead to crashes that are non-deterministic in nature, so comparing running times is meaningless. AGet is a networking application with non-deterministic input timing so, again, the comparison is meaningless. Apache and MySQL run with the same configuration as in AVIO [49]. All benchmarks were compiled to the x86_64 architecture using the gcc compiler and the standard *-O2* optimization flag.

3.6.2. Comparison with commercial tools

We use the Intel Thread Checker version from the *Intel Inspector XE 2011 Update 7 (build 189290)* and the Helgrind version from the *valgrind-3.6.1* package obtained via the Debian package manager.

Given that we are running these workloads native and not performing any cache simulation, we have used PARSEC [56], a more current benchmark suite with larger inputs. It is compiled with *x86_64-linux-gnu-gcc-4.6.1* using the standard optimization flags included in the predefined configuration files, namely *-O3 -funroll-loops -fprefetch-loop-arrays -static-libgcc* for *C* code and *-O3 -funroll-loops -fprefetch-loop-arrays -fpermissive -fno-exceptions -static-libgcc* for *C++* code. We used one of the following supplied input sets: *sim-small*, *sim-medium*, *sim-large*. The initial comparison is done for 8 threads and the *sim-small* input set. The scalability on # of threads is done for 4, 8 and 16 threads again with the *sim-small* input set. Finally, the three input sets are used for the scalability study on input size (16 threads).

The aforementioned applications have been run to completion through the tools on an AMD Opteron 6172 48-core machine with 48 GB of main memory. For every combination of tool (including native execution), # of threads and input set we have performed 10 runs, and used the minimum. We checked that the variation of the obtained execution times is small enough to ensure that no external effects are artificially increasing or decreasing the run time.

	Speedup ($\frac{HapB}{[AccB, FastT]}$)			Space overhead (%)		Accuracy (%)
	AccB	AccB++	FastT †	AccB avg	AccB max	AccB
barnes	1.11	1.99	0.03	0.8	77.1	97.8
cholesky	1.03	1.31	0.01	9.4	13.0	–
fft	1.02	1.27	0.07	31.8	87.5	–
fmm	1.16	1.55	0.01	3.3	29.6	95.4
lu_{cnt}	0.98	1.54	0.08	19.9	33.3	–
lu_{ncnt}	0.95	1.23	0.08	20.2	33.2	–
$ocean_{cnt}$	1.19	1.19	0.33	25.6	110.6	100
$ocean_{ncnt}$	1.21	1.21	0.33	26.1	113.3	100
radix	0.98	1.04	0.18	32.9	116.5	–
raytrace	1.08	1.23	0.21	41.0	40.4	100
volrend	5.71	5.90	0.29	0.2	0.4	100
$water_{nsqr}$	1.23	1.71	0.08	15.8	25.7	–
$water_{spat}$	0.99	1.30	0.06	30.6	80.1	–
aget	–	–	–	0.1	0.1	100
pbzip	–	–	–	5.6	32.1	100

Table 3.4: Summarized comparison of AccB and HapB. The dashes (–) in the accuracy column correspond to those benchmarks that have not shown any races in any of the runs. In that istuation, reporting 100% accuracy could be misleading.

3.7. Evaluation

3.7.1. AccB versus HapB and FastT

To back up our claims that AccB is a competitive algorithm, we present a summarized comparison of AccB and HapB in Table 3.4 along three metrics: performance, space overhead and accuracy. We show the best HapB configuration for each metric – 64-byte granularity for performance and storage overheads and 4-byte granularity for accuracy. This gives all possible advantages to HapB. Even in face of these disadvantages, AccB compares favorably.

Table 3.4 compares AccB and HapB in terms of performance, space overhead and accuracy. The first group of columns in Table 3.4 show the speedup of an application instrumented with AccB, and with extra hardware support (AccB++), compared to HapB. For example, *barnes* instrumented with AccB runs 11% faster than when instrumented with HapB. The speedup grows to almost 2× with AccB++. Overall, AccB++ achieves speedups of up to almost 6×. A few benchmarks (lu, radix, and water spatial) experience modest slowdowns with AccB, caused by the type of synchronization used in these benchmarks: most synchronization is based on barriers, which allow HapB to clean up all information about old epochs and significantly reduce its checking overheads. AccB incurs extra overheads because it performs table checks on every memory access in addition to end-of-epoch checks. Note that AccB++ always shows speedups [1].

[1†] The results show that FastT is much slower than HapB. The reason is twofold: first, FastT experiences additional overheads compared to its original Java implementation due to the global table required by C++; second, HapB performs intersections at the end of each epoch, FastT

The second group of columns shows average and maximum space overheads for AccB, compared to HapB. For example, AccB uses only 0.8% on average and at most 77% of the storage used by HapB for barnes. For most benchmarks, AccB uses significantly less space than HapB. In some cases (ocean cnt, radix), AccB incurs a higher maximum space overhead compared to HapB (although the average space overhead is still lower). This is a pathological case caused by uncommon program behavior and our data structure selection for each algorithm, which results in a lower amortized data structure cost for HapB when barriers are frequent and accessed sets are large.

Finally, the last column shows the accuracy, *i.e.*, how many races are detected by AccB compared those observed by HapB. AccB is capable of detecting all races for most benchmarks. This demonstrates the effectiveness of our heuristics in exposing a wider variety of interleaving to AccB and allowing it to detect more races. Section 3.7.3 provides more insight into those very few races not detected by AccB.

3.7.2. Overheads Characterization

Performance. Table 3.5 characterizes the performance overheads of AccB and AccB++ compared to HapB, aggregated for all benchmarks. We did this study in a data structure independent manner by counting high level operations to each algorithm's internal data structures, *i.e., look-ups* and *updates*. The numbers show the relative frequency of events related to manipulation of internal data structures for AccB and AccB++, normalized to HapB. Look-ups (row 2) and updates (row 3) are direct accesses to AccB's local table and to HapB's sets. Branches (row 4) refer to branches taken while manipulating these data structures. AccB incurs many more look-ups than HapB because AccB performs look-ups at every memory access, while HapB performs them only at epoch ends. Even though AccB's look-ups are more frequent, AccB still incurs slowdowns lower than HapB. The reason is that there is high locality in AccB's table accesses and most hit in the cache. On the other hand, HapB is very control flow intensive, as demonstrated by the large number of branches we observe. In addition, HapB's data structures are larger (it maintains information about multiple epochs, not just the current), which results in worse cache behavior. Finally, HapB requires transferring vector clocks and epoch information, which implies additional communication among threads, *i.e.*, costly misses.

We now turn our attention to AccB++. With simple additional hardware support (i.e., three bits per cache block and a little logic), we were able to implement AccB at even lower overheads. AccB++ cuts down the number of look-ups by two orders of magnitude, an impressive reduction, even if many of those look-ups hit in the cache anyway. It also cuts down on updates by more than 60%. All this comes at the cost of a higher number of branches, although still much lower than HapB.

Space. Table 3.6 characterizes the space overhead of HapB and AccB. We count how many access-recording entries are used on average across all benchmarks and

performs checks at every access.

	AccB	AccB++
Look-ups	3326.7%	15.6%
Updates	97.0%	29.8%
Branches	4.2%	5.1%

Table 3.5: Number of operations executed by AccB and AccB++ compared to HapB.

	HapB	AccB
Avg. entries per epoch	360.3	623.2
Avg. epochs in history	16.5	0
Avg. simultaneous entries	71.6k	9.1k
Size (MB)	2.15	0.28

Table 3.6: Overheads, storage requirements of HapB and AccB.

report the number of entries per individual epoch (row 2), overall number of epochs kept in history (row 3), total number of entries used by all epochs in all threads simultaneously (row 4) and overall storage requirements (row 5). Note that these numbers are averaged over all benchmarks and there is a large variation across benchmarks.

The number of entries per epoch (row 2) shows that AccB records more entries than HapB for individual epochs. This is due to the optimization we propose in Section 3.4.3 for AccB, which makes epochs longer for AccB by not ending them when their thread is the destination of a synchronization. AccB does not keep any history while HapB keeps information about 16.5 epochs on average (row 3). When we compare total number of entries (row 4), we observe that AccB requires a much lower number of entries (less than 7× fewer). Overall, we again see a reduction of more than 7× in space overhead. These savings come from AccB only needing information about accesses in the current epoch. HapB keeps epoch information until an epoch is guaranteed to have been ordered before the current epochs of all threads, and this has a base cost per epoch due to the data structures used for keeping epoch information, in addition to the storage required by the access entries themselves. The storage requirements for AccB++ are nearly the same as AccB (the only difference is that AccB++ does not need the previous coherence state to be stored along with the variable address and instruction address). Note that, in addition to being larger, the storage required by HapB is also *shared* and accessed by all threads when their epochs end. Conversely, the storage required by AccB is much smaller and *purely local*.

3.7.3. Accuracy Characterization

False positives. In Table 3.7, we show the number of false positives detected by optimized AccB compared to HapB as we increase the cache block size. What stands out from this comparison is that AccB never has more false positives than

	16-byte	32-byte	64-byte
barnes	92.3	96.0	99.1
cholesky	100.0	91.7	98.3
fft	100.0	100.0	100.0
fmm	70.6	61.2	81.3
lu_{cnt}	100.0	100.0	100.0
lu_{ncnt}	100.0	100.0	100.0
$ocean_{cnt}$	100.0	100.0	100.0
$ocean_{ncnt}$	91.7	97.1	98.6
radix	100.0	100.0	100.0
raytrace	100.0	100.0	100.0
volrend	100.0	100.0	100.0
$water_{nsqr}$	100.0	100.0	100.0
$water_{spat}$	100.0	100.0	100.0
aget	7.7	34.4	51.8
pbzip	0.0	0.0	75.0

Table 3.7: Percentage of false positives in AccB compared to HapB. Each cell represents $100\frac{\#\ of\ false\ positives\ AccB}{\#\ of\ false\ positives\ HapB}$. Lower is better

HapB. These false positives are inherent to the tracking granularity (cache blocks). We believe they can be reduced with additional software support (*e.g.*, by changing the data layout to avoid false sharing), but this is beyond the scope of this thesis.

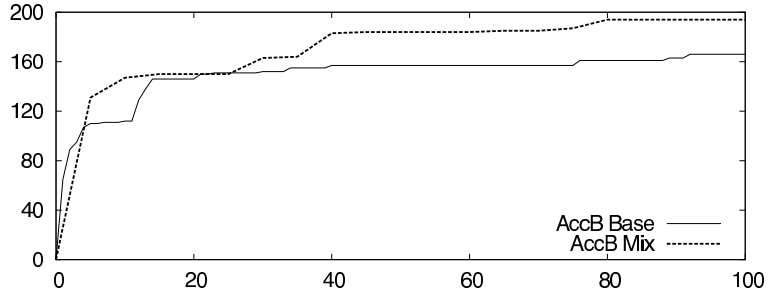
False negatives. As explained in Section 3.3.2, AccB has two sources of false negatives (*i.e.*, missed races). The first one is due to the limited capacity of caches, which causes cache blocks containing downgrade information to be evicted and the information to be lost (*CBE* – cache block evictions). The second is due to epochs containing the first access involved in a race finishing before they have a chance to observe the downgrade (*EEE* – early epoch ending). We separate the two effects by modifying our simulator with unbounded space to store evicted cache blocks, such that the CBE problem is completely eliminated. The reduction in races from a regular cache to a cache augmented with unbounded space gives us the number of races that go undetected due to the CBE problem. We observed no difference in our results, thus all races missed by AccB for these benchmarks are due to the EEE problem.

EEE optimization. Table 3.8 shows the effect of ending an epoch only when a thread is a synchronization source. It reports the relative improvement in data race detection accuracy when the optimization is used compared to ending epochs at every synchronization point. Improvements vary widely, but the optimization never hurts accuracy. In addition, it never slows down execution by more than 5% (typically 2% slowdown to some speedup).

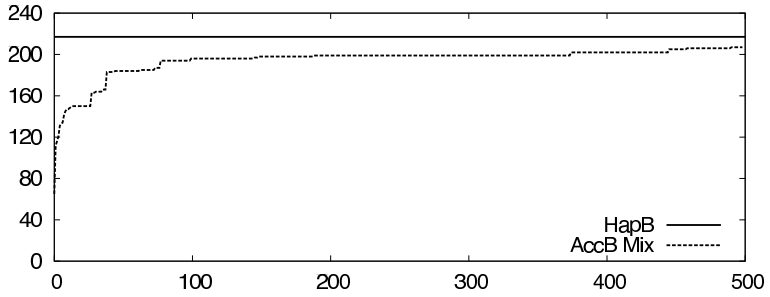
Sensitivity to scheduling perturbations and number of runs. Figure 3.5(a) shows how the aggregate number of static races detected by AccB, with (AccB Mix) and without (AccB Base) scheduling perturbations, grows with the number of executions for *fmm*. Initially, it is not clear which approach leads to the most detected static races. However, after about 25 runs, AccB Mix clearly shows new races

	barnes	*fmm*	*ocean*		*raytrace*	*volrend*	*aget*	*pbzip*
			cont	*ncont*				
(%) Accuracy improvement	4.0	62.0	0.0	0.0	0.0	0.0	38.0	203.0

Table 3.8: Accuracy improvement with EEE optimization for benchmarks with races, calculated as the difference between detection rate qith optimization on and off.



(a) Aggregate number of static races found as the number of executions increases for AccB (AccB Base) and AccB with scheduling perturbations (AccB Mix).



(b) Aggregate number of static races found as the number of executions increases for AccB Mix, compared to HapB.

Figure 3.5: Sensitivity to scheduling perturbations and number of runs.

while AccB Base does not. This happens when the scheduling perturbations start exposing more diverse epoch overlaps. These results also show that scheduling perturbations indeed help AccB find races faster.

Figure 3.5(b) shows how fast AccB Mix approximates the number of static races detected by HapB throughout 500 runs (horizontal line). AccB detects most races in the first few executions (about 2/3 are detected within the first 10 runs). The number of races AccB Mix detects continues growing after that, although increasingly more slowly.

We manually inspected a few of the races that AccB had not detect after 500 runs and found that for each undetected race there was another race that originated at the same programming mistake (*e.g.*, missing critical section) and that was successfully detected by AccB.

Apache and MySQL. Due to their nature, defining an experiment for reliably

	AccB	Helgrind	InspeXE
blackscholes	55.7	150	326.3
bodytrack	196.5	265.5	34.4
canneal	8.8	101.9	34.4
dedup	22.1	545.3	43.6
ferret	130.9	325.9	121.5
fluidanimate	162.2	6845.8	291.7
raytrace	2.9	41.2	7.3
streamcluster	109.2	–	651.6
swaptions	59.1	172.4	343.6
vips	141.1	1189.7	135.1
*x*264	221.2	805.2	340.1
Geometric mean	58.2	357.0	136.2

Table 3.9: Slowdown for $\#threads = 8$ and *sim-small* input. The three algorithms are compared to native execution. The application *streamcluster* doesn't finish when instrumented with Helgrind.

measuring performance and space overhead for these applications is challenging. Therefore we have just tested whether AccB detects the races in these applications reported in [49] (Apache#1 and MySQL#3). Due to the nature of this bugs, there are only two possible outcomes, either the race is detected (success), or the race is not detected (failure). Both tests succeeded.

3.7.4. Comparison with Commercial Race Detectors

Finally, we show how AccB performs when compared with current commercial race detection tools. We only compare them with AccB (not AccB++) because it represents the simplest hardware support and serves as a lower bound (AccB++ is strictly faster than AccB).

Table 3.9 shows the slowdown experienced by AccB and the other two detectors when compared to *native* execution. AccB outperforms Helgrind for all benchmarks and InspeXE for all benchmarks except ferret and vips. On average, the slowdown incurred by AccB is slightly half the slowdown experienced by InspeXE and a fifth of that experienced by Helgrind.

The only benchmarks for which AccB does worse than InspeXE are *ferret* and *vips*. *Ferret* is a content based similarity search that works as a 6-stage pipeline: *load, segment, extract, vec, ran* and *output*, the first and the last being sequential stages. N Threads are spawned for the remaining four stages. On every stage, threads iterate over a queue, extracting totally independent items to process and enqueue for the next stage. Given the balance in the input sizes: 256 inputs, $avg\,size = 3.58KB$, $cv = 0.74$, all threads synchronize through the queue quite often, and thus the history is cleared, making the advantage AccB has when history is a few epochs long insignificant. Additionally, the ratio of synchronization operations to memory operations for *ferret* is low, so AccB, which instruments memory operations

	4 threads			16 threads		
	AccB	Helgrind	InspeXE	AccB	Helgrind	InspeXE
blackscholes	47.8	110.0	502.5	77.0	234.0	995.5
bodytrack	166.2	179.0	199.9	249.7	851.9	461.4
canneal	18.1	80.9	38.8	6.2	142.7	42.6
dedup	33.4	500.4	48.6	16.6	545.0	57.0
ferret	131.6	317.5	157.9	128.2	328.7	151.1
fluidanimate	60.2	802.2	139.6	381.8	15292.3	691.5
raytrace	3.0	41.1	7.4	2.8	41.6	7.5
streamcluster	171.5	–	234.6	102.2	–	986.3
swaptions	53.4	97.4	200.4	68.0	342.0	860.7
vips	144.8	695.0	110.8	129.6	1662.2	163.4
x264	193.5	620.0	356.7	217.6	797.1	397.8
Geometric mean	59.4	226.2	117.3	62.1	520.9	216.3

Table 3.10: Slowdown for 4 and 16 threads, compared to native execution. The application *streamcluster* doesn't finish when instrumented with Helgrind.

more heavily than InspeXE, ends up experiencing a slightly higher slowdown. *Vips* behaves similarly. It breaks the input image into independent chunks that are processed by different threads, with a master thread creating the chunks and assigning them to other threads. This main-thread-centric synchronization again makes per-thread histories short, which, along with similarly low synchronization to memory access ratios, prevents AccB from outperforming InspeXE.

Even if under-performing for two benchmarks, AccB, a roughly tuned academic implementation, performs quite well overall compared to InspeXE, a commercially available tool developed by a major microprocessor manufacturer.

Scalability with number of threads: We also show that AccB scales better than InspeXE and Helgrind as the number of threads increases. This is because AccB maintains all the required information locally to each thread, so increasing the number of threads does not increase the overhead incurred by the algorithm. Table 3.10 shows slowdowns relative to native execution for 4 and 16 threads to illustrate this point.

The performance of AccB shows little variation for some applications such as *ferret*, *raytrace* and *x264*. Even more interestingly, the slowdown decreases as the number of threads increases for some benchmarks. The reason is twofold. First, the overhead is only introduced in the parallel sections making them slower, and thus reducing the weight of the sequential part. Therefore, a simple application of Amdal's Law shows that if the parallel sections execution time were to be cut in half, this optimization would translate into a higher overall speedup for the instrumented tool than in the plain run because the sequential part is a smaller part of the program. This is specially important in *raytrace*, because loading all the geometric data (sequential) takes much longer than rendering the image (parallel). Second, increasing the

	$4 \rightarrow 8$	$8 \rightarrow 16$
streamcluster	0.72	0.42
ferret	1	0.95
raytrace	1.00	0.99
canneal	1.05	1.02
dedup	1.05	1.02
x264	1.3	1
blackscholes	1.33	1.5
bodytrack	1.38	1.3
fluidanimate	1.58	1.51
vips	1.63	1.37
swaptions	1.90	1.83

Table 3.11: Speedup for the baseline run when the number of threads is doubled from 4 to 8, and from 8 to 16. The application *streamcluster* doesn't finish when instrumented with Helgrind.

number of threads implies that threads have a smaller chunk of the input set to process, which indirectly translates into less pressure on memory structures and faster access to data structures. Although Helgrind and InspeXE also show good scalability for these applications, (*raytrace* for example), they scale worse. On average, the slowdown of AccB grows only 4.6% when going from 4 threads to 16, while it grows 84.2% for InspeXE and 130.31% for Helgrind. These and the previous results show that it is beneficial to avoid inter-thread communication and to expose information already tracked by the coherence protocol to the software layer.

Table 3.11 shows the speedup for plain runs (native run, no instrumentation) for the benchmarks. In the following rationale "speedup" refers to the speedup achieved by increasing the number of threads in the native run, and "slowdown" and "overhead" refer to the slowdown incurred by AccB when compared to native run for a given number of threads. There is a general trend to reduce the slowdown in those benchmarks in which the gain is low: *canneal*, *dedup*, *ferret*, *raytrace* and *streamcluster*. For the applications that experiment a moderate speedup for the baseline (*blackscholes*, *bodytrack* and *x264*), the slowdown of AccB grows, but not much. Finally, for the applications that achieve a big speedup in the baseline, AccB shows the 3 possible behaviors. In the case of *fluidanimate*, the instrumented version is unable to execute any faster when the number of threads is increased. This is because the number of synchronizations executed per thread does not vary, and is quite large (1,1M epochs), and the average table size is also quite constant, yielding a similar per-synchronization overhead. *fluidanimate* is a particle simulator that assigns a chunk of the space per thread, and there is a step in which each chunk checks its neighbors requiring extra synchronization. The more threads there are, the more extra synchronization is required (in a 2x2 grid – 4 threads–, each tile has 2 neighbours, in a 2x4 grid – 8 threads – 4 tiles have 2 neighbors, but the other 4 have 3, and finally in a 4x4 grid –16 threads– 4 tiles have 4 neighbors, 8 have 3 and 4 have 2). This on top of the barrier at the end of the step, that makes all threads to wait for the slowest one, increase the execution time making the slowdown incurred grow dramatically. For *swaptions*, the slowdown incurred by AccB is stable. This is because AccB doesn't prevent the application to scale performance. This

	sim-medium			sim-large		
	AccB	Helgrind	InspeXE	AccB	Helgrind	InspeXE
blackscholes	25.2	205.4	528.9	11.5	182.2	405.3
bodytrack	199.5	1296.9	401.1	208.0	1766.2	388.6
canneal	10.8	242.3	41.9	25.8	409.8	40.0
dedup	24.5	634.3	82.9	39.0	849.4	121.8
ferret	95.6	710.5	134.0	123.8	1274.1	162.3
fluidanimate	417.5	7678.8	744.5	317.1	2680.0	693.4
raytrace	2.8	45.8	8.0	3.1	53.2	10.2
streamcluster	217.6	–	881.4	416.8	–	646.8
swaptions	54.7	343.3	673.8	53.9	328.8	604.6
vips	143.0	2644.0	127.0	144.0	2923.5	92.6
*x*264	177.9	2131.9	348.0	158.9	2616.1	449.6
Geometric mean	61.3	675.6	195.0	68.4	742.8	193.6

Table 3.12: Slowdown for 16 threads compared to native execution, for sim-medium and sim-large input sets. The application *streamcluster* doesn't finish when instrumented with Helgrind.

happens because in *swaptions* threads don't communicate, and therefore they don't synchronize, also the size of the table per thread is quite close, incurring the same overhead per access. Finally, *vips* experiments a overhead reduction. When the number of threads is doubled, each thread synchronizes around 30% less and the table also reduces size, decreasing this way the overhead both in each memory access and in each synchronization, allowing for better scalability. The case of *streamcluster* deserves special attention, because the baseline loses performance when more threads are used. Taking a look to the number of atomic memory instructions performed per synchronization as a measure of the contention, it almost doubles with the number of threads. This means that there is a serialization produced by the fine granularity of the synchronization, that in the end leads to a performance loss instead of performance gain. In the runs instrumented with AccB, given that the overhead elongates the execution time of the epochs, this contention is not exposed, therefore performance gain is experienced when more threads are used, which leads to overhead reduction.

Scalability with input set size: Table 3.12 shows the slowdown experienced by AccB, Helgrind and InspeXE with the *sim-medium* and *sim-large* input sets. InspeXE scales slightly better than AccB as the input set size grows, mostly due to two reasons. First, increasing the input size does not necessarily increase the amount of inter-thread communication, which is what favors AccB over InspeXE and Helgrind. Second, although AccB has been tuned to a certain degree and it is implemented using the C++ Standard Template Library, InspeXE is a commercial tool that has been under development for long time, and that uses data structures specific to the task, which do not suffer as much with large inputs (unlike AccB, as mentioned in Section 3.7.1).

Generally speaking, AccB scales well, increasing slowdown significantly only for *canneal* and *streamcluster*. For the rest of the benchmarks, it either (1) stays stable (*e.g.*, *raytrace*) because the input increase simply generates more epochs with

roughly the same data footprint, which is largely the factor determining the slowdown; or (2) makes a rebound in one of two directions. For *bodytrack* and *ferret*, the slowdown goes down for *sim-medium*, but then grows for *sim-large*. This is because growing the input affects the balance of synchronization and memory operations and hurts native execution more for *sim-medium*, but rebounds for *sim-large*. For *fluidanimate*, the effect is the opposite for AccB and InspeXE (and there is a dramatic slowdown decrease for Helgrind). *Fluidanimate* is a particle simulator that partitions the space in almost-independent plots. Synchronization is mainly barrier-based, although for those particles that cross borders some mutex-based synchronization exists. When going from *sim-small* to *sim-medium*, the data footprint accessed in each epoch grows, leading to larger slowdowns, but when going to *sim-large* particles move plots often, requiring mutex-based synchronization. This extra synchronization allows threads to clear history faster, effectively reducing the epoch length and therefore the size of the data structures added to keep track of accesses, speeding up insertions and look-ups. Helgrind really takes advantage of these reductions, largely reducing its slowdown. InspeXE and AccB also take advantage of this effect, although the extra synchronizations take a toll on InspeXE, which does not experience as much slowdown reduction as AccB with *sim-large*. Overall AccB scales well with the input size, slightly increasing its slowdown (10.09%) when going from the *sim-small* to the *sim-large* input set.

3.8. Conclusions

Efficient data race detection is instrumental in supporting safe multithreaded programming. This chapter introduces a new race detection algorithm that can be accelerated significantly via a simple instruction that exposes coherence state. Our experiments show that our approach has significantly lower space and time overheads and achieves virtually perfect accuracy. We also proposed minor hardware extensions that further reduce performance overheads to levels that have the potential to be fast enough to be continuously active. Our low-complexity hardware mechanisms can be easily adopted in a processor design.

Our experiments also show that with this hardware support, a first implementation our a data-race detection algorithm challenges two commercial tools that have been used and enhanced over the years. In addition, our evaluation shows good scalability results: the performance overhead of AccB remains stable or decreases as the number of threads and input set size grow for the majority of the workloads, and on average.

Chapter 4

Emerging Memory Technologies

This chapter addresses the bottom of the memory hierarchy. In this third and last piece of work we propose *Compression for Endurance in PCM RAM (CEPRAM)*, a technique to elongate the lifespan of PCM-based main memory through compression.

We introduce a total of 3 compression schemes based on existent compression schemes, but targeting compression for PCM based systems. We do a two-level evaluation. First, we quantify the performance of the compression, in terms of compressed size, bit-flips and how they are affected by errors. Next, we simulate those parameters in a statistical simulator to study how they affect endurance of the system.

This chapter is arranged as follows. Section 4.1 is an introduction of resistive memory technologies. Section 4.2 provides some background. Section 4.3 presents the technique in detail. The state of the art is discussed in Section 4.4.In Section 4.5 we show the experimental methodology and environment, and next, in Section 4.6 we show the results yielded by that evaluation, both in isolation and comparing it to previous proposals. Finally, Section 4.7 concludes.

4.1. Introduction

It has been some years since the dead of DRAM as main memory technology was foreseen. DRAM faces problems of scalability beyond 30nm, and power [3]. These two problems have led research efforts towards new technologies as PRAM, STT-MRAM [57] or FRAM [58], that are promising as a further-scalable replacement for DRAM. Among these technologies, PRAM is has been the most appealing to researchers, for it is the closest to commercialization.

In DRAM, information is stored as a charge, or the absence thereof, in a condenser. Condensers need to be charged or discharged to be written. Also, writes are destructive: in order to read a cell, it is discharged to check if it produces any current, making necessary for the cell to be re-written after every read. Also, the retention

time is not very long, making it necessary to refresh all the stored values periodically. This has implications in power that we would like to overcome. On the other hand, PRAM is a resistive technology. PRAM cells consist of a chalcogenous material that can swap from crystalline to amorphous state fast enough, in the order of $\sim 1\mu s$ which translates to around 680 CPU cycles [59]. This material is attached to a heater that melts the material and then cools it to either amorphous or crystalline state. This two states have different electrical properties, electrical resistance among them, therefore, by driving a current through the material we can read the state.

Spin-transfer torque memory (STT-RAM) is a kind of magneto-resistive RAM technology that exploit spin-transfer torque, as the mechanism for writing. Cells, called spin-valves, are made of two ferromagnetic plates separated by an insulating layer. One of these two layers holds a fixed magnetic field, while the other is changed to hold the logic value. Values are read measuring the electrical resistance of the cell, that due to the magnetic tunnel effect, is different depending on whether the magnetic field of both plates accord or differ. There are different kinds of MRAM depending on how the magnetic field of the "writable" plate is "written". In an electrical current, the carriers have a physical property called spin, which is a measure of the angular momentum intrinsic to particles. If an electrical current has the same amount of spin-up and spin-down carriers, it is considered non-polarized, and it is considered polarized otherwise. Spin-polarized currents have the ability to transfer the angular momentum into a magnetic layer when they go through, thus setting the orientation of the magnetic layer.

FRAM is much quite like DRAM, but it achieves non-volatility by including a ferromagnetic material in the dielectric condenser. The process of writing is similar: a field is applied across the ferroelectric layer by charging the plaques on either side of it, forcing atoms inside into the "up" or "down" orientation. Reading, however, is different, much more like it was with ferrite core memories: The transistor forces the cell into a particular state, let's say "0". If the cell already held that value, nothing happens in the output, but if the cell held a "1", the transition produces a small magnetic field that induces a current in the output line.

PRAM is the candidate receiving the most attention, be it because it is compatible with CMOS process, because it can be scaled down beyond 20nm, or because it does not require a periodical refresh. For those reasons, many researchers have tackled the problem of short endurance from a variety of aspects. For example, doing wear leveling, to avoid early failures in hot-spots of the memory, or building a hybrid hierarchy, placing a DRAM based last-level cache over a PRAM based main memory.

4.2. Background

There are things that are required to understand our proposal. This section introduces the concepts of *Phase Change Memory* and coding theory, doing special

emphasis in compression, and the concept of entropy, key in the motivation of the technique.

4.2.1. Phase Change Memory Technology

Phase Changing Memory (PCM) is a memory technology that uses the electrical properties of a material to store memory[60]. More precisely, it uses the change in the electrical resistance of materials when they are transformed between an amorphous state and a crystalline state. The idea is not new, it first appeared in the 1960s, but it has been on the last years, with the use of chalcogenous alloys as $Ge_2Sb_2Te_5$ (or GST for short), when it has gained popularity, and some factories have already made prototypes.

Figure 4.2 depicts a PRAM cell, which consists of the two electrodes enclosing a heating element, and the chalcogenous material. The heater element is just a material that produces Joule heat when a current is driven through, warming the chalcogenous material.

In Figure 4.1 we can see how the writing process works, in terms of time and temperature. By applying a fast, high-intensity current pulse the material reaches over 600^oC and melts. Then, it is cooled down quickly, making it amorphous. If the pulse is longer and with lower intensity, the material goes though an annealing process allowing the molecules to re-crystallize, lowering the electrical resistance. The process for reading the stored value consist in applying a small current to the cell to measure the resistance of the cell.

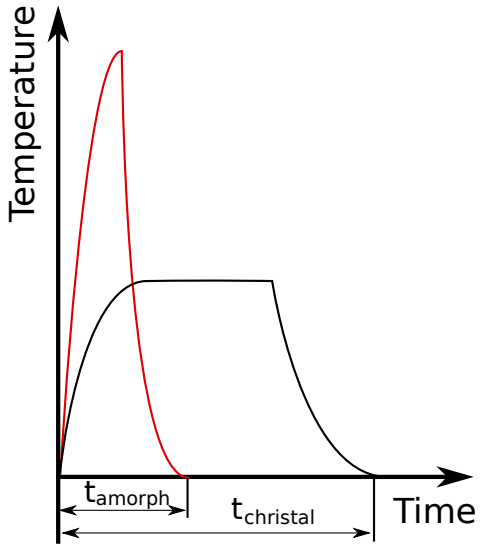


Figure 4.1: Heat pulses used to set and reset a PCM cell.

The limitations of PCM as a replacement for DRAM are the higher write latency, and the limited write endurance. Next-generation PCM devices can endure just 10^7-10^9 writing cycles [61]. The continuous expansions/contractions of the cell produced

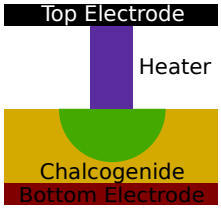


Figure 4.2: PCM cell. A heating element (purple) is attached to a chalcogenous material (yellow/green), and enclosed between the two electrodes. The bit of material attached to the heater forms the programmable volume(green), *i.e.* the part of material that will experiment the phase change.

by write operations result in a detachment of the heater from the cell, leaving the cell in a *stuck-at* failure state, from that moment on the cell is still readable, but the value cannot be changed anymore.

On the other hand, PCM has some features that DRAM lacks. One of those good features is the fact that there are intermediate states between amorphous and crystalline. This states can be differentiated, allowing for multi-bit cells[62]. Another feature is that, not being a charge based technology as DRAM, PCM is not affected to particle-induced errors [2, 63].

4.2.2. Coding theory

Error-correcting codes have been used to overcome errors in computer systems for a number of years. Any code $\mathcal{C}$ has 4 defining aspects:

- The **alphabet**: $\mathcal{A}$ is the set of symbols that form words of the code.
- The **length**: n is the number of symbols from $\mathcal{A}$ that form the codewords, this is, $\mathcal{C} \subset \mathcal{A}^n$.
- The **size**: k is the amount of different codewords in the code: $k = |\mathcal{C}|$.
- The **distance**: d is the minimum of symbols that differ between any two words of the code. The number of errors that $\mathcal{C}$ can correct is $t = \left\lfloor \frac{d-1}{2} \right\rfloor$.

For a binary code, *i.e.* $\mathcal{A} = \{0, 1\}$, there are some inequations that relate the length, the size and the distance of $\mathcal{C}$. One such inequation is the Hamming Bound that states:

$$k \leq \frac{2^n}{\sum_{i=0}^{t} \binom{n}{i}}$$

or an equivalent form used in [64]:

$$s_{min} = \log_2 n - \log_2 k \geq \left\lceil \log_2 \sum_{i=0}^{t} \binom{b+t}{i} \right\rceil$$

Where s_{min} is the lower bound of the space overhead required to correct t failures in a code of size $k = 2^b$.

Codes were devised to be applied in a context in which you want to keep the amount of data sent/stored low, but you can afford some extra symbols in those cases that happen rarely. In our context we have a physical limitation in the amount of symbols, and requiring more symbols than we actually have turns into a non-recoverable failure.

Compression: Is a important part of coding theory. Compression is an application of coding theory that tries to transform symbols, or collections of them, into symbols of a, maybe different, output alphabet, such that the frequent cases uses the least amount of symbols as possible. Common techniques used are ordering symbols by frequency and encode them as new strings such that the common ones require less symbols than the uncommon ones (Huffman encoding) or storing in a dictionary symbols/strings as they are processed to encode subsequent occurrences as a backward reference. This achieves a reduction in the amount of symbols required to store/transmit information, in the average case.

There have been some proposals about using compression at different points in the memory hierarchy[65, 66, 67], but they had in mind speeding up bus transactions or virtually expanding the available space. In this proposal, we have in mind using the healthy bits of a block to store the result of compressing the data in the block. This requires the data to be compressible in, at most, as many bits as we have remaining healthy in the block. The entropy as a measure of the information conveyed by a word gives some insight of how well data will compress, regardless of the compression scheme used.

4.2.3. Entropy as a measure of compressibility

In information theory, entropy is a way to estimate how much information is conveyed by a symbol, a word or a whole text. Entropy is related with the probability of the symbols. For example, if we have $\mathcal{A} = \{0, 1\}$, and we consider the word "00000010000100000010", the probability of a symbol in the word to be 0 is $\frac{17}{20}$ and the probability of 1 is $\frac{3}{20}$, in this sense, the statement *the $n^t h$ symbol is a 1* conveys much more information than the statement *the $n^t h$ symbol is a 0*, because 0 is the most likely value. The existence of this imbalance lowers the entropy, and we could just store the positions of the 1s to compress the word. On the other hand, if we take a look at the word "01100111001100110010", $P(0) = P(1) = \frac{10}{20} = \frac{1}{2}$, this makes the symbols totally unpredictable, and thus the word is not compressible.

Average	Total	Meaning
low	low	Both the symbols and the language are regular ⇒ Highly compressible.
low	high	Although the language as itself is not regular, words are, and word-level compression works fine.
high	low	Words are not compressible, but the language is ⇒ It is possible to encode symbols in a different way, so words are still 64 symbols but each symbol smaller in size.
high	high	Compression is unlikely to work well.

Table 4.1: Meaning of the different possible values of Total and Average entropy.

The formula to calculate the entropy of a language/word, L, from a $q-$ary alphabet, this is, $|\mathcal{A}| = q$ is:

$$E(L) = -\sum_{s \in \mathcal{A}} (p_s \cdot \log(p_s))$$

Where p_s is the probability of s appearing in L. If the base for log is 2, the unit for $E(L)$ is $bits/symbol$.

In the following we present two different values for the entropy of applications from the suite SPEC2006. We consider byte symbols: $\mathcal{A} = \{0x00, 0x01, ...0xff\}$, and the words are cache blocks, that in our target architecture are 64-byte.

- Average Entropy: This value is the entropy of all blocks that are evicted from LLC and written back to main memory averaged. It hints about the regularity of the symbols in a word or the absence thereof. Small values mean that a few symbols are repeated in the same word, and there are a few symbols that only appear a small number of times. In contrast, big values mean that there is not predictability in the symbols of a word, because most of the symbols differ from one another. Given that words are 64 symbols long, the entropy ranges from 0 to $6 = \log_2(64)$.
- Total Entropy: This value is the entropy of the language formed by all blocks (the multiplicity is the probability of each word) evicted from LLC and written back to main memory. It gives information about the whole data footprint. Small values means that words are formed, mainly, by a small set of symbols that appear in most of the words, although it doesn't tell us anything about how many times these symbols appears inside a word, to that end we have the Average Entropy. A big value means that the footprint is not regular. Since $|A| = 256$, this value ranges from 0 to $8 = \log_2 256$.

Table 4.1 shows the meaning of Average and Total entropy when considered together. The best case is to have a low Average entropy, because that means compression goes well, and we can stick to it. If Average entropy is high but Total entropy is low, programs would need to go through a *common symbol extraction*

phase, then a function $f : \mathcal{A} \rightarrow \cup_{n=0}^{N}\{0,1\}^n$ such as Huffman encoding built to re-encode the alphabet to achieve symbol-level compression. Although it is possible, the hardware resources to support this are larger.

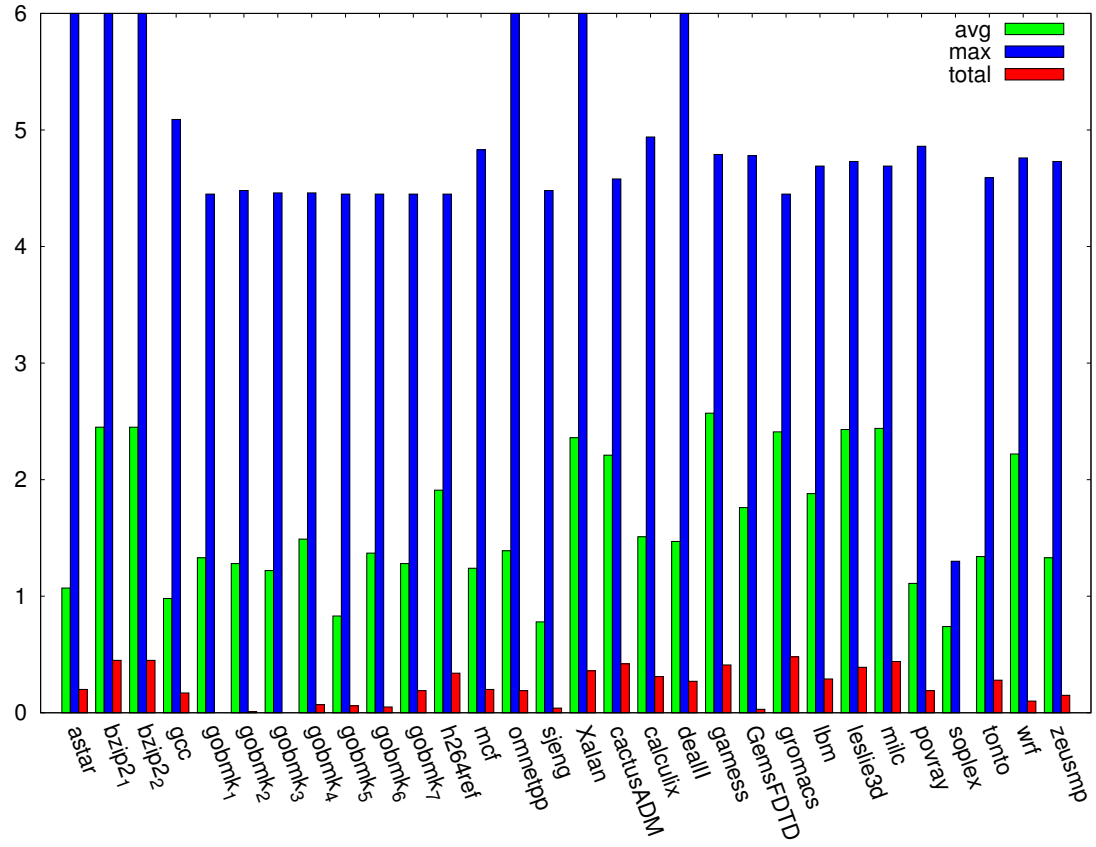


Figure 4.3: Average (green), Max (blue) entropy per block, and Total (red) language entropy for a 2MB LLC.

Figure 4.3 holds the values for Average, Max and Total entropy for the applications from the suite SPEC2006. Table 4.2 contains the same information for different LLC sizes. Max entropy refers to the maximum entropy found among the evicted words. If it is 6 means that at least one block in which all the bytes were different is evicted from the LLC. This number gives a bound for the worst case, and it is interesting the fact that for a big amount of benchmarks this number is lower than 5, meaning that there is more than 16.6% redundancy in each word. Concerning Average Entropy, the results are promising, even for floating point applications, that have much more variability than integer application. For all applications and cache sizes but 4, the Average Entropy is smaller than 2.5, what hints for good compression rates.

Concerning Total Entropy, the values are really low, this is due to the large amount of zeroes that are written to memory. This values vary slightly with the cache size. This is because as the LLC size grows, the amount of write-backs decreases and the statistical variety with it. The lack of enough statistical variety may bias the

	Average			Max			Total		
	1M	2M	4M	1M	2M	4M	1M	2M	4M
astar	1.13	1.07	1.17	6	6	5.89	0.26	0.2	0.22
$bzip2_1$	2.4	2.45	2.54	6	6	6	0.44	0.45	0.46
$bzip2_2$	2.4	2.45	2.54	6	6	6	0.44	0.45	0.46
gcc	1.03	0.98	1.01	5.09	5.09	5.04	0.17	0.17	0.17
$gobmk_1$	1.33	1.33	1.33	4.46	4.45	4.45	0.00	0.00	0.00
$gobmk_2$	1.21	1.28	1.22	4.47	4.48	4.45	0.01	0.01	0.01
$gobmk_3$	1.16	1.22	1.12	4.45	4.46	4.46	0.00	0.00	0.00
$gobmk_4$	1.49	1.49	1.47	4.46	4.46	4.46	0.07	0.07	0.06
$gobmk_5$	0.81	0.83	0.79	4.45	4.45	4.45	0.07	0.06	0.06
$gobmk_6$	1.43	1.37	1.52	4.45	4.45	4.45	0.05	0.05	0.05
$gobmk_7$	1.13	1.28	1.44	4.45	4.45	4.45	0.17	0.19	0.21
$h264ref$	1.92	1.91	1.79	4.45	4.45	4.45	0.35	0.34	0.33
mcf	1.37	1.24	1.22	4.83	4.83	4.83	0.22	0.2	0.19
omnetpp	1.39	1.39	1.38	6	6	6	0.19	0.19	0.19
sjeng	0.78	0.78	0.77	4.49	4.48	4.48	0.04	0.04	0.04
Xalan	2.32	2.36	2.52	6	6	6	0.36	0.36	0.38
cactus	2.15	2.21	2.24	4.65	4.58	4.56	0.41	0.42	0.42
calculix	1.73	1.51	1.38	4.94	4.94	2.03	0.35	0.31	0.21
dealII	1.56	1.47	1.44	6	6	6	0.28	0.27	0.27
gamess	2.16	2.57	2.23	4.83	4.79	4.15	0.39	0.41	0.34
Gems	1.87	1.76	1.76	4.76	4.78	4.79	0.06	0.03	0.02
gromacs	2.47	2.41	2.37	4.45	4.45	4.45	0.48	0.48	0.47
lbm	1.88	1.88	1.88	4.69	4.69	4.69	0.29	0.29	0.29
leslie3d	2.43	2.43	2.44	4.75	4.73	4.72	0.4	0.39	0.43
milc	2.43	2.44	2.45	4.69	4.69	4.67	0.44	0.44	0.44
povray	1.11	1.11	1.56	4.87	4.86	3.96	0.2	0.19	0.27
soplex	1.05	0.74	–	6	1.3	–	0.08	0.03	–
tonto	1.86	1.34	0.94	4.63	4.59	2.77	0.37	0.28	0.17
wrf	2.12	2.22	2.22	4.75	4.76	4.75	0.11	0.1	0.09
zeusmp	1.33	1.33	1.33	4.73	4.73	4.73	0.16	0.15	0.15

Table 4.2: Values of Average, Max and Total entropy, for different sizes of the LLC. The upper half is for the SPEC CPU and the lower half corresponds to the SPEC FP. Some applications are not in the table because they never evict LLC blocks that need to be written back, other than, maybe, after the process finishes. The same happens to *soplex* for LLC size of 4M.

value making it differ with the value experienced when different processes share the cache, and the effective size used by each process is smaller than the actual size.

4.2.4. Error Correcting Pointers (ECP)

The usage of ECP to survive stuck-at faults has been one of the most successful techniques in the last few years. They were introduced by Schechter et. al [64], outperforming all techniques proposed to the date by large.

The idea behind ECP is using pairs $< p_i, r_i >$ of pointers (p_i) and replacement cells (r_i), such that when a cell in the block fails, a pair is allocated. The pointer stores the index of the cell to mark the failure and the replacement cell is used to store the value that would be held in that cell. With an overhead lower than 12.5%, a 512-bit block can be provided with 6 pairs. There are also 6 extra bits marking whether or not a given pair has been allocated and therefore taken into account for both reading and writing the block not.

When a block is written to, the actually-written value is compared with the intended value. If a new discrepancy is discovered, a pair $< p_i, r_i >$ is allocated, and the index of the cell in the block is written to p_i. All replacement cells are also updated with the new value. If any of them fails, let's say r_j, a new pair $< p_i, r_i >$ is allocated. p_i is set to p_j to point the same cell, and r_j takes the intended value.

When a block is read, all the allocated pointers are read and the failing cells values are substituted by the values in the replacement cells. This is done in index order, so if $i > j$ and $p_i = p_j$, the value used is r_i. This scenario happens if and only if r_i has experienced a failure. This priority-based substitution allows to correct up to 6 failures in both the block and the extra storage.

In the next section we show the foundations of a technique using compression which is used to extend the lifetime of a device beyond the limits of ECP.

4.3. Technique

In this section, we present three techniques: COMP, that compresses the information associated to a block, and fit it in the healthy bits of the block to expand the effective lifetime. $COMP_{CP}$ is a variation of COMP using a compressing scheme much better suited for the context. Finally, we introduce CEPRAM, the last technique. It uses fine grain block pairing plus backspace capabilities to expand the effective lifetime even further.

In the following, we introduce the compression algorithms from the initial definition, explaining the modifications we have made, along with COMP and $COMP_{CP}$. Next, we show how CEPRAM works using block-level pairing. After that, we remind how wear leveling techniques work, and how the behave when using our techniques. This section is concluded by a rough estimate of the lifetime of a system using CEPRAM.

4.3.1. No-Table LZW compression (NTZip)

LZW is a traditional text compression algorithm used in many data formats and applications. Being well know and widespread, we take LZW as the starting point for our compression scheme. Algorithm 4.1 shows the pseudo-code for compression. LZW considers a dictionary preloaded with the symbols of the alphabet, in this case $\mathcal{A} = \{00_{hex}, 01_{hex}, ..., ff_{hex}\}$. Then symbols from the input are concatenated until the word they form is not in the dictionary. Then, the index of the prefix is output with the width necessary to write the size of the dictionary: $\lceil \log_2(sizeof(Dic)) \rceil$, the whole word added to the end of the dictionary, and the considered word becomes the last read symbol. This process is iterated until all the input is consumed

Algorithm 4.1 LZW compression

```
Dic ← A
S ← λ
while inputLeft() do
    N ← getChar()
    while concat(S, N) ∈ Dic do
        S ← concat(S, N)
        N ← getChar()
    end while
    output(index(S, Dic), width = ⌈log2(sizeof(Dic))⌉
    insert(concat(S, N), Dic))
    S ← N
end while
```

Due to the dynamic construction of the dictionary, in the decoding process the dictionary is also generated *on the fly*, so it doesn't need to be transferred along with the encoded words.

The power of LZW resides in the repetitions of string in the input text, that are encoded as smaller symbols.

LZW, as many compression schemes is designed for compressing big amounts of data. As such, it focuses in achieving good compression rates for the average cases, and doesn't care about the worst case, because there is enough statistical variety to absorb its effect resulting in a good compression ratio. In our context, we are dominated by worst case, therefore we need some extra mechanisms to further compress.

NTZip is a variation of LZW in which the dictionary is not pre-loaded. It starts empty, and symbols are added as they appear. This requires output symbols to have one extra bit. The first bit of the output symbol has following meaning:

- 0: The following 8 bits represent a symbol from $\mathcal{A}$ that was not present yet in the table.
- 1: The following $\lceil \log_2(sizeof(Dic)) \rceil$ bits represent a symbol from the table.

At first sight, this modification seems to penalize blocks with many different symbols, because it requires 9 bits per symbol: the leading 0, plus the 8 bits of the symbol itself. Giving a second thought, in LZW, after the first string is added to the dictionary, the width of output symbols is $\lceil \log_2(sizeof(Dic)) \rceil = 9$, as in NTZip, so we are not using more bits for the insertion of new symbols than LZW does. The main advantage of this modification is that $sizeof(Dic)$ starts at 0 and may grow up to 64 at most, so output symbols that refer to the dictionary are, at most 7 bits long (1 preceding bit plus $log_2(64)$, as opposed to symbols in LZW that are, in our context, 9 bits long, exception made of the first output symbol which is just 8 bits.

Algorithm 4.2 NTZip compression

```
Dic ← φ
S ← λ  /* Output the first symbol, and insert it in Dic */
insert(concat(N), Dic))
output(concat(1, N))  /* Proceed with the rest of the input */
N ← getChar()
ns ← true
while inputLeft() do
    while concat(S, N) ∈ Dic do
        S ← concat(S, N)
        N ← getChar()
    end while
    if ns then
        ns ← false
    else
        if S is just a new symbol then
            output(concat(1, S[0]))
            ns ← true
        else
            output(concat(0, index(S, Dic), width = ⌈log2(sizeof(Dic)))))⌉
        end if
    end if
    insert(concat(S, N), Dic))
    S ← tail(concat(S, N)
    N ← λ
end while
```

Algorithm 4.2 shows how the pseudo-code of the decoder looks after the modifications. First, we need to handle the first symbol separately, because $Dic = \phi$ is a special case. Another modification is the calls to output, that need to be prefixed by 0 or 1 depending on whether we found a new symbol that is not on the table yet or not. The last modification is the variable ns which controls that the same symbols are not output by error: When N gets a new symbol s, the condition $concat(S, N) \in Dic$ becomes false, we can assume S is formed of some previous symbols and $ns = false$, so the compressor outputs the index of S in

Dic, inserts $concat(S, N)$ in the dictionary, sets $S \leftarrow s$ (given that $N = s \neq \lambda$, $tail(concat(S, N)) = s$) and N gets the empty symbol. The outermost loop starts a new iteration, $concat(S, N)$ cannot be in the dictionary, because $S = s$ which is a new symbol. Therefore, the inner loop is not entered, and the condition of the following if is satisfied. Then the new symbol is output, and inserted in the dictionary. The last two sentences of the outer loop leave S and N unchanged, because $N = \lambda$. In the next iteration $concat(S, N) \in Dic$, so N gets the next symbol from the input, and $concat(S, N) \notin Dic$. Here is when ns appears in the scene: if we didn't check ns the algorithm would just output the index of s in the dictionary, leading to s output twice, first as a new symbol and then as an indexed symbol, which is wrong. Having ns allows the algorithm to just insert $concat(S, N)$ in the dictionary and proceed. Using NTZip we achieve better results in average and almost the same worst case (just 1 more bits over a total of 575), as argued in Section 4.6.

Using NTZip requires hardware support to do the compression/decompression, but also requires extra bits to hold information about the block. In the design of COMP, we want to keep the overhead at bay, we target, at most 12.5% space overhead, same as ECP_6 [64], SEC_{64} [68], $Pairing_8$ [69], $Wilkerson_4$ [70], and a perfect code correcting up to 9 errors. That makes for 64 bits of extra space.

The idea behind COMP, depicted in Figure 4.4 is using a ECP_6 scheme until the 6^{th} failure takes place (a). Once the 7^{th} failure arises (b), instead of discarding the block, it is compressed. Out of the 64 extra bits, 4 are used to encode that the block is compressed (yellow in the picture), 56 bits hold 7 8-bit pointers (red in the meta-data) that point to 7 bit pairs that are discarded (512 bits $\Rightarrow$ 256 pairs $\Rightarrow$ 8-bit pointers), so we can point to the 7 failing pairs (red/black), and the data is compressed as long as it fits in the remaining space, calculated by subtracting $7 * 2$ for the seven pair of bits discarded (failing bit plus accompanying bit due to granularity of addressing), from the block size: $512 - 7 * 2 = 498$ bits. When the 8^{th} failure takes place (c), the 4 bits encode that the block is compressed and there are 8 pointers 7-bit wide to groups of 4 bits that are discarded, because at least one fails. This allows us to use $512 - 8 * 4 = 480$ bits to store the compressed data. The next step is widening the discarded chunk size again, narrowing the pointer size to 6 bits. This allows us to discard 9 bytes when the 9^{th} failure happens (d), and 10 (e), when another cell fails. This method can survive up to 10 failures, as long as the compressed data fits in $512 - 10 * 8 = 432$ bits. If at some point the data is compressed and it doesn't fit in the block, the block is deemed useless and discarded.

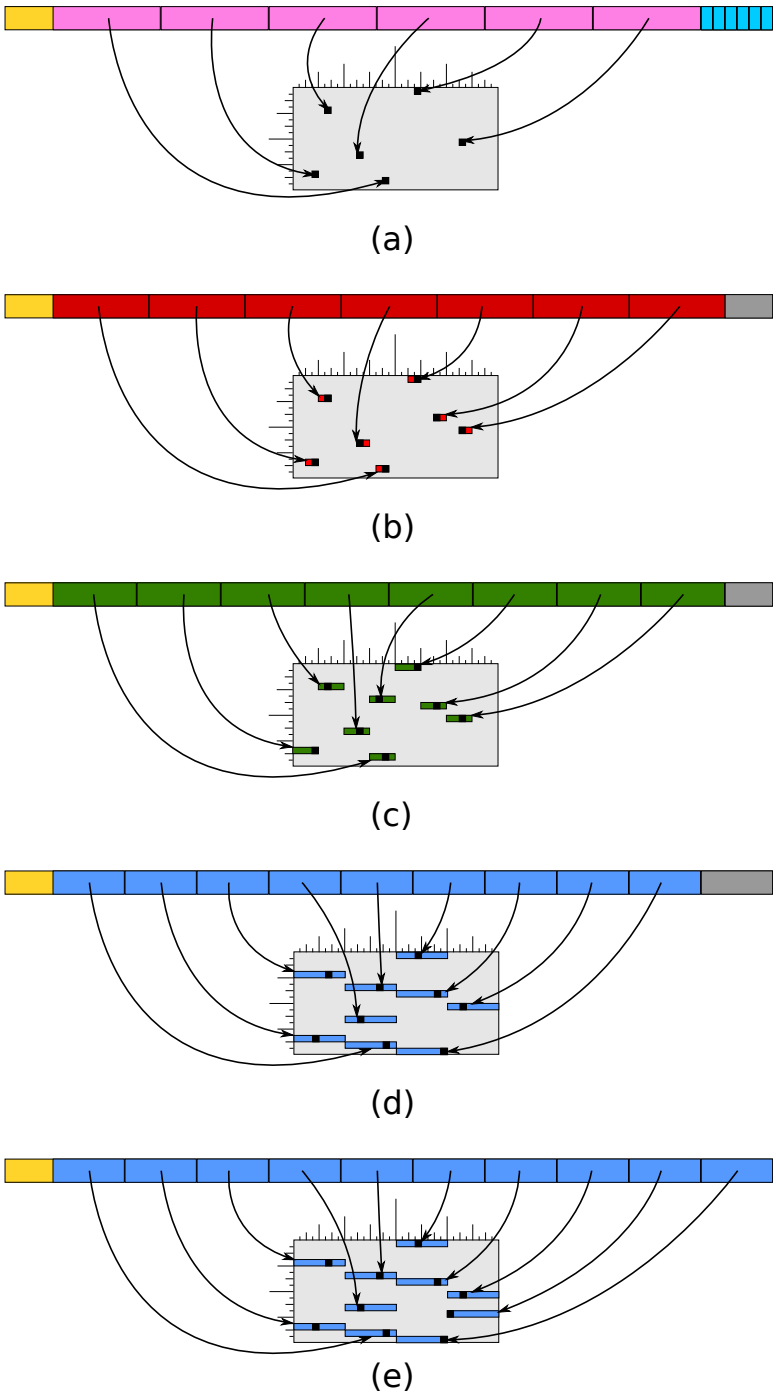


Figure 4.4: In every scenario the top horizontal bar symbolizes the 64 bits of meta-data, and the light-grey square models the data block. In the meta-data section color grey means unused bits. (a) ECP is used while the number of failures is 6 or less. 4 bits (yellow) encode how many ECP pointers are in use. There are up to 6 9-bit pointers (magenta) pointing the failing bits (black) and 6 replacement cells (cyan), for a total of 64 bits overhead. (b) If the block has 7 failures, then 4 bits (yellow) encode it, 7 8-bit pointers (red) point to the failing pair. Out of each pair, only one bit is failing (black) and the other one is discarded, even if it is healthy (red). (c) 4 bits (yellow) encode that the block has experienced 8 failures, and the following bits are 8 7-bit wide pointers (green) to groups of 4 bits (green) that are discarded because at least 1 bit is failing (black). (d) and (e) show the last two scenarios this technique can survive, the 9^{th} and 10^{th} errors. Again, 4 bits (yellow) encode the state, and there are 9 for (d) and 10 for (e) 6-bit pointers (blue)to whole bytes that are discarded (blue) because at least one bit (black) is failing.

State	Encoding	Description
NF	0000	Initial state, no failures.
ECP_1	0001	First failure, corrected applying ECP.
ECP_2	0011	Second failure, corrected applying ECP.
ECP_3	0010	Third failure, corrected applying ECP.
ECP_4	0110	Fourth failure, corrected applying ECP.
ECP_5	1110	Fifth failure, corrected applying ECP.
ECP_6	1100	Sixth failure, corrected applying ECP.
COMP $_2$	1000	Seventh failure, discard bit pairs and apply compression.
COMP $_4$	1001	Eighth failure, discard bit fours and apply compression.
COMP $_{8a}$	1101	Ninth failure, discard whole bytes and apply compression.
COMP $_{8b}$	0101	Tenth failure, discard whole bytes and apply compression.

Table 4.3: States for COMP and $COMP_{CP}$ and a proposed encoding to minimize bit-flips in transitions.

This is feasible, because we devote 4 bits to encode the state, as Table 4.3 shows.

This totals 11 different states. We can encode this with 4 bits, as proposed before.

The results for applying this technique improve ECP, if only because it is build on top of it, but are not very good. The problem is that we are limited by worst case. This means that it doesn't matter if a block compresses down to, let us say, 300 bits, so it fits even with 10 failures and 10 bytes discarded, on average, because if in a write-back operation it doesn't fit in the available space, the block is deemed useless and discarded, and the whole page with it. Therefore, this is not the best context for compression.

Another shortcoming of this scheme is discarding 10 bytes when there are 10 bits failing, the rate of wasted space is 10x, this motivates adapting NTZip to make it more suitable for our purpose.

4.3.2. NTZip with backspace

The main problem with COMP is that a lot of space is wasted due to the limitation in the size of the pointers. In this section we propose not doing explicit recording of the failing cells. Given that stuck-at cells are still readable, and thus, the failure is exposed only 50% of the times. The rationale is that, if we assume equiprobability of 0 and 1 in the values of each bit, the probability of a stuck-at bit being written a different value that the stuck-at value is 50%. This means that if we have n failing bits, on average only $\frac{n}{2}$ of the failures will manifest. Knowing this, we propose NTZipBs, a modification of NTZip in which the symbol "0" of the output code encodes a backspace, that lets the code express that there have been a failure in the previous symbol, and should be fixed. This is done by modifying the symbol table (dictionary). The very first entry is allocated upon initialization. All match tests to index 0 return *false*, so the output symbol is never 0. If during the writing of a symbol a fault is exposed, then a 0-index symbol is output to mean that there

is an error in the previous symbol. Algorithms 4.3 and 4.4 show the code for an optimistic simplification of NTZipBs that assumes that no failures are exposed in the backspace, and that no symbol is, due to failures, transformed into a backspace. The idea is the same as in NTZip, but when writing the block, if a failure is detected in one symbol, a backspace is inserted afterwards. Appendix A has a more detailed explanation of the algorithm, along with some problems with ambiguity in the decoder.

Algorithm 4.3 NTZipBs compression

$Dic \leftarrow \mathcal{A}$
$insert(< bs >, Dic)$
$S \leftarrow \lambda$
while $inputLeft()$ **do**
  $N \leftarrow getChar()$
  while $concat(S, N) \in Dic$ **do**
    $S \leftarrow concat(S, N)$
    $N \leftarrow getChar()$
  end while
  $output\_symbol(index(S, Dic), width = \lceil \log_2(sizeof(Dic)) \rceil$
  $insert(concat(S, N), Dic))$
  $S \leftarrow N$
end while

Algorithm 4.4 $output\_symbol(S : integer, width : integer)$

$error \leftarrow$ **true**
while $error$ **do**
  $write_memory(S, width)$
  $error \leftarrow check_memory(S, width)$
  if $error$ **then**
    $write_memory(0, width)$
  end if
end while

The two major shortcomings of this modification are:

- A block with n failures can requires up to $n+1$ iterative writes: If in the first write we detect one failing cell, we perform a second write of just the tail of the block after the failure with the backspace and the rest of the symbols. If, in turn, this second write shows a failure, a third write is required, and so on up to $n+1$ writes.

- The decoding of LZW, NTZip and NTZipBs is not parallelizable for such small blocks. The coding is not parallelizable either, but write-backs are not in the critical path. This has a mayor impact, because it will insert non-negligible time overheads to LLC failures in the system "early".

Code	Pattern	Output	Length(b)
00	ZZZZ	(00)	2
01	MMMM	(01)bbbb	6
1101	MMMX	(1101)bbbbB	16
1100	MMXX	(1100)bbbbBB	24
10	XXXX	(10)BBBB	34
111	$< bs >$	(111)bbbbbB..B	16..48

Table 4.4: Pattern encoding for C-PackBs

The first problem is not critical, it is reasonable to slow down writes as the system gets old. We assume that it is better to have some slow blocks than to discard some pages. The second problem is more important, and is what we address in the following section.

4.3.3. C-Pack and C-PackBs

Chen [65] proposed C-Pack, a cache compression scheme targeting high performance. $COMP_{CP}$ is a variation of COMP using C-Pack instead of NTZip as compression scheme. The evaluation in [65] shows that C-Pack is a good scheme for this context (small blocks) and performance is not much affected. They show a high level schematic of the encoding and decoding hardware. We slightly adapt the hardware decoder by adding some logic to discard the failing sets of bits, which can done in a few gate levels.

C-Pack input alphabet are 4-byte symbols. Each byte can be in one of the following 3 categories: Z if it is $0x00$. M if it matches the byte in the same position inside the symbol of a word in the dictionary. X if the byte is neither Z nor M. According to this classification, C-Pack focuses on the patterns in Table 4.4. For example, $0x00000000$ matches the pattern $ZZZZ$ because all four bytes are zero, and is inserted in the dictionary. If the next word in a block is $0x12345600$, it matches $XXXZ$, or $XXXM$ because the last byte equals the last byte of the previous word, but out of the considered patterns (Table 4.8) it is treated as $XXXX$, and inserted in the dictionary. If later on a word is $0x12341234$, the pattern is $MMXX$ because the 2 MSB match those of $0x12345600$.

Using C-Pack also requires discarding faulty bits as well as NTZip, what led us to modify it including a *backspace* character. Table 4.4 shows the modification of pattern encoding done to C-Pack (Table 1 in [65]).

The pattern *ZZZX* is eliminated. We do that without hurting compression much by initializing the dictionary with the word **0**, *i.e.*, the word formed by 32 zeroes, so *ZZZX* is a subcase of *MMMX*, also we can recognize *ZZXX* as *MMXX* with **0**. We substitute it with the backspace ($< bs >$) and also shorten it in one bit.

In C-PackBs, the $< bs >$ is, strictly speaking, not a *backspace*, but a correction symbol, always followed by 5 bits, to make a 1-byte symbol. The longest symbol is

34-bit long, requiring less than 5 whole bytes. In the 5 bits following $< bs >$ each 1 means that a byte contains at least one error, and accordingly, after the bs-byte, are as many bytes as 1s in the 5 bit pattern, each to be xor-ed with the failing byte of the previous symbol. Figure 4.5 depicts an example.

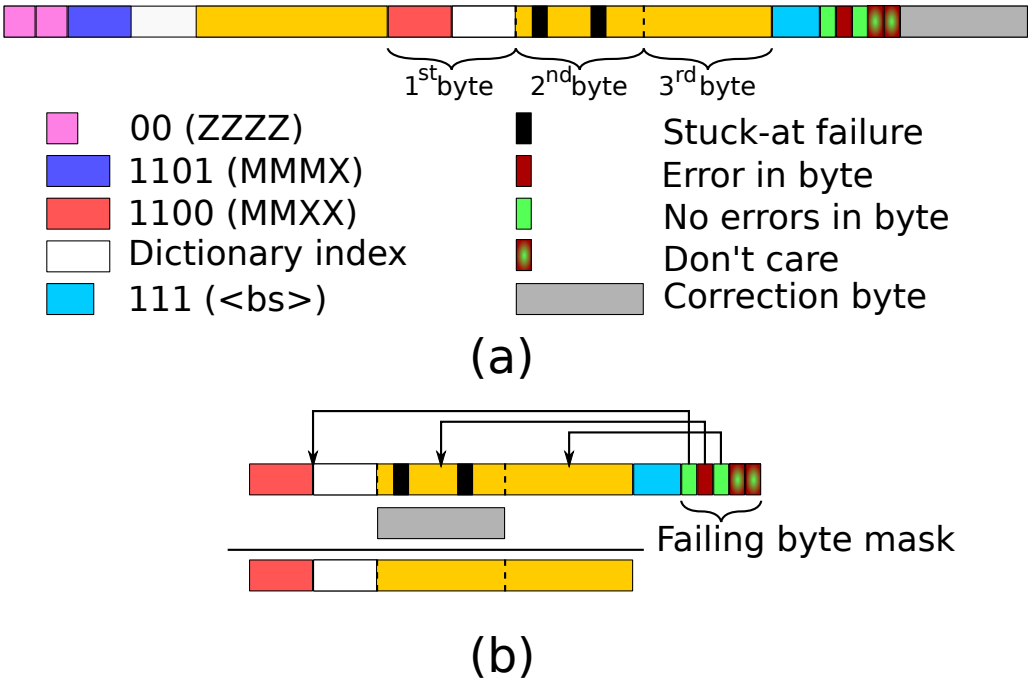


Figure 4.5: Error handling in C-PackBs. (a) The first part of a compressed block: First 2 symbols encode two zero-words. Next symbols is a MMMX pattern, so following the symbol are 4 bits to index the dictionary, and the less significant byte. The fourth symbol is MMXX, so it is followed by the 4-bit index of the matching word in the dictionary, and the two less significant bytes. in the first byte (pattern descriptor+index), there are no errors, but in the next byte there are two errors: bits 6 and 2. Next symbol is a ¡bs¿ with the pattern "01000" because the failure is in the second bit, and the next bytes is the correction byte. (b) In order to correct the error, the symbol is XOR-ed with the correction byte (in this case "01000100") to generate the correct symbol.

In 4.5 (a) there is an extract of a compressed block. The first two symbols correspond to the pattern *ZZZZ*, next symbol is *MMMX*, next one is *MMXX*, and is the symbol containing failures, and therefore, next symbol is $< bs >$. The failing symbol is comprised by the pattern descriptor, "1100" (in red), the 4-bit corresponding to the index in the dictionary of the matched word (in white), and the 2 least significant bytes of the uncompressed word (in yellow). In this example the failing bits are in the second byte. After writing the symbol we discover the failures, so, instead of writing the next symbol, we insert a backspace (cyan), followed by 5 bits: 0, because the first byte contains no errors (green), 1 because the second byte contains errors (deep red), and three more zeroes. After it, a byte containing ones in the position of the failures is inserted. After that, the rest of the block is written. When we decode the word (b), the decoder detects the $< bs >$ symbol, and reads the following 5 bits. Given that there is only one bit set, it reads one correcting byte that is XOR-ed with the second byte to produce the corrected symbol that

can then be decoded. The two bytes ($< bs >$, 5-bit mask and correcting byte) are eliminated and the decoding proceeds. There are more complex situations in which the $< bs >$ contains errors or even situations in which the error is in the header of a symbol, resulting in an output symbol which length is different than expected. These situations only affect the step of writing into memory. The coding is unaffected, and the decoding is the same regardless of where the error happened, and consists in a pre-pass that detects $< bs >$ headers and corrects symbols.

4.3.4. Further increasing the life through block-level pairing

The last proposal of this thesis is CEPRAM: Memory compression with block-level pairing for an improved life-span in resistive memories. The idea of CEPRAM is working as $COMP_{CP}$ for the 10 first failures. CEPRAM keeps a pool of discarded pages. When a block hits the 11^{th} failure, if the pool is empty, the whole page is discarded from the system and added to the pool. If the pool is not empty, the first available block is allocated as "overflow" block of the failing one. This is done by using one of the available encodings for the state in the meta-data for this new state, *PairLeader*. Another one of those available encodings will be used for the linked block, *PairSlave*, and yet another one for when a block is not usable anymore *Useless*. The remaining 60 bits are used to store the physical address of the linked page. Actually, those 60 bits allow for writing the address with redundancy, to survive stuck-at failures in the meta-data region. The linked block has its state modified accordingly, and from then on, when the block is written to, the block is compressed with C-PackBs, using the second block to write data overflowing from the first, if at some point the compressed data doesn't fit in the 1024 bits of the combined blocks, then the *PairLeader* is discarded with its whole page, and added to the pool. In this initial proposal, the linked block is lost. The decoding of a *PairLeader* proceeds as usual, but continuing with the *PairSlave*, if after processing the first block the decoder has found less than 16 non $< bs >$-symbols.

4.3.5. Wear Leveling

In CEPRAM, we use wear leveling techniques as in [71] and [72] . However, we use that only for the data part, the meta-data is not *wear-leveled*. The reason for this is that we need the 4 state bits to be reliable. If a line goes through all states, it will be around 3 writes to each bit. The probability of these bits to get *stuck-at* is really close to 0. Also we need the pointers to discarded blocks to be as healthy as possible, and the same for the last stage of a block life, when an overflow block has to be linked, because if a block cannot be linked or cannot transition states, the block (and the whole page with it) is immediately discarded.

With this decision, we are trading more wear in the data bits for less wear in the more-critical meta-data bits.

When two blocks are paired, the leader suffers wear more often than the slave (overflow) block. Allowing for wear leveling inside the pair affects performance because the start of the block may be physically placed in the second block, and that scenario requires fetching the *PairLeader* from memory, getting the *PairSlave* address, fetching it, and then start the $< bs >$ expansion. If wear is leveled only inside each block, then the $< bs >$ expansion can be started as soon as the first block arrives, and the fetching of the second is done in background.

4.3.6. Multiple Linking

CEPRAM only links once, but an scheme can be devised in which we have plenty of overflow blocks. We discarded that option because that situation arises only when the system is close to disfunctionality, and there is no point in "prolonging the agony".

4.3.7. Lifetime estimation

The average compressed block size is 320 bits. For every failure, in the worst case, we use 2 bytes, one for the $< bs >$ and the mask, and another one for the correction byte. If two failures are in the same symbol, then the algorithm may insert either 2 or 3 bytes, depending on the two failures being in the same byte or in different bytes. We can do an average case estimation of the lifetime measured in *survived failures*. If a block is 512 bits, on average, there are $512-320 = 192$ bits $= 24$ bytes, that let us correct up to $\left[\frac{24}{2}\right] = 12$ failures before discarding the first block. After pairing, the amount of available bits is $1024 - 320 = 704$ that allow for $\left[\frac{704}{16} = 44\right]$ failures corrected. On average, if the probability of a stuck-at failure to manifest is 0.5, this can correct on average 90 failures per paired block.

In the next Section, we show the results of the evaluation of this proposed techniques to back up the claims that this is a competitive technique.

4.4. Related work

The shortcomings of resistive technologies, particularly those regarding PCM have been a matter of interest recently. The problem of cell lifetime has been addressed mainly in three different ways. The first way is to reduce the wear experienced by cells by reducing the number of PCM writes. This can be achieved with a variety of techniques implemented at different levels. The first idea was in [73], and consists in detecting and avoiding silent writes. Differential writes (DCW) were introduced in [74] showing a big improvement in lifetime. Buffering techniques have also been explored in [72, 75, 76] presenting a level of DRAM buffering to alleviate the wear on PCM. In [76], the authors propose some techniques to overcome the process variation in terms of power and lifetime. The also use compression, but just to reduce

the width of writes to some blocks that are deemed compressible by an OS-guided procedure. Another technique called Flip-N-Write [77] reduces wear by flipping those words that incur in a lot of bit-flips in combination of DCW to reduce the number of effective bit-flips, thus extending the lifetime. The LLC has the ability to help in this purpose as in [78]. A modification on the replacement policy of the LLC to make it PCM-aware can help in reducing the amount of writes done to main memory without hurting performance. Another way to tackle the limited lifetime is to evenly spread the wear among all cells [74, 71, 79] at different granularity, and from different perspectives: the two former are hardware techniques, and the latter is a software technique at OS level. These techniques prevent the appearance of hot spots that can lead to an early failure of the memory system. The third trend is to detect and survive errors [69, 64, 80, 81, 82, 83, 84] as this piece of work does. DRM [69] does coarse grain pairing using parity to detect failing bytes. As long as two pairs have not overlapping bytes failing, they can operate as a single page tolerating some failures. ECP [64] uses $< pointer, replacement\_cell >$ pairs to point and substitute failing cells, allowing for one failure survival per pointer. FGCR [83] and PAYG [84] are techniques built on top of ECP. The former does current regulation and voltage up-scaling techniques to reduce the wear of cells, and the latter an abstraction of wear leveling applied to ECP pointers: At system failure time, a big share of the blocks have used less than 2 pointers, therefore, it would be more efficient to have a pool of pointers, and allocate them on demand as faults appear. Free-p [81] applies fine-grain pairing as CEPRAM. The main difference is that FREE-p is simple to implement and requires few extra hardware, but when a block fails it just points a "healthy" one instead of combining them somehow. On the other hand, CEPRAM is more complex because it adds a technique to survive failures that arise when two blocks are combined. SAFER [80] and RDIS [82] try to break a block down in pieces with different properties. SAFER makes n partitions, equal in size, such that there are at most one error per partition. Each partition is provided with a bit to express whether the that partition data should be flipped or not after reading. When a partition is written to, if no faults are detected, the corresponding bit is clear. If a fault is detected, then the bit is set, the word is flipped and written to the cells, so the fault is masked. RDIS does something similar. It tries to dynamically identify a set comprised only by exposed failures and healthy cells such that no exposed failures lie out of the set. Such a set is called *invertible set*. The intended contents of the invertible set are inverted and written to the hardware, avoiding the exposure of any fault, allowing for stuck-at fault survival. There are also PCM-aware software techniques to reduce the wearing such as [59] that proposes a modification of database algorithms to make them PCM-aware.

Many of the aforementioned technique also have an impact in power, either directly or as a beneficial side-effect. Reducing the amount of writes to PCM translates in power reduction. There are, nevertheless, pieces of research that focus on that problem [85].

There is work done in performance improvement. PCM read latency is in the same grounds of DRAM read latency, therefore the technique have focused on actually keeping PCM writes off the critical path. Write cancellation and Write pausing [86]

is a technique to prevent writes to block the main memory access for long, preventing reads to access the memory, thus degrading performance. The LLC replacement policy introduced in [78] is also aware of the occupancy of the write queues to have them balanced. This way they don't get full, what would imply damaging performance.

As far as cache compression is concerned, other than compression algorithms as C-Pack [65], some work has been done on architecture and interfaces for systems with compressed memory systems [66].

Our proposal is the first one to use compression for endurance, extending the lifetime of a PRAM based system. It builds on top of ECP as some other proposals, and takes the lifetime well beyond the limit of ECP with a novel way of dealing with exposed errors through the use of a custom backspace encoding to ignore (NTZipBs) or repair (C-PackBs) exposed failures.

4.5. Experimental Environment

In this section we present the experimental environment and the methodology of simulation used to evaluate the technique proposed in the previous section. Next section contains the results obtained though this simulation and evaluation. We start by introducing the applications used for benchmarking and following it is a description of the two simulation tools used.

Benchmarks: We use all the applications from the suite SPEC2006 [87]. We have compiled the applications using GNU compilers, namely, *gcc*, *g++* and *gfortran* version 4.6. The optimization flags used are those enabled by the standard optimization flag *"-O2"*. All applications are run to completion with the test input.

Simulator for compression: For the compression schemes, we use a memory hierarchy simulator based on the cache simulator from SESC [88], a super-scalar processor simulator. This simulator is connected to a pintool [31] which instruments guest applications so the cache model takes the appropriate actions prior to every single memory access. Both instructions and data accesses are simulated. Table 4.5 gathers the parameters of the memory hierarchy simulated. Whenever the simulator evicts a block from the LLC, it performs the appropriate actions. For entropy and bit-flip probability calculations, these actions are just appropriately modifying the counters according to the contents of the block. For NTZip and C-Pack, these actions are compressing the block a do the due accountancy of compressed size. For NTZipBs and C-PackBs, these actions are more complex. First, the block is compressed. Second, for $n = 1, 2, ..., 50$ a 512-bit wide error mask containing exactly n errors is generated. The compressed block is written into a single block using the error mask, and it is accounted as an overflow if the required size exceeds 512 bits. Otherwise, it is accounted for as a success. Next, for $n = 1, 2, ..., 50$ a 1024-bit wide error mask containing exactly n errors is generated. The compressed block is written into a single block using the error mask, and it is accounted as an overflow if the required size exceeds 1024 bits. Otherwise, it is accounted for as a success.

DL1	size	32KB
	associativity	8 ways
	block size	64 Bytes
	repl. policy	LRU
IL1	size	32KB
	associativity	8 ways
	block size	64 Bytes
	repl. policy	LRU
L2	size	256KB
	associativity	8 ways
	block size	64 Bytes
	repl. policy	LRU
L3	size	1M,2M,4M
	associativity	16 ways
	block size	64 Bytes
	repl. policy	LRU

Table 4.5: Parameters of the memory hierarchy.

This is done to differentiate the probability of overflow when the block is in its own from the probability when the block is paired. In addition, to calculate the amount of bit-flips, the "previous value" is kept, compressed and written for each mask, and both output are compared to count the number of bits that flip.

Memory system simulator: we use an in-house simulator.Doing faithful, cycle accurate simulation is hard and impractical, because it will require simulating workloads until the end of the lifetime of the system. For that reason, our simulator does a number of assumptions. First, we assume that there is an underlying wear-leveling technique to evenly wear all the memory cells, or at least, those not devoted to actual data, depending on the technique. Next, we assume that memory chips store data in 512-bit blocks (rows), and that each contiguous block of memory is spread over eight chips. Finally, writes are performed at block level. When a block is written, each bit is modified with probability p extracted from the study on flip probability. This is needed, because compressing data will narrow the size of writes, but as a side effect will increase the probability of bits being flipped, because compression reduces size by eliminating regular patterns, therefore, compressed blocks are less regular and more prone to have more bits modified per write.

As shown in Table 4.6, we simulate a system with $4KB$ pages. Each technique is simulated by creating a number of memory pages. Each bit inside every page, including meta-data, is created with a lifetime randomly distributed according to a Gaussian distribution $N(\mu = 10^8, \sigma^2 = 2.0 \cdot 10^7)$. Initially, the wear ratio, w is calculated as a function of p and the number of pages in the system, as Equation 4.1 shows. Then, according to the wear ratio, lifetime calculation is performed, and the simulation proceeds by locating the next failing cell, wearing all the system accordingly, applying the actions corresponding the simulated technique: allocating

Page size	4KB
Row size	64 Bytes
Rank	1
Chips per rank	8
Bit lines per chip	$x8$
Lifetime distribution	$N(\mu = 10^8, \sigma = 2.0 \cdot 10^7)$
Pages	2000

Table 4.6: Memory system simulator parameters.

an ECP, discarding a page, pairing a page, ignoring it because an ECC is capable of correcting the error, *etc.* After actions are taken, w is updated if needed, and lifetime recalculated.

$$w = p \cdot \frac{\#starting\_pages}{\#alive\_pages} \tag{4.1}$$

w is affected by 2 factors:

1. Page deaths: When physical pages die, the remaining pages need to "absorb" writes to that page. Due to the assumption of wear leveling techniques, that extra wear is evenly spread among the remaining pages. That requires recalculating w and the lifetime for all alive pages in the system.

2. Technique application: When a technique takes an action that modifies either p or the size of block writes, the wear ratio of a block, a page or of the whole system may be modified. For example, when a block start being compressed, both p and the average compressed write size are modified requiring w to be recalculated along with the remaining lifetime.

4.6. Evaluation

In this section, we show the results of the evaluations of the technique proposed in Section 4.3. First, in Sections 4.6.1 and 4.6.2 we show the performance of the compression schemes considered, in terms of frequency of best case, frequency of overflow and average compressed size. Next, in Sections 4.6.3 and 4.6.4 we evaluate NTZipBs and C-PackBs in terms of survived failures, and evolution of the average bits flipped per write with the number of errors. To conclude the evaluation, in Section 4.6.5 we show the results of the simulation of a PCM based system in terms of the lifespan, comparing it with previous proposals.

4.6.1. NTZip performance

The two aspects of a compression scheme that we have chosen to focus on are the following:

- *% Constant Block*: This number is the percentage of blocks that are constant, and therefore are the most compressible ones. With NTZip those blocks compress down to 48 bits.
- *% Overflows*: This number is the percentage of blocks that exceed 512 bits when compressed. This number represents how often a block lies in the *worst case scenario*. For NTZip the maximum compressed size is 576 bits: each byte is "new" so it is encoded as a 9-bit symbol, a 1 identifying the new byte plus the byte itself.

The reason for the modification of LZW is that, for the case of the constant block, LZW manages only to compress down to 98 bits: first symbol is 8 bits, and the remaining 10 symbols required to encode the word ($\frac{(1+10)*10}{2} = 55 < 64 < 66 = \frac{(1+11)*11}{2}$) require 9 bits, because the number of symbols of the dictionary is greater than 256 after the first byte is encoded and output. For the smallest compression ratio, NTZip is 576 bits, while LZW only improves it by 1 bit: 1 symbol of 8 bits, plus 63 symbols 9-bit wide total 575 bits. It is easy to proof that for input sizes of 64 bytes, NTZip compresses always to smaller sizes than LZW but for the case of 64 different symbols.

Figure 4.6 shows these two aspects for the spec2006 applications when LLC is 2MB. Table 4.7 completes this information with the values when the last level cache (LLC) size is varied from 1MB to 4MB. The upper half corresponds to SPECINT applications and their weighted average. The lower half to SPECFPU, and the respective weighted average. The last row, labeled **Aggregate** corresponds to the weighted average across all applications. Unless otherwise noted, the tables in the remainder of the chapter follow the same structure.

NTZip does an amazing job compressing blocks from SPECINT applications. The one for which it performs the worse is *bzip*2. Since *bzip*2 is a compression program in itself this is totally reasonable, because compressed data doesn't re-compress well.

Compressing SPECFP is harder. The compression algorithms considered are not FP oriented, and therefore, the higher variability at byte level intrinsic to floating point data is the reason why they perform worse than for integer code. Nevertheless, on average 75% of the blocks that floating point applications write back to memory compress to sizes ≤ 512, and 20% of the total are the same byte repeated over and over again.

The LLC size doesn't have a noticeable impact on either the % of constant blocks or the % of overflows. The only variation is for those applications for which a bigger LLC is able to capture the locality of the blocks devoted to flow-control that compress better than those holding data with high variability that show smaller locality and compressibility. The biggest change happens for *povray* which grows 5% to 45%. This sudden growth is due to a reduction in the amount of write-backs LLC performs from over 9 millions for 1MB, over 4.5 millions for 2MB, to less than 50.000 for 4MB. This weird behaviour is absorbed by the rest of values in the average because of the small weight.

	% Constant block			% Overflows		
L3Size	1MB	2MB	4MB	1M	2MB	4MB
astar	1.71	9.66	8.8	0.06	0	0.01
$bzip2_1$	3.15	3.18	4.01	2.43	3.71	6.06
$bzip2_2$	0.36	0.39	0.81	1.15	1.48	3.82
gcc	11.78	8.96	7.39	0	0	0
$gobmk_1$	98.63	98.62	98.62	0	0	0
$gobmk_2$	97.14	97.14	97.14	0	0	0
$gobmk_3$	98.93	98.93	98.9	0.01	0.01	0.01
$gobmk_4$	80.84	81.18	81.48	0	0	0
$gobmk_5$	99.41	99.4	99.38	0.01	0.01	0.01
$gobmk_6$	86.04	86.11	86.19	0	0	0
$gobmk_7$	22.94	23.99	24.7	0.02	0.01	0
h264ref	1.49	2.09	6.45	0	0	0.02
mcf	0.01	0.01	0.01	0	0	0
omnetpp	0.12	0.15	0.23	0	0	0
sjeng	73.85	74.35	75.06	0	0	0
xalancbmk	0.32	0.28	0.45	0	0	0
SPECINT	27.32	38.49	60.88	0.46	0.5	0.42
cactusADM	7.38	7.66	7.93	41.45	42.45	44.46
calculix	2.68	7.07	0	14.76	0.14	0
dealII	13.1	8.77	6.69	12.73	6.38	5.54
gamess	16.77	26.45	43.71	48.86	54.31	31.48
GemsFDTD	92.06	96.4	96.61	0.44	0.01	0
gromacs	6	5.18	5.41	64.89	61.21	56.59
lbm	–	–	–	0.43	0.43	0.43
leslie3d	18.79	19.05	18.34	28.66	28.06	27.87
milc	0.61	0.44	0.35	74.94	75.48	75.58
povray	9.53	16.31	14.29	3.17	5.32	45.54
soplex	74.19	45.45	0	0.29	0	0
sphinx3	26.02	27.74	26.81	2.61	2.81	4.69
tonto	6.74	7.4	4.55	21.15	5.99	4.55
wrf	84.6	86.5	88.02	4.05	3.01	2.97
zeusmp	65.13	65.02	64.84	5.92	5.95	5.97
SPECFP	21.05	21.2	22.67	24.77	24.38	20.31
Aggregate	22.54	24.42	28.65	18.96	19.93	17.19

Table 4.7: Percentage of blocks evicted from L3 that are constant (columns 2 to 4), thus compress down to 48 bits, and percentage of blocks evicted from L3 that compress to more than 512 bits using NTZip (columns 5 to 7). SPECINT, SPECFP and aggregate are the aggregate of all the blocks evicted corresponding to integer applications, floating point applications and all applications respectively

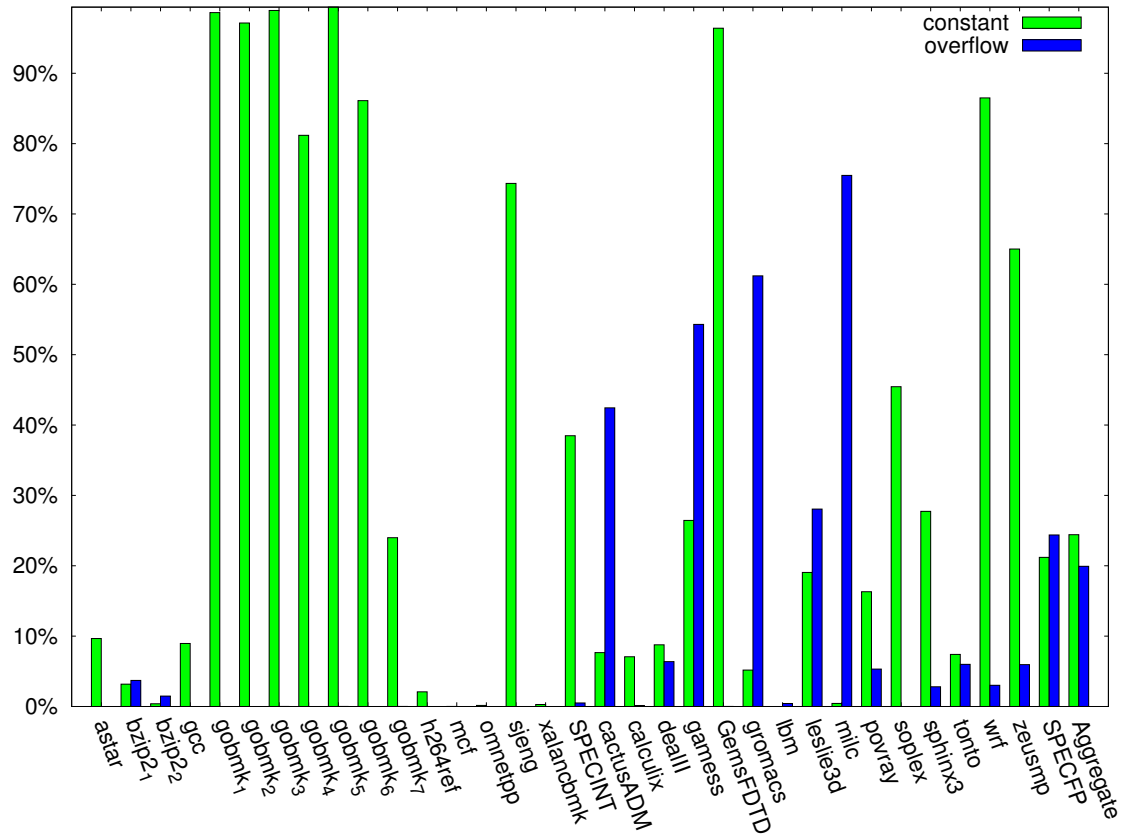


Figure 4.6: Percentage of blocks evicted from LLC that are the same byte repeated 64 times (green), and that compress with NTZip to more than 512 bits (blue).

Regarding those blocks that neither are constant nor compress to more than 512 bits, Table 4.9, columns 3 and 5 shows, for each application, the average compressed block size and the coefficient of variance, *i.e.*, the standard deviation divided by the average. For these blocks, the distribution of the size to which they compress is pseudo-normal. There is no general trend, not even if we distinguish between integer and floating point code. However, 335 bits is a low average size if we take into account that the input size is 512 bits, and with so few input symbols, just 64, it is hard to achieve high compression rates.

4.6.2. C-Pack performance

For C-Pack we show the same two quantities to characterize it. There is an slight difference, though. In the case of NTZip, a block being the same byte repeated over and over again is the best case scenario. C-Pack is, in that respect, more restrictive. It requires the block to be all zeroes, in which case C-Pack is able to compress it down to 32 bits, improving the 48 bits achieved by NTZip. Given that most of the constant blocks are zeroes, this restriction is not a big constraint. Another difference in the numbers is that in the presence of a block with 64 different bytes,

the compressed size for NTZip is 576 as discussed above, while C-Pack manages to use just 544 bytes. Although this two figures are not enough to state that the compression ratio of C-Pack is higher than that of NTZip, they are a good reason to use C-Pack. Since CEPRAM pairs two blocks and writes the compressed block in the two blocks, having a smaller maximum size makes C-Pack very well suited for this purpose.

Figure 4.7 gathers the percentage of written back blocks that are zeros, and the percentage that compress to more than the initial size when the LLC is 2MB. Table 4.8 shows that information augmented with the corresponding values for LLC size 1MB and 4MB.

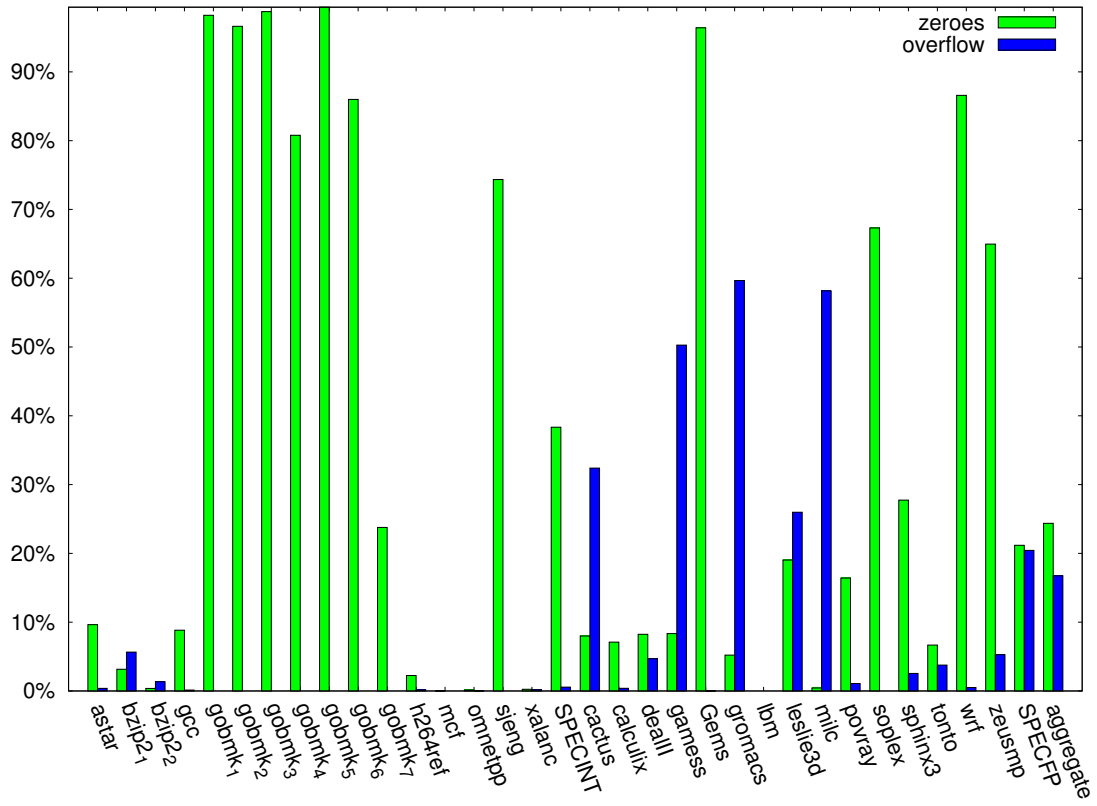


Figure 4.7: Percentage of blocks evicted from LLC that are $0x00$ repeated 64 times (green), and that compress with C-Pack to more than 512 bits (blue).

The numbers don't differ much from those achieved by NTZip. On the whole, C-Pack manages to keep overflows lower than NTZip, although the difference is just around 15%-20%.

As far as the rest of the written-back blocks are concerned, Table 4.9 shows the average and the CoV of the distribution of the compressed size. This distribution is pseudo-normal, differing from a normal more than in the case of NTZip. This is because being a word-oriented algorithm, there are only a small number of different sizes that can be achieved.

	% Zero block			% Overflows		
L3Size	1M	2MB	4MB	1M	2MB	4MB
astar	1.72	9.64	8.89	2.02	0.38	0.64
$bzip2_1$	3.14	3.16	3.91	4.08	5.65	7.72
$bzip2_2$	0.35	0.37	0.69	1.06	1.35	3.6
gcc	11.62	8.83	7.41	0.1	0.13	0.09
$gobmk_1$	98.22	98.22	98.21	0	0	0
$gobmk_2$	96.62	96.62	96.62	0	0	0
$gobmk_3$	98.77	98.76	98.74	0	0	0
$gobmk_4$	80.41	80.78	81.06	0	0	0
$gobmk_5$	99.4	99.4	99.38	0	0	0
$gobmk_6$	85.93	86	86.09	0	0	0
$gobmk_7$	22.74	23.77	24.42	0.02	0	0
$h264ref$	1.49	2.25	6.23	0.09	0.16	1.51
mcf	0.01	0.01	0.01	0	0	0
omnetpp	0.12	0.15	0.23	0.01	0.01	0.02
sjeng	73.85	74.35	75.06	0	0	0
xalanc	0.28	0.25	0.4	0.17	0.18	0.21
SPECINT	27.2	38.34	60.66	0.53	0.56	0.47
cactus	7.26	7.99	7.81	32.22	32.48	32.62
calculix	2.6	7.1	5.88	6.14	0.39	0
dealII	12.57	8.24	6.17	10.26	4.69	4.31
gamess	10.06	8.34	30.16	45.94	50.28	36.68
Gems	92.06	96.4	96.6	0.33	0.01	0.01
gromacs	5.99	5.21	5.22	64.54	59.66	54.4
lbm	–	–	–	–	–	–
leslie3d	18.56	19.03	17.72	26.29	25.98	26.45
milc	0.6	0.44	0.37	57.58	58.16	57.84
povray	9.27	16.42	16.67	0.78	1.08	1.39
soplex	74.9	67.31	0	0.07	0	0
*sphinx*3	26.03	27.75	23.29	2.31	2.55	4.67
tonto	5.93	6.67	4.76	17.14	3.76	0
wrf	83.48	86.58	87.98	2.24	0.5	0.49
zeusmp	65.07	64.96	64.77	5.22	5.27	5.36
SPECFP	20.93	21.18	22.39	20.75	20.44	17.08
aggregate	22.43	24.37	28.32	15.91	16.74	14.5

Table 4.8: Percentage of blocks evicted from L3 that are full of zeroes (columns 2 to 4), thus compress down to 32 bits, and percentage of blocks evicted from L3 that compress to more than 512 bits using C-Pack (columns 5 to 7). SPECINT, SPECFP and aggregate are the aggregate of all the blocks evicted corresponding to integer applications, floating point applications and all applications respectively

	$\bar{X}$		CoV	
L3Size	C-Pack	NTZip	C-Pack	NTZip
$astar$	198.5	211.9	0.35	0.21
$bzip2_1$	290.9	282.5	0.44	0.43
$bzip2_2$	439.6	450.3	0.13	0.13
gcc	152	175.6	0.35	0.3
$gobmk_1$	169.7	210	0.36	0.35
$gobmk_2$	175.3	217.3	0.33	0.31
$gobmk_3$	178.3	190.9	0.43	0.51
$gobmk_4$	198.2	232.7	0.36	0.33
$gobmk_5$	163	146.9	0.51	0.67
$gobmk_6$	196.8	232.5	0.32	0.29
$gobmk_7$	179	207.7	0.49	0.47
$h264ref$	273.1	292.3	0.21	0.21
mcf	205.6	232.7	0.23	0.19
$omnetpp$	194.3	217.9	0.09	0.08
$sjeng$	163.6	182.5	0.34	0.34
$xalancbmk$	353.2	294	0.31	0.18
$SPECINT$	286.3	306.8	0.53	0.54
$cactusADM$	310.4	298.3	0.4	0.4
$calculix$	315.4	299.2	0.5	0.42
$dealII$	206.9	229.1	0.72	0.67
$gamess$	194.4	196.4	0.41	0.33
$GemsFDTD$	285.1	299.7	0.49	0.42
$gromacs$	268.8	292.7	0.3	0.31
lbm	226.1	227.3	0.26	0.25
$leslie3d$	420.9	444.6	0.21	0.27
$milc$	439.6	437.6	0.46	0.39
$povray$	179.5	180.7	0.61	0.51
$soplex$	161.4	153.6	0.47	0.42
$sphinx3$	239.8	256	0.56	0.55
$tonto$	268.8	290.6	0.45	0.36
wrf	349.5	359.8	0.7	0.56
$zeusmp$	148.8	170.9	0.3	0.28
$SPECFP$	329.8	343.2	0.45	0.41
$Aggregate$	319.4	334.5	0.37	0.33

Table 4.9: Comparison of the average compressed block size for both C-Pack and NTZip, and the coefficient of variance ($CoV = \frac{S}{\bar{X}}$).

C-Pack as well as NTZip doesn't show any pattern or regularity in the average compressed size. There is no correlation between the average compressed size for both algorithms either. For some applications as $gobmk_{1,2,3,4}$, mcf or $zeusmp$ C-Pack outperforms NTZip, but for $xalancbmk$, $gobmk_5$ or $calculix$ it is NTZip outperforms C-Pack. There are also some applications ($bzip2$,$gamess$,$soplex$) for which the difference is negligible. Overall C-Pack achieves a smaller average size, but with a higher CoV.

These results back up the design decision of changing the compression algorithm for C-Pack, because there is already a high performance encoder/decoder, specially designed for the place in the hierarchy we intend to place it.

After this performance evaluation, we analyze the error surviving ability of these two schemes when they are augmented with a *backspace* symbol, to allow to identify and correct failures manifested in a paired block. We also analyze how using this backspace modifies the average number of bits flip per memory write.

4.6.3. NTZipBs performance

When we augment NTZip with the backspace character the average compressed size is hard to quantify. Instead, we have taken the amount of (exposed) failures survived in a block pair as the figure of merit. This number, M, is the maximum number of exposed errors, N, for which after doing the following process for all blocks that are written back to memory, none reported a final size greater than 1024 bits (paired block).

1. A block B that has been modified is evicted from LLC an requires to be written back.
2. Calculate C as the result of applying NTZip to B.
3. Generate a random 1024-bit mask with N errors, $mask_e$.
4. For each symbol $s \in C$
   - Write s and check $mask_e$.
   - If no errors are detected, proceed with next symbol.
   - If an error is detected, insert a backspace, write s and check for errors again.

The other feature we have focused is the probability (ratio) of blocks that actually exceed 1024 bits when written back with $M+1$ errors. This second number provides an insight on how long it will take for the block to fail after the $(M+1)^{th}$ failure is hit.

In our context, the behaviour of a system is decided by the worst case, because due to wear leveling technique the worst-case takes place in a number of different

	M (Max. survived exposed failures)	Overflow probability with $M+1$ failures
$astar$	29	3.94e-07
$bzip2_1$	28	8.92e-08
$bzip2_2$	27	1.12e-08
gcc	44	1.27e-06
$gobmk_1$	35	4.03e-07
$gobmk_2$	33	9.78e-08
$gobmk_3$	36	2.69e-06
$gobmk_4$	31	5.79e-08
$gobmk_5$	36	9.05e-07
$gobmk_6$	32	2.35e-07
$gobmk_7$	30	1.57e-07
$h264ref$	31	5.36e-08
mcf	–	——-
$omnetpp$	47	8.76e-06
$sjeng$	33	5.53e-08
$xalancbmk$	48	4.27e-06
$bwaves$	28	6.61e-07
$cactusADM$	28	4.94e-07
$calculix$	32	1.34e-04
$dealII$	28	1.70e-07
$gamess$	28	1.35e-06
$GemsFDTD$	29	8.58e-08
$gromacs$	29	5.00e-06
lbm	32	2.45e-08
$leslie3d$	–	——-
$milc$	28	6.14e-07
$povray$	34	1.94e-05
$soplex$	48	1.05e-03
$sphinx3$	30	1.54e-07
$tonto$	31	7.75e-05
wrf	31	2.21e-06
$zeusmp$	27	1.80e-08

Table 4.10: Behaviour characteristics of NTZipBs. The second column M, is the amount of (exposed) failures successfully survived for each benchmark. The last column shows the empirical probability of a evicted block to overflow a pair manifesting $M+1$ failures.

memory places as time passes. For this reason the average M and the average overflow probability with $M+1$ failures lack any meaning in this context and they are not shown.

Table 4.10 shows these two numbers for all the benchmarks. A paired block is able to survive, at least, 27 failures. The theoretical limit is 26:

- The maximum compresses size is 576. Therefore, there are at least $1024 - 576 = 448$ bits for correcting errors.
- In order to correct an error, we need, at most, 8 bits for the backspace (1 bit to signal we are indexing in the table, plus 7 bits to index the table if the error is in the 64^{th} symbol. After the backspace we need to output the symbol again, which if it wasn't in the table previously takes 9 bits, for a total of 17 extra bits.
- Dividing, NTZipBs is able to correct, at least, $\left[\frac{448}{17}\right] = 26$ failures.

The theoretical limit is a strange scenario and doesn't show up in our simulations. Moreover, for many of the applications more than 30 failures are survived.

In addition, this is the number of exposed failures. If a cell is stuck-at a value, and the block requires the cell to be that value, the failure is not exposed, and therefore no actions are required. This is important, because previous schemes as ECP don't take into account if the error is exposed or not, and just avoid the failing bit all the time. Moreover, if probability of a cell holding a 0 is the same of that of a cell ending up stuck-up at 0, then, the expected number of exposed failures per block write is half the number of failures. In other words, on average, half of the stuck-at cells are written the same value they hold, and only the other half require correcting actions. This makes NTZipBs able to at least $26 * 2 = 52$ failures per paired block, which dramatically improves the errors achieved by 2 ECP blocks in their own.

After the $(M+1)^{th}$ error, the overflow probability is quite low, but for *calculix* and *soplex*, allowing the block pair to be usable for a number of times before it actually overflows and the whole page has to be discarded.

When a block contains failures, the amount of bits that flip varies with the number of failures. Figure 4.8 shows this variation on the average number of bits that flip (y-axis) when the number of errors (x-axis) grows. The variation is quasi-linear. Those curves in the upper cluster show a small bending, but it can be approximated by a horizontal line with a admissible error. Here, all curves show a slope that ranges from around 0.3 to around 1, what means that the wear rate of the surviving cells is not vastly increased as cells start getting stuck-at.

4.6.4. C-PackBs performance

Concerning C-PackBs, we focus on the same two figures analyzed for NTZipBs in Section 4.6.3, this is, M: number of failures survived, and overflow probability when there are $M+1$ exposed failures.

	M (Max. survived exposed failures)	Overflow probability with $M+1$ failures
$astar$	33	1.18e-06
$bzip2_1$	33	8.03e-07
$bzip2_2$	32	5.64e-08
gcc	37	1.70e-06
$gobmk_1$	41	2.69e-07
$gobmk_2$	40	1.95e-07
$gobmk_3$	45	6.73e-07
$gobmk_4$	38	2.31e-07
$gobmk_5$	45	9.05e-07
$gobmk_6$	40	4.71e-07
$gobmk_7$	35	1.05e-07
$h264ref$	34	3.57e-08
mcf	45	9.64e-08
$omnetpp$	44	8.78e-06
$sjeng$	40	5.53e-08
$xalancbmk$	38	4.27e-06
$bwaves$	31	1.20e-07
$cactusADM$	31	8.23e-08
$calculix$	36	5.38e-05
$dealII$	32	4.24e-07
$gamess$	32	4.07e-06
$GemsFDTD$	33	8.58e-08
$gromacs$	33	1.60e-05
lbm	36	9.81e-09
$leslie3d$	30	8.61e-09
$milc$	30	7.14e-09
$povray$	38	3.89e-05
$soplex$	45	3.61e-04
$sphinx3$	32	1.54e-07
$tonto$	34	2.70e-05
wrf	33	7.37e-07
$zeusmp$	31	1.80e-08

Table 4.11: Behaviour characteristics of C-PackBs. The second column M, is the amount of (exposed) failures successfully survived for each benchmark. The last column shows the empirical probability of a evicted block to overflow a pair manifesting $N+1$ failures.

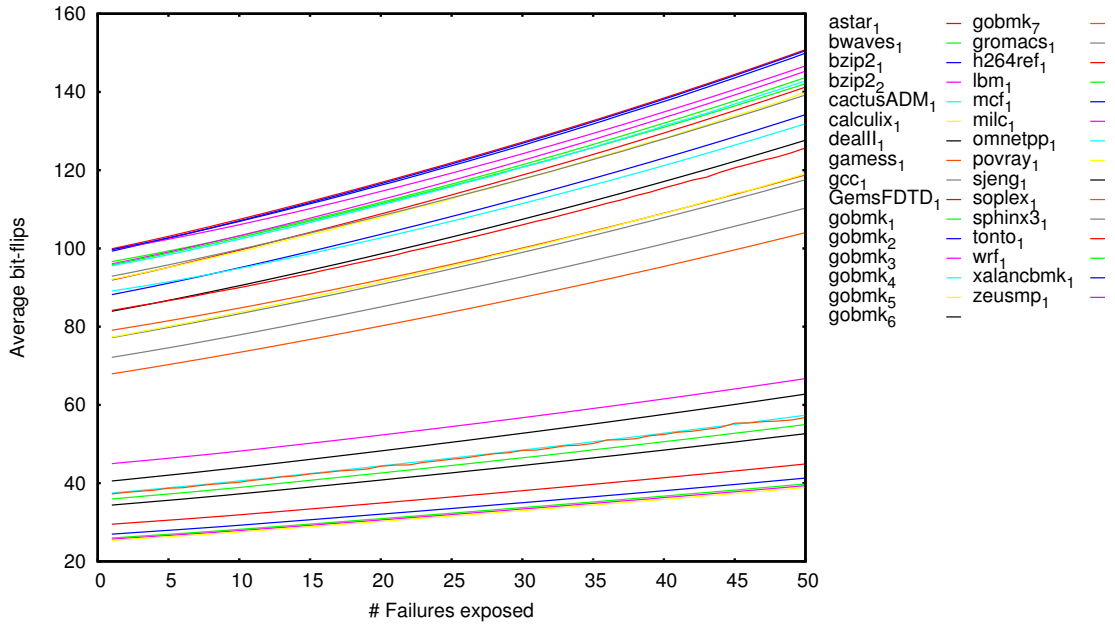


Figure 4.8: Average bit-flip per block variation for the different benchmarks as the number of failures grows for NTZipBs.

For C-PackBs the theoretical limit is 30 failures:

- The maximum compresses size is 544. Therefore, there are at least $1024 - 544 = 480$ bits for correcting errors.
- In order to correct an error, we need, at most, 16 bits: 3 bits for the BS code, plus the 5 bit mask, and then 8 bits with the correction mask. In the case of errors happening in the same byte/symbol, this increases the total bits required but decreases the bits required per error. Likewise, if a failure transforms a symbol into a BS, we just discard that byte, clearing the mask, and repeat the symbol again.
- Dividing, C-PackBs is able to correct, at least, $\left\lceil \frac{480}{16} \right\rceil = 30$ failures.

In this case the theoretical limit is actually reached for some applications, although a number of them show a more favorable behaviour, managing to get more failures survived.

In the same line that happened with average compression size between NTZip and C-Pack, NTZipBs and C-PackBs don't compare in a per-application basis. There are examples of both beating the other. On the big figure, C-PackBs is able to correct, at least 30 failures, which is 4 more than the minimum for NTZipBs.

Other than *soplex*, after the $(M+1)^{th}$ failure arises blocks are still usable for long. It is worth noticing that *soplex* is the application with the highest M, and for that, we don't think having a high overflow probability after surviving 45 failures is really a shortcoming.

To finish the high level analysis of C-PackBs, Figure 4.9 shows how the average amount of bits flipped per block varies with the number of failures exposed in the block. It is interesting the fact that this number increases linearly with the number of errors. In this Figure we don't give much importance to which curve belongs to which applications, but to the linear shape with slope between 0.25 and 1 that all lines show. This hints that having more errors will not, necessarily, increase the rate at which cells wear.

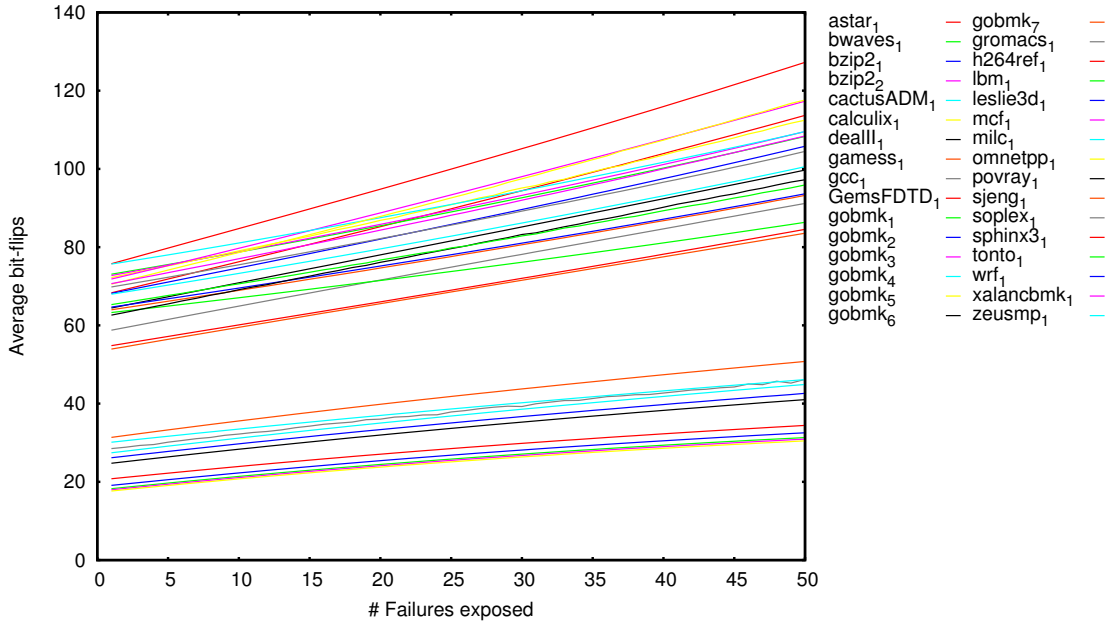


Figure 4.9: Average bit-flip per block variation for the different benchmarks as the number of failures grows for CPackBs.

If we compare C-PackBs and NTZipBs in this field, C-PackBs clearly outmatches NTZipBs. Although the slopes and the general shape of the curves is the same, those belonging to C-PackBs are at lower positions, meaning that the amount of bit-flips in a block compressed with C-PackBs is smaller than the amount of bit-flips in a block compressed with NTZip, meaning that NTZip wears the memory faster than NTZip does. This is mainly due to the design of the algorithms. NTZip is string-oriented, while C-Pack is word-oriented. If a byte inside a chain changes, the output is likely to vary significantly. It is likely that what was previously a chain in the dictionary now has to be compressed into two different tokens, or the other way round. On the other hand, in C-PackBs words match one of the patterns shown in Table 4.11. Any change in the word that doesn't alter which pattern the word matches will not change the output symbol encoding, maintaining the width, and the symbol header, maintaining the output largely unmodified.

4.6.5. Comparison to previous proposals

In this last part of the evaluation, we provide data illustrating how well a memory system implementing our technique performs.

We compare our technique with some proposals in the literature. For a number of years, DRAM memories with Error Correcting Codes (ECC) are available. It is intuitive to use ECC for PCM before exploring other alternatives, this is why we evaluate the performance of a system implementing a SECDED (Single Error Correction, Double Error Detection) scheme, that allows to correct one error per 64-bit chunk inside a block, but when a chunk manifests a second error, its whole page is discarded from the system. The kind of code used is (71,64), so the overhead is 10.9%, with some simple hardware in the memory controller to do the encoding and decoding.

Dynamically Replicated Memory (DRM) [69] was proposed by Ipek *et al.* and is an scheme that combines some hardware extension with support of the OS to tolerate a number of failures within pages. Roughly speaking, DRM assigns a parity bit to each byte inside a block, which allows to single error detection. If a write operation discovers more failures in the same byte, then the byte parity failure is enforced flipping a healthy bit, if necessary. The first error inside a page means a page failure. The system is provided with a pool of failed pages. When a page fails, it scans the pool for a page with which the failing bytes don't overlap. If it is unable to find one, the page is added to the pool. If a candidate is found then, the two pages are "paired", being one the primary and the other the replica, and from that moment on operations are done to the primary, unless a failing byte is involved, in which case are done to the replica. In [69], they report that this scheme can correct up to 160 failures per page. Therefore, our simulator doesn't simulate the whole system, but just pages that are paired, and once paired, then can sustain up to 160 failures, being the 161^{th} the trigger for page failure. The overhead is 1 bit per byte, this is, 12.5%, with important hardware and OS modifications.

Schechter *et al.* introduced Error Correction Pointers (ECP) [64] as an alternative to ECC for error correction in phase change memories. ECP is a clever technique that uses pointers to point errors and replacement cells to survive them. Using 12.5% overhead, they are able to correct and survive up to 6 errors (ECP_6). With an extra bit and a smart block rotating scheme, they can even survive 7 errors. We evaluate ECP_6 without the extra bit, and independently of the intra-block wear-leveling scheme used. When the 7^{th} failure takes place in a block, the whole page is discarded.

First, Table 4.12 shows a comparison of all the schemes in big numbers: the overhead, the failure unit, the correcting capability per unit, the number of failures successfully corrected on the moment of system failure, and the average number of errors in pages at system failure (failed pages are expected to have one more errors than the correction capability). We can see that CEPRAM dramatically increases both numbers, hinting for a longer lifetime. By design, CEPRAM cannot do worse than ECP, but this numbers show the big increase achieved in error-correcting capability.

First thing, point out that the numbers in the last two column correspond with the quantities when all pages are broken, therefore, they don't reflect how soon/late errors manifest. There are two interesting things to observe:

Scheme	Overhead	Failure Unit	Failures/unit survived	total failures	failures per page
SECDED	10.9%	64 bits	1	29072	26.21
DRM	12.5%	4KB page	160†	161216	149.78
ECP_6	12.5%	512 bits	6	149160	149.16
CEPRAM	12.5%	512 bits	≥ 30†	1446613	1447.11

Table 4.12: High level comparison of schemes. The first three columns show the characteristics of each algorithm while the last two show the results of our evaluation of a system with 1000 pages using that technique.
†: This amount of errors can be corrected only when the corresponding unit is paired with another such unit. When the unit is on its own this number is smaller.

Theoretical/real failure survival: For SECDED, each 4KB page is made of 512 64-bit blocks, meaning that in the presence of a good leveling of the failures, up to 512 failures can be corrected, but on average only 26.21 are. This is less than a 5%, and that is a poor ratio. DRM does somehow better, getting quite close to its capacity. Note that the 160 survived errors refers to a pair of pages. ECP_6 is, again, not very good in this respect: there are 64 blocks in each page, accounting for a total 384 ECPs, but on average, only 149.16 are used, which is below 50%. To end with, CEPRAM corrects, on average 22.611 failures per block (1447.11/64), which is close to the "at least" 30 per block pair, but not as close DRM is to its limit.

The reason behind CEPRAM being closer to the theoretical limit is, mainly, because cell lifetime follows (and is modelled as) a normal distribution, therefore, lifetimes are clustered around the average. If a scheme is able to survive those failures outlaying in the curve, then the errors are *evenly spread* among blocks. One example of the opposite behavior can be found on SECDED. Given that the failure survival ability is so small, the probability of having a byte with two cells having shorter lifetime in a block is quite high, leading to early failure of pages.

This analysis shows that although the wear is spread evenly through all the memory cells, there are many failure-surviving-resources not used at the point of failure. Developing techniques to make a more efficient usage of this resources is out of the scope of this work.

4.6.6. Dynamic analysis

In this section, we show and discuss the dynamic behavior of CEPRAM, comparing it with the aforementioned techniques.

Figure 4.10 shows curves for all the techniques. This curve containing the point (x, y) means that after x billions writes to every page (wear leveling is assumed), the y% of the memory pages are still usable. Same graph as shown in [64], although the scale changes because they assume that in each write, a bit is flipped with probability $p = 0.5$. Again, given that CEPRAM is built on top of ECP, it is bound to improve it.

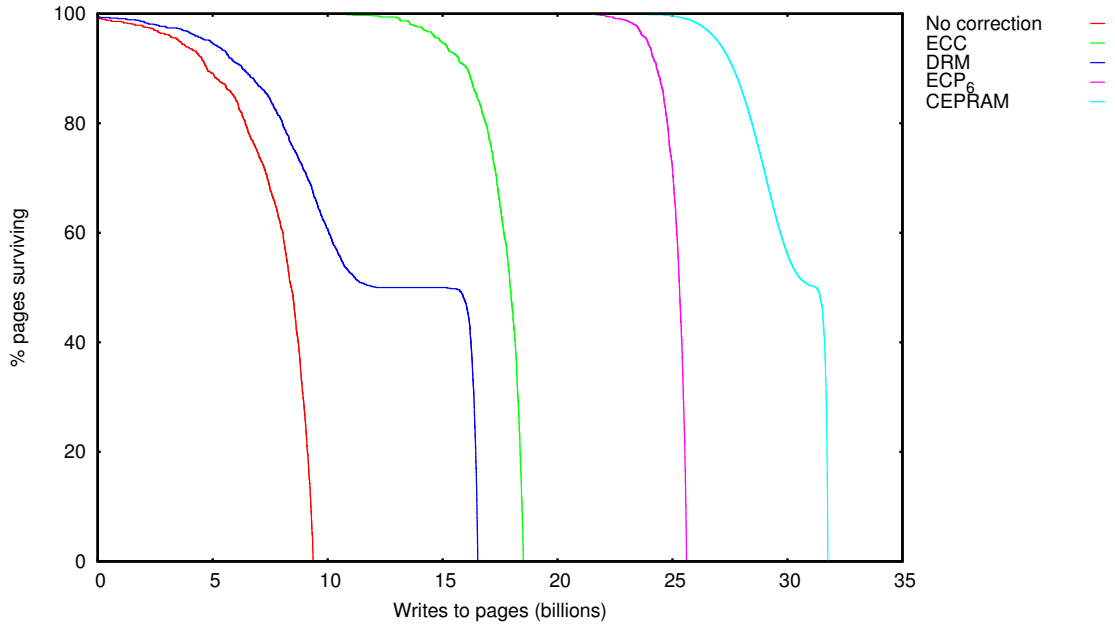


Figure 4.10: Percentage of pages functional as the time passes. Time is measured in blocks written back (to physical pages).

This graph is much more illustrative of the effectiveness of the schemes when it comes to keeping as much memory alive as possible. According to Table 4.12, the number of failures per page survived by ECP_6 and DRM is quite close. Nevertheless, the lifetime of a system implementing DRM is much shorter. This is due to dynamic behaviour. More precisely, pages are discarded upon first failure, rapidly decreasing the memory size, making other pages *absorb* writes to those discarded pages, increasing the pace at which cells wear out, leading for an early system failure. The other techniques are somehow characterized by the appearance of the first failure, moment in which, due to the aforementioned clustering of failures when getting closer to the average cell lifetime, and to the increased wear ratio due to page failures, the amount of memory available decreases exponentially, quickly leading to a system failure. CEPRAM increases the lifetime beyond ECP due to the fine-grain pairing (as opposed to the coarse grain pairing of DRM). This pairing requires a technique to survive the existing failures. DRM uses parity to mark *dead* bytes in the primary page, CEPRAM uses compression with backspace symbol to encode the information plus the failure information. There is a small phase in which the block is compressed with C-Pack, without backspace, encoding the bits to be discarded in a mask, prior to the pairing. Our experiments show that the worst case is frequent enough to make the length of this phase negligible, showing that the strength of CEPRAM is not as much in the compression as a means of reducing size, but as a means of encoding both the data and information about errors in the block.

The results above correspond to *dealII* which is an *average* application in terms of bit-flip probability, which is a measure of how wear-some the application is to the memory. If we take a look at $gobmk_5$ and *lbm* which are the application with the lowest and highest bit-flip rate depicted in Figures 4.11 and 4.12 respectively. Looking at these two figures, we can observe that CEPRAM is specially powerful

when the amount of flips is high. This is counterintuitive, because less flips mean lower entropy which usually translates in better compressibility. This happens because when the wear rate is high, cells suffer a lot of wear before an overflow happens, meaning that a lot of errors can be survived. On the other hand, if the wear ratio is low, an overflow will take place before the toll is to high on the cells, not leveraging the extra error correction capacity provided by CEPRAM.

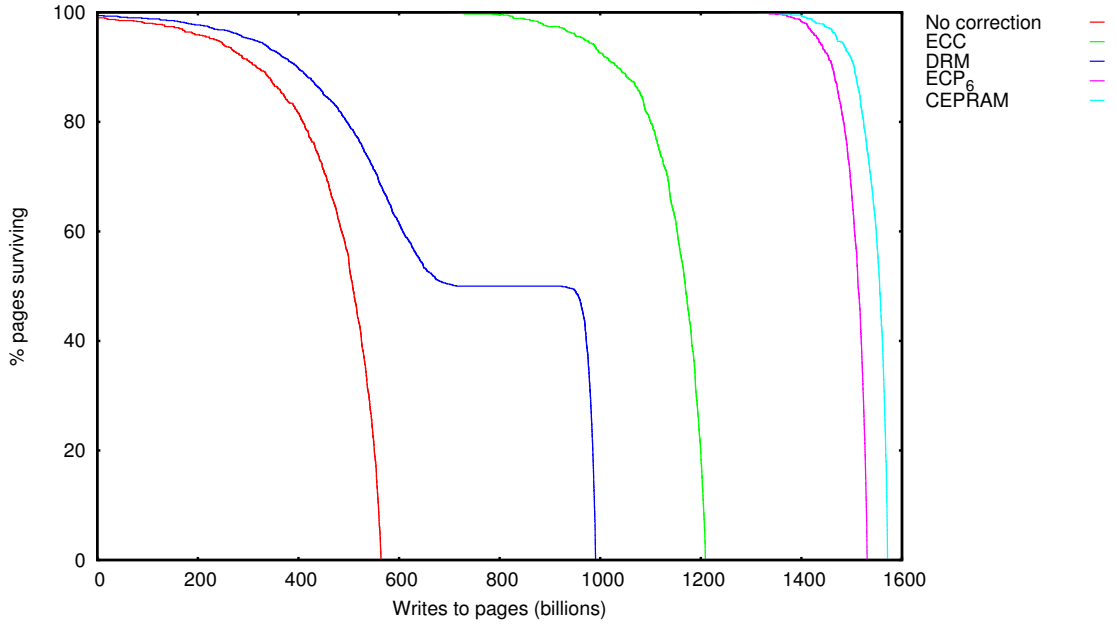


Figure 4.11: Percentage of pages functional as the time passes for the flip/overflow probabilities of $gobmk_5$. Time is measured in blocks written back (to physical pages).

4.6.7. Ideal case study

To finish the evaluation, we present some discussion of the *ideal scheme* that can correct up to n errors with a 12.5% overhead (*Id n*). Figure 4.13 shows the available percentage of pages after a given number of writes.

Surviving the first failure almost doubles the lifetime of the system. To double that time again requires surviving 8 failures. ECP_6 does slightly better than *Id 6*. This is not weird, taking into account that failures can be hidden. For example, if two cells fail, and due to wear leveling techniques they lay in the same pointer, the two errors are hiding each other, making one less pointers necessary, saving some wear. This delays the death of some pages, reducing the write-pressure over the rest of the pages, slightly spanning the lifetime. If the system failure is triggered by the available memory size falling below 50% as done in [69], CEPRAM is halfway between *16* and *32*. On the other hand, if we are more restrictive and use the ratio $\frac{1}{\sqrt{2}}$ *i.e.* 70.7% CEPRAM is placed between ECP_6 and *Id 16*, around $\frac{4}{5}$ of the way closer to the latter.

Figure 4.14 shows the accumulated number of *stuck-at* failures as a function of the lifetime. When the first failures start to happen, correcting one failure significantly

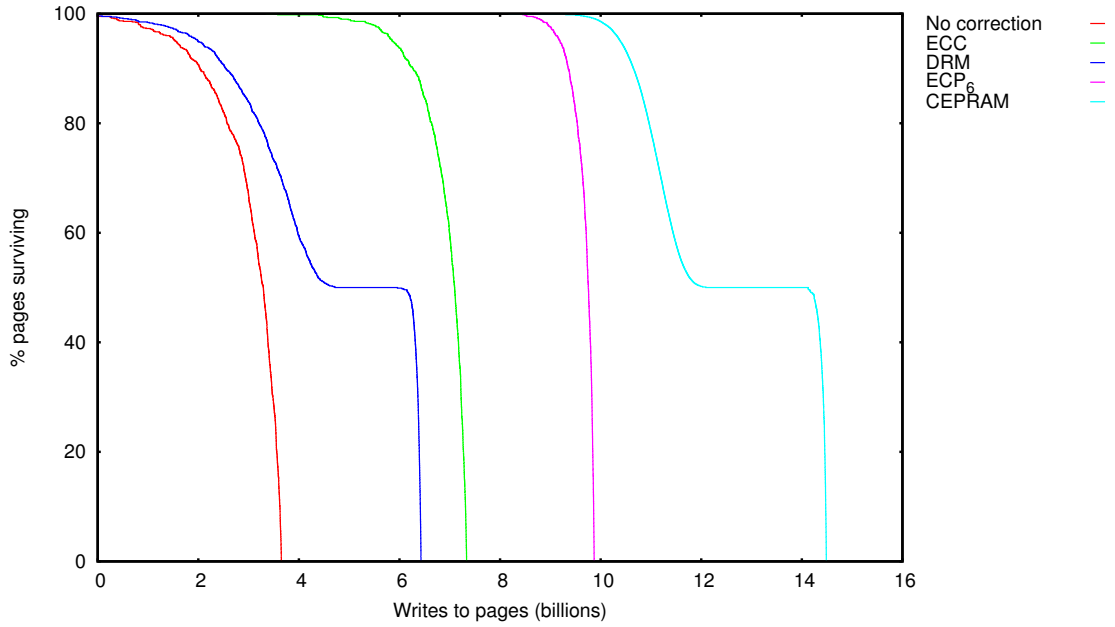


Figure 4.12: Percentage of pages functional as the time passes for the flip/overflow probabilities of *lbm*. Time is measured in blocks written back (to physical pages).

improves lifetime, but as soon as we correct the 29^{th} failure, the lifetime gain per corrected failure drops down below 0.5%failure. We think that this low return of investment makes correcting more than 29 failures unprofitable. According to this reasoning CEPRAM does a great job pushing the lifetime curve towards the Id 32 curve shown in Figure 4.13.

4.7. Conclusions

Resistive memories are closer to industrial adoption every day. That makes it important to develop techniques to overcome the limitations they present, so they can substitute DRAM, improving the memory in aspects such as scalability, low leakage.

In this chapter we have introduced CEPRAM, an attempt to make a PRAM system more durable through memory block compression. Our technique is built on top of ECP, and using a high-performance, cache-oriented compression algorithm, modified to better suit our purpose. It manages to further extend the lifetime of the memory system. In particular, it guarantees that at least half of the physical pages are in usable condition for 25% longer than *ECP*, which is slightly more than 5% more than an scheme that can correct 16 failures per block.

In addition to presenting the technique, we do an analysis of why PCM is not a very good context for compression, and it is probably not a good idea to invest efforts in it, unless compression is conceived differently: focusing more on improving the worst case scenario.

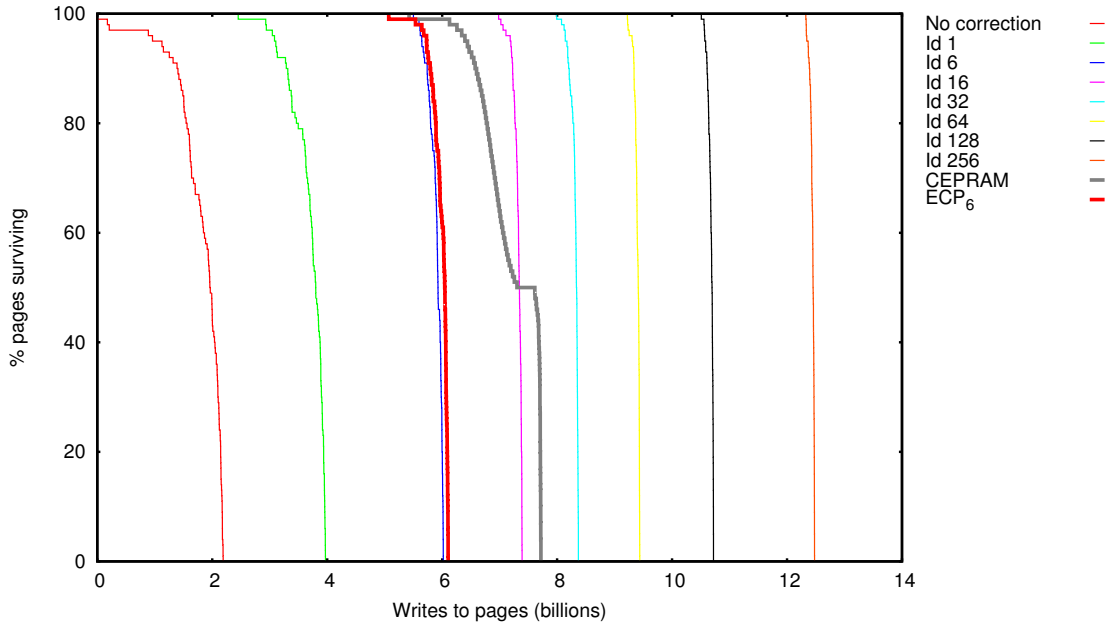


Figure 4.13: Percentage of pages functional as the time passes. Time is measured in blocks written back (to physical pages).

Finally, we show an study on the memory life time improvement as a function of the error correcting ability, offering some insights to help choose a target failure recovery ability depending on the lifetime expansion we want to get. This study shows that there is room for improvement beyond CEPRAM, although CEPRAM pushes the lifetime to a point in which the cost of extending the lifetime any further is quite high in terms of the amount of per-block error correcting ability.

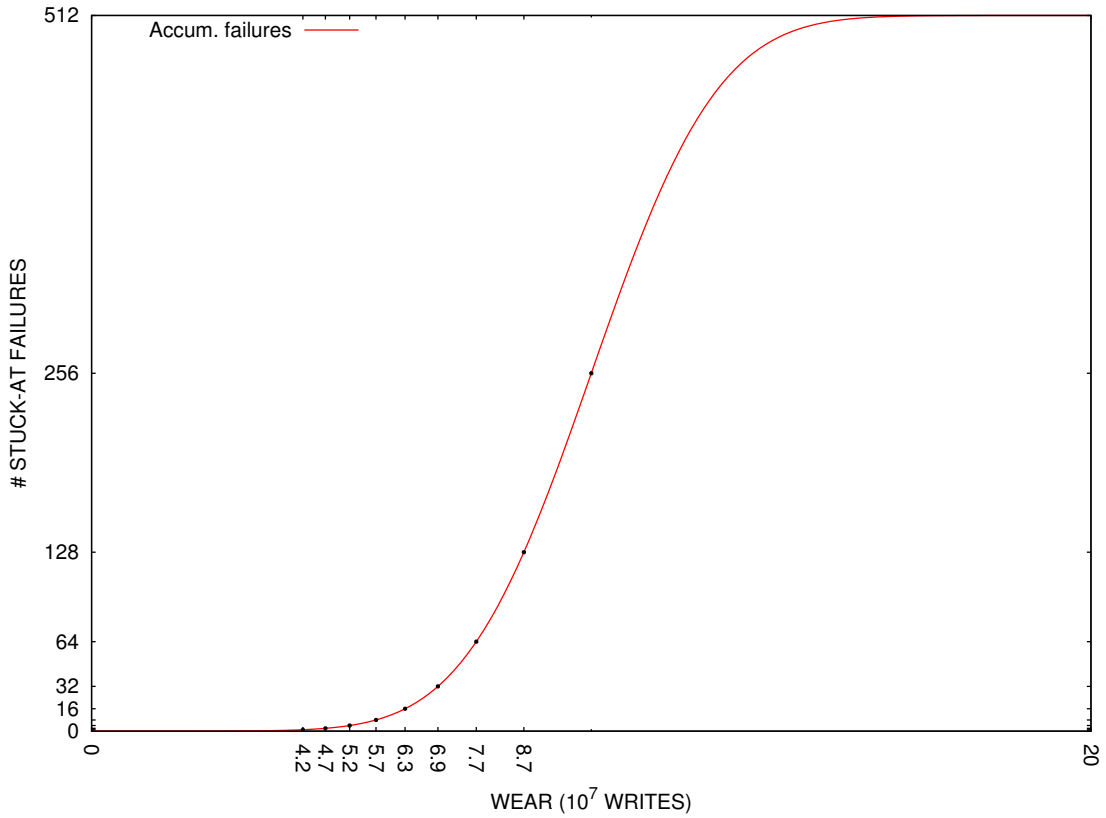


Figure 4.14: Accumulated count of errors in a 512-bit block as a function of the number of writes to the block, assuming a write modifies all the block. If wear is leveled, the fact that not all the block is written merely changes the scale of the x axis.

Chapter 5

Conclusions and future work

The memory system is an important part of every computer system, and so is shown by the amount of research that has been done since the dawn of computers as we know them today.

The memory system is a vast topic, presenting many challenges. This challenges come from different sources. Some of them are intrinsic, like power or performance, some of them appear to supply the need of software in different points in time, as for example, the need of an operating system to isolate processes memory map, what gave birth to main memory, and presented performance issues that were solved through the use of a TLB. Other challenges find their root in the technology. PCM is a good illustration: the adoption of this new technology brings along some new issues to deal with in exchange for all the advantages the new technology may offer. There are also some application-derived challenges. Some security issues have not been a problem until the technology gave us the opportunity to, for example, do operations as payments or banking on the internet. Or, as another example, tracking memory accesses for debugging was not that helpful prior to the spread of shared-memory computers.

Instead of focusing in a problem, or a specific part of the memory, in this work we have tackled three big problems from the point of view we considered the most interesting.

Our Stack Filter (SF) relies on the "make it simple" approach to reduce power. It leverages the information provided by a special purpose register to know which memory references address the stack, leveraging the special sematics of this access that provide this region with special locality properties. This locality can be exploited with a really simple structure. Our results also show that this structure doesn't need to be large to capture a fairly big amount of the accesses to the stack and drive them through a low power part of the hierarchy rather than the more flexible, power hungry data cache. Our experiments show that the power requirements can be reduced while the performance remains largely unchanged.

Along with power there is a big concern about computers being reliable both in terms of the hardware and the software components. Software bugs lead to sit-

uations that may damage the company behind a buggy program, lead to data loss/corruption, problem in communications with spatial probes and many undesirable scenarios. As time passes and computer systems become bigger, so do the applications running on top of those systems, and with it the difficulty to test and verify. This difficulty gets dramatically increased when a program evolves from running in a single processor into running in a multi-processor machine in which threads communicate. The non-deterministic behavior introduced by this change is a motivation for the development of techniques to track memory references to ensure that the semantics intended by the programmers are preserved by the used synchronization primitives.

Tracking memory references is a challenging task, it imposes constraints in the performance and the memory overheads incurred in by the different techniques. Hardware techniques don't suffer from the performance, but have limited resources and sofware techniques are usually too slow. With AccB we offer a hybrid alternative that is not yet ripe to be an *always-on* feature, but offers a speed up over commercial software-only detector with just a small hardware modification. Compared to hardware-only detectors, it is much slower, but the hardware requirements are really small, and the flexibility provided by the software part make it competitive. One example of the flexibility is the possitiblity to combine it with techniques like Aikido [89] that can reduce the extents of the software that are instrumented to a bare minimum drastically speeding up detection with realistic hardware support.

The evolution of computers has not only accentuated the need for tools to help testing software for correctness, but has also pushed some components and technologies to their limits. One such technology is DRAM. In the face of a limit in scalability, manufacturing companies need a replacement to deal with this issue. Phase Change technology is a promising candidate to replace DRAM as technology for main memory in comodity systems. The consideration of PCM as a replacement brings into consideration new challenges for the industry and the academia. We have addressed the limited lifetime of a PCM device from a different perspective. So far, no one has used compression as a means to provide backspace capacity in the data of a block. With this extension, we are able, after (fine-grain) pairing, to survive around 30 exposed failures per block, that turn into a much higher number, because the probability of a falure to get exposed is 0.5 on average. CEPRAM goes further in the direction of extending the lifetime of PCM-based devices. We also try to provide enough information to back up the claim that although we have proved it beneficial, this is not a good context for compression as it is conceived today, and therefore discourage spending resources in refinements of the compression algorithm. In addition, we do an analysis of the point beyond which the benefits of correcting one extra fault are not big enough to make it worth the effort, and have shown that CEPRAM achieves results not very far from that point, making it a close-to-a-reasonable-bound approach.

Our final conclusion is that the memory system is a vast area in which there are many issues to deal with. Some of them are constant, other are motivated by changes in the technology or the organization of computers, and others arise to supply a need of the software part of a system. Regardless of the origin all of

them are important and have to be addressed. This thesis tackles three problems, improving the state of things, but there is much work to do to achieve the final solution, if it exists.

5.1. Future work

In this thesis we have alleviated the problems we have adressed. However, the work is far from done. There is still much room for improvement, and short term goals to bring the status closer to a final solutions, among which we highlight the following:

- Taking semantic filtering beyond DL1 cache. We have shown the potential of knowing which memory region a piece of data lies in. This can be used at other levels of the hierarchy. Aikido [89] provides a way to tell appart shared pages from private pages. This can be used to filter coherence traffic by declaring data private to threads and shared read-only data as non coherent, and declaring shared data as coherent. This can be done to reduce coherence actions to the coherent set. Semantic filtering can also be applied to a PCM main memory: Some segments of a process fit better in PCM (text, read-only data), some other fit better in DRAM (the stack), and parts of the heap may exhibit different behaviors that will make them a better candidate for PCM or DRAM. This insight can be used to make a hybrid hierarchy, part DRAM, part PRAM.

- The challenges offered by PCM have been addressed mainly at PCM level itself. The whole hierarchy can help, because it is the resources in the hierarchy and the way they are managed what determines which piece of data lies where. There are techniques implemented at register level [90], and at LLC [78]. There is still room for improvement from the point of view of the cache management: replacement policy, inclusive vs non-inclusive, and other aspects that may reduce, directly or indirectly, the amount of writebacks. This aspects also have an impact on PCM, but given that the power/latency model of PRAM differs largely from the one of DRAM all the side effects have to be taken into account and balanced to fit the new model and constraints.

5.2. Publications

This work has produced the following publications:

- R. Gonzalez-Alberquilla, F. Castro, L. Piñuel and F. Tirado. Stack Oriented Data Cache Filtering. In *International Converence on Hardware/Software Codesign and System Synthesis*, pages 257-266. Grenoble, France, October 2009.

- R. Gonzalez-Alberquilla, F. Castro, L. Piñuel and F. Tirado. Stack Filter: Reducing L1 Data Cache Power Consumption. In *Journal on Systems Architecture* 56(12), pages 257-266. December 2010.
- R. Gonzalez-Alberquilla, K. Strauss, L. Ceze and L. Piñuel. Accelerating Data Race Detection with Minimal Hardware Support. In *Proceedings of the 17th international conference on parallel processing* Europar'11, pages 27-38. Bordeaux, France. September 2011.
- R. Gonzalez-Alberquilla, F. E. Frati, K. Strauss, L. Ceze and L. Piñuel. Data Race Detection with Minimal Hardware Support. To appear in *Computer Journal*, Oxford University Press.

Appendix A

Algorithms

This sections contains the algorithms regarding C-Pack and the writing process for error correction. Algorithms A.2 and A.3 are just the pseudocode for the modification of the algorithm in [65].

Taking a look at Table 4.4, third column, the bits in parethesys are the *header* of a symbol, and the rest (if any) are the *payload*. We call *information* symbol to all of them that encode actual information, which is all but the $< bs >$. Also, the word *intended* is used to refer to what would be written if there were no faults, and *actual* means what was written in a maybe-faulty piece of memory.

For clarity, we would show first how the decoding process works. Algorithm A.1 shows the pseudocode for the $< bs >$ interpretation stage prior to decode. The process consists in bit-masks, shifts and appendations. This stage proceeds as follows:

- Upon initialization, the offset of the first symbol is set to 0.
- Symbols are decoded until a non-$< bs >$ symbol is found after the 16^{th} non-$< bs >$ symbol has been found.
- The biggest *information* symbol is a XXXX which is 34 bits, but the biggest symbol is $< bs >$ when the mask is "11111". That takes 48 bits. So the first thing is extracting the at-most 48 bits to be decoded.
- The *header* is then decoded. If it is a *information* symbol (line 20), then the offset of the next symbol is calculated adding the symbol length to the offset, and it is recorded in a vector.
- If the symbol is a $< bs >$ (line 8) the 5 bit mask is extracted and depending on the amount of ones in the mask, as many correction bytes are extracted (line 12). With the mask and the correction bytes, the input is modified: first, the mask is applied as depicted in Figure 4.5, and after that the $< bs >$ symbol is removed (by shifting the remaining bits, and appending them to the previous bits).

- Given that the correction may have altered the previous symbol, thus, modifying where to look for the next one. For that reason, we have to "reconsider" the previous symbol (lines 17,18).

Once the $< bs >$ processing is over, the decoding cand proceed as usual. We introduce the following scenarion to illustrate the situation dealt with in the last item: A MMXX symbol is found at offset 50. This symbol is followed by a backspace, so it looks like:

$$...1100iiiibbbbbbbbbbbbbbbb111mmmmm...$$

Where "1100" is the *header*, *iiii* is the index in the dictionary of the matched word and the *bb...b* are the two bytes that don't match. Next symbol is at offset $50+length(MMXX) = 50+24 = 74$. It is a $< bs >$, this is "111". If the correction bytes state that there is an error in the fourth bit, the bits are transformed into:

$$...1101iiiibbbbbbbbbbbbbbbbNNNN...$$

Where $NN...$ are the bits that would be considered for the next symbol, at offset 74 as previously calculated. This is wrong, because if we reconsider the previous symbol (at offset 50), the correction transformed it into a MMMX, which is just 16 bits wide, what means that the next symbol starts at offset $50+length(MMMX) = 50+16 = 66$. This is the reason for the need to decoding the previous symbol again.

Applying $< bs >$ this way makes them "independant" in the sense that any symbol decoded to be a $< bs >$ is immediately applied instead of waiting for any subsequent $< bs >$ that could modify the former $< bs >$. There is, however, the posibility that a non-$< bs >$ symbol is corrected into a $< bs >$ in which case it is applied to the previous symbol. This "nesting" of $< bs >$ is what requires keeping track of all the previous symbols offsets.

To show that this process can be done by a combinational circuit, consider Figure A.1. The module has two inputs: the offset vector, and the input block. Ashifter places the symbol at the head for the decoder to calculate the size of the current symbol. If the current symbol is a $< bs >$, the decoder is also in charge of extracting the mask and the correction bytes. If the symbol is not a $< bs >$ the correction circuit doesn't modify the input, and the output offset vector is calculated appending the addition of the symbol length and the current offset to the input offset vector (10 bits). If the symbols is a $< bs >$ the correction circuit is in charge of applying the correction based on the previous symbol offset and the input, and of removing the $< bs >$ symbol. This is done shifting the correcting bytes into position before XOR-ing them with the input, masking out from the $< bs >$ symbol on, and OR-ing it with the bits after the $< bs >$ symbol shifted into place. In order to preprocess the input, we just need to cascade N unit, where N is a bound to the number of (exposed) faults we are willing to correct, along with a control unit to count how many non-$< bs >$ symbols have been found to know from which of the modules take the output.

Algorithm A.4 shows the pseudocode for the writing of the compressed data symbol by symbol in the memory. The pseudocode assumes that $output(bits)$ gets a string

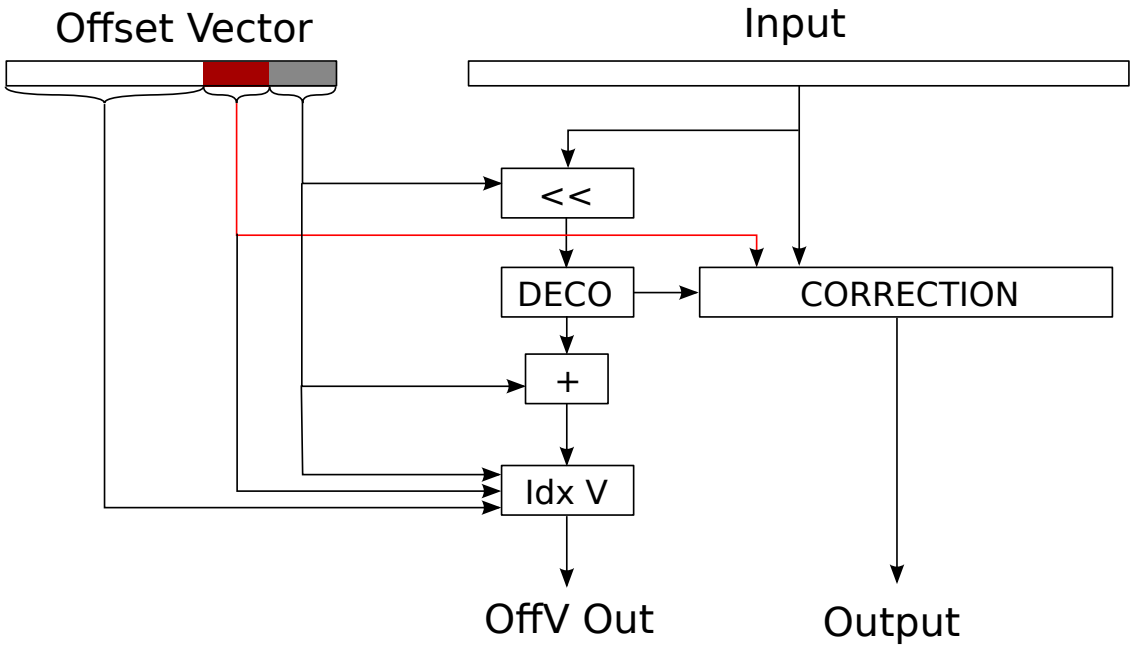


Figure A.1: $< bs >$ expansion in C-PackBs.

of bits that outputs to the PCM row or row pair. The call *checkOutput*() returns the value in PCM of the bits that have just been written. This is always done in PCM to detect stuck-at faults. The algorithm is modeled as a state machine, to get a better idea of how it should be implemented on hardware. Initially, the operation can have 4 outcomes:

- No failures exposed. The writing process finishes.
- A failure has affected the *payload*. The hardware just needs to output a correction for those bits.
- A failure has affected the *header*, changing it into anything but a $< bs >$. This situation is more delicate, because it changes where the decoding hardware will look for the next symbol, so the circuit needs to fix this. this is done by, if the *intended* symbol is shorter, the extra room must be filled with bits from the next symbol. If the *intended* symbol is longer, the "tail" is appended after the backspace symbol.
- A failure has transformed the *header* into a $< bs >$. The way of dealing with that is by "neutralizing" the backspace. It can be done in 2 ways. First attempt is to clear the 5-bit mask after the $< bs >$-*header*. That way the $< bs >$ will be ignored, because no corrections are requested. If that fails, the decoding hardware is going to expect as many "correction" bytes as ones (stuck-at failure induced) are in the mask. If those bytes are 0, then the $< bs >$ would also be neutralized. In each case, the way to proceed is to retry writing the initial symbol. If that fails and bytes are flipped by mistake, then the hardware needs to replicate any correction. Due to the self-cancelling

property of a bit-flip, the way to correct a non-needed bit-flip is to flip that bit again. This feature allows the decoder to do a greedy decoding of backspaces, applying corrections as soon as they are found.

There are four states transitions for each of the outcomes. The subsequent transitions take care of the actions needed to address the errors derived by the faults.

Algorithm A.1 C-Pack: Pre-decode stage

```
1: decodeWords ← 0
2: symbolStart ← 0
3: input ← getInputBlockPair()  /* Read the whole thing */
4: while symbolsFound < 17 do  /* We need to make sure there is no < bs > after the 16^th symbol */
5:     symbolBits ← extract(input, symbolStart, 48)  /* Chunck for decoding */
6:     symbolType ← decodeHeader(symbolBits)  /* Decode symbol */
7:     if symbolType =< bs > then /* If symbol =< bs > apply it */
8:         /* Get the 5-bit error mask */
9:         mask ← extract(symbolBits, 3, 5))
10:        faultyBytes ← population(mask)  /* Count population */
11:        /* There are as many bytes as ones in the mask */
12:        correctionBytes ← extract(symbolBits, 8, faultyBytes * 8)
13:        /* Apply correction to previous symbol */
14:        input ← applyCorrection(input, symbolStartX[symbolsFound − 1],
15:                                mask, correctionBytes)
16:        /* Reconsider previous symbol, in case the header changed */
17:        symbolStart ← symbolStartX[symbolsFound − 1]
18:        symbolsFound ← symbolsFound − 1
19:    elseIf not a < bs > go to next symbol
20:        symbolStartX[symbolsFound] ← symbolStart
21:        symbolsFound ← symbolsFound + 1
22:        symbolStart ← symbolStar + length(symbol)
23:    end if
24: end while
25: /* Do normal HW C-Pack decoding */
26: return CPackDecode(input)
```

Algorithm A.2 C-Pack: compressWord(in)

if $in = [00000000]_{16}$ **then**
/* ZZZZ output 00 */ **return** $[00]_2$
else if $fullMatch(in, idx, Dictionary)$ **then**
/* MMMM output 01bbbb where bbbb is the index in 4 bits */ **return** $< [01]_2, idx_4 >$
else if $3byteMatch(in, idx, Dictionary)$ **then**
/* MMMX output 1101bbbbB where bbbb is the index in 4 bits, and */
/* B is the input less significant byte */ **return** $< [1101]_2, idx_4, in_8 >$
else if $2byteMatch(in, idx, Dictionary)$ **then**
/* MMXX output 1100bbbbBB where bbbb is the index in 4 bits, and */
/* BB are the input two less significant byte */ **return** $< [1100]_2, idx_4, in_{16} >$
else
/* XXXX output 10BBBB where BBBB is the input word */ **return** $< [10]_2, in >$
end if

Algorithm A.3 C-Pack: compress(in[16])

for $i = 0$ to 15 **do**
$output(compressWord(in[i]))$
end for

Algorithm A.4 C-Pack: outputCorrect(out)

```
1:  state ← START
2:  indendedOuput ← compressWord(in[i])
3:  while state ≠ END do
4:      output(intendedOutput)
5:      actualOutput ← checkOutput()
6:      switch ( state )
7:      case START :
8:          if actualOutput = intendedOutput then
9:              /* Everything went fine */
10:             state ← END
11:         else if decode(actualOutput) = decode(intendedOutput) then
12:             /* The kind of symbol has not changed */
13:             intendedOutput ← buildCorrection(intendedOutput, actualOutput)
14:             state ← CORRECT
15:         else if decode(actualOutput) ≠ BS then
16:             /* The symbol kind has changed, but not to BS */
17:             intendedOutput ← buildCorrectionHead(intendedOutput, actualOutput)
18:             state ← CORRECT_SYM
19:         else /* decode(actualOutput) = BS */
20:             /* A faulty bit has turned the output into a BS */
21:             push(intendedOutput)
22:             /* Clear the error mask to prevent undesired bit-flips */
23:             intendedOutput ← assemble(BS,'00000')
24:             state ← ARTIFICIAL_BS
25:         end if
26:     case CORRECT :
27:         if actualOutput = intendedOutput then
28:             /* Everything went fine */
29:             intendedOutput = pop()
30:             if intendedOutput = φ then
31:                 state ← END
32:             else
33:                 state ← START
34:             end if
35:         else if decode(actualOutput) = decode(intendedOutput) then
36:             /* The BS header is okay, but something went wrong */
37:             intendedOutput ← buildCorrectionTail(intendedOutput, actualOutput)
38:             state ← CORRECT
39:         else /* decode(actualOutput) ≠ decode(intendedOutput) */
40:             /* The BS header got corrupted */
41:             intendedOutput ← buildCorrectionHead(intendedOutput, actualOutput)
42:             stat ← CORRECT_SYM
43:         end if
```

```
44:     case CORRECT_SYM:
45:         if actualOutput = intendedOutput then
46:             /* Everything went fine */
47:             intendedOutput = pop()
48:             if intendedOutput = φ then
49:                 state ← END
50:             else
51:                 state ← START
52:             end if
53:         else if decode(actualOutput) = decode(intendedOutput) then
54:             /* The header is okay */
55:             intendedOutput ← buildCorrection2(intendedOutput, actualOutput)
56:             state ← CORRECT
57:         else /* decode(actualOutput) ≠ decode(intendedOutput) */
58:             /* The BS header got corrupted */
59:             intendedOutput ← buildCorrectionHead2(intendedOutput, actualOutput)
60:             stat ← CORRECT_SYM
61:         end if
62:     case ARTIFICIAL_BS:
63:         if actualOutput = intendedOutput then
64:             /* The mask has been successfully cleared: retry */
65:             intendedOutput ← pop()
66:             state ← START
67:         else /* getBSMask(actualOutput) ≠′ 00000′ */
68:             /* The mask has not been cleared. Output 0x00 bytes */
69:             intendedOutput ← build0(population(getBSMasc(actualOutput)))
70:             state ← CLEAR_CORRECT
71:         end if
72:     case CLEAR_CORRECT:
73:         if actualOutput = intendedOutput then
74:             /* The correction bytes have been successfully cleared: retry */
75:             intendedOutput ← pop()
76:             state ← START
77:         else
78:             /* Clear the fault-induced corrections and then retry */
79:             intendedOutput = buildCorrection(intendedOuntput, actualOutput)
80:             state ← CORRECT
81:         end if
82:     end switch
83: end while
```

Apéndice B

Resumen

B.1. Introducción

El subsistema de memoria es un componente clave de las computadoras. Cualquier cambio en sus características tiene un impacto directo en todo el sistema. Por ello, tanto la comunidad académica como la industria han invertido gran cantidad de esfuerzos en mejorarlo. En un principio el objetivo era mejorar el rendimiento, pero gradualmente el interés se ha ido expandiendo para incluir eficiencia energética, robustez, soporte a la depuración y escalabilidad en el proceso de integración. En este contexto, las tecnologías resistivas han surgido como solución a los problemas de escalabilidad que presentan las tecnologías de memoria actuales. Sin embargo, estas nuevas tecnologías sufren desgaste y se inutilizan rápidamente, lo que requiere de técnicas específicas para prolongar su vida útil. Por ello, en los últimos años este es también un aspecto del diseño del sistema de memoria que ha recibido una notable atención.

El *rendimiento* es un centro de atención constante ya que un gran porcentaje de las instrucciones dinámicas hacen uso del subsistema de memoria. Los procesadores modernos cuentan con múltiples mecanismos para reducir u ocultar su elevada latencia, tales como *caches*, unidades de prebúsqueda, mecanismos de ejecución fuera de orden, ... Aún así, cualquier mejora del rendimiento de este subsistema suele traducirse en una ganancia de rendimiento global del sistema.

El *consumo de energía* se ha convertido en un aspecto crítico en la última década. El consumo de energía es un problema que tiene dos consecuencias diferenciadas. Por un lado está el consumo de energía en sí mismo. En el caso de los sistemas portátiles, alimentados por baterías, reducir el consumo es vital para prolongar la autonomía del sistema así como para reducir el peso del mismo, que son dos características – autonomía y ligereza – que todos los fabricantes quieren ofrecer a sus clientes. En el caso de los sistemas conectados a la red eléctrica, reducir el consumo de energía es necesario para reducir el coste de la factura eléctrica, cada vez mayor. Por otro lado, está el problema de la densidad de potencia. Aunque pudiéramos proporcionar electricidad al sistema en todo momento, la potencia se convierte en calor que tiene

que ser disipado y hace ya tiempo que se alcanzó la denominada barrera energética, *power wall [1]*, un límite en la cantidad de potencia que un chip puede disipar por unidad de superficie sin sobrecalentarse. Estos dos problemas, consumo de energía y densidad de potencia disipada, han sido objeto de múltiples investigaciones y son críticos a la hora de afrontar el diseño del subsistema de memoria.

La *robustez* es también una preocupación constante. Las variaciones durante el proceso de fabricación, la radiación y otros factores afectan a la fiabilidad de la memoria. Por ejemplo, las celdas pueden sufrir inversiones ante el impacto de rayos cósmicos, o por efecto de la radiación, y es preciso detectar y corregir esos fallos temporales. Técnicas como ECC [2] han ganado popularidad y son una característica común en servidores.

Asistencia a la depuración es una característica deseable de cualquier computadora, y el sistema de memoria puede ayudar de múltiples maneras. En un programa secuencial, el orden de los accesos a memoria depende únicamente del programa y del sistema operativo. En un programa multihilo, este orden también depende del estado de todos los procesadores que comparten la memoria y del estado de la propia memoria, reduciendo el determinismo del propio programa. La probabilidad de que algún posible intercalado de accesos no sea tenido en cuenta por el programador es alta lo cual es una fuente de errores. Además, no sólo es más probable introducir errores en aplicaciones multihilo – carreras de datos o violaciones de atomicidad – si no que además, estos errores son más difíciles de recrear y depurar que los secuenciales, debido a la carencia de determinismo, lo que complica la reproducción de las condiciones bajo las que se manifiestan. Es por lo tanto imprescindible desarrollar herramientas HW y/o SW que ayuden al programador en esta labor.

La *escalabilidad* es más bien un parámetro tecnológico, pero también es una propiedad de la memoria. La tecnología más común a día de hoy para memoria principal, DRAM, ha alcanzado los límites de escalabilidad. Tiene problemas para bajar de los 30nm [3]. Esto ha llevado a los investigadores a considerar nuevas tecnologías. Algunas de estas tecnologías son *Phase Change Memory (PCM)*, *Spin Torque Transfer RAM (FTT-RAM)* o *Ferroelectric RAM (FRAM)*. Mientras que DRAM es una tecnología capacitativa, es decir, los valores se almacenan en forma de carga en un condensador, estas nuevas tecnologías son resistivas o magnéticas. En estas tecnologías las celdas están hechas de un material que puede alterar sus propiedades físicas, modificando su impedancia eléctrica, permitiendo codificar valores como distintos valores de impedancia. Todas estas tecnologías no tienen problemas en reducirse por debajo de 16nm, pero comparten un nuevo problema: el rápido desgaste que suponen los cambios de estado. Este cambio de escalabilidad por durabilidad hace que estas tecnologías no estén aún maduras para su uso comercial, y motiva nuevas investigaciones orientadas a mitigar este problema.

En esta tesis proponemos varias técnicas para resolver o aliviar algunos de los problemas mencionados:

- *Consumo:* Hemos atacado el problema del consumo sacando partido de la información que el hardware puede inferir sobre la semántica de los accesos a memoria.

- *Ayuda a la depuración:* Las carreras de datos son el objeto de nuestro trabajo en este ámbito. En concreto, hemos propuesto un mecanismo híbrido HW/SW para monitorizar la ejecución de un programa *al vuelo* y, en caso de carrera de datos, informar.
- *Escalabilidad/Durabilidad:* En el contexto de la tecnología PCM hemos puesto nuestra atención en hacer que los bloques de memoria sean útiles el mayor tiempo posible. Esto requiere detectar y sobrevivir a tantos fallos como nos sea posible, para lo que hemos propuesto un nuevo mecanismo de gestión de este tipo de memorias.

Cada aspecto ha sido abordado dentro de un nivel distinto de la jerarquía de memoria. Por ello, a continuación presentamos un compendio de esta jerarquía multinivel.

B.1.1. El sistema de memoria como jerarquía multinivel

La memoria está organizada como una jerarquía multinivel. En la parte de abajo están situados los registros arquitectónicos. En la parte de arriba está la memoria principal. Desde su primera aparición en 1963, las memorias cache han sido incluidas en múltiples sistemas como niveles intermedios debido a las ventajas que suponen.

Cada nivel en la jerarquía tiene 4 características intrínsecas a la tecnología usada en su construcción: latencia, consumo, escalabilidad y coste (dinero) por byte. Los niveles inferiores, más cercanos a la CPU tienen menor latencia, pero por otro lado, a mayor cercanía mayores son las restricciones de tamaño y latencia, lo que se traduce en un mayor coste en términos energéticos, monetarios o ambos. El tamaño de cada nivel se decide como un compromiso entre coste, consumo y latencia, ya que las estructuras cache ven incrementados su consumo y latencia con el tamaño. Hay otros aspectos funcionales, dependientes del diseño, como son el tamaño de bloque, cómo se alojan los bloques de memoria principal dentro de la cache, cómo se elige un bloque para desalojar cuando hay que hacer sitio para uno nuevo (política de remplazamiento), inclusividad, propagación inmediata de escrituras a niveles superiores o no, y si los bloques son emplazados o no cuando falla un acceso de escritura. Estas decisiones de diseño han sido investigadas en profundidad y distintos fabricantes tienen distintas especificaciones en sus productos.

Esta jerarquía es un poco más compleja en máquinas multiprocesador. En estos sistemas hay, normalmente, una memoria principal compartida por todos los procesadores. En este caso el uso de caches es posible, pero los niveles pueden ser, por diseño, privados a cada procesador, compartidos entre un grupo, o compartido por todos. Si un nivel es compartido por todos, no hay ninguna complejidad extra, a parte de la interconexión. Sólo se requiere que todos los procesadores tienen que comunicarse con la cache como si se tratase de la memoria principal. Si un nivel es privado a cada procesador, es deseable que la jerarquía mantenga la coherencia. Una cache mantiene la coherencia cuando todos los procesadores observan las modificaciones a una dada posición de memoria en el mismo orden. Esto requiere que ninguna caché mantenga copias desfasadas del dato. En otras palabras, en los

sistemas multiprocesador la jerarquía de memoria tiene otras dos características importantes: la *coherencia* y la *consistencia*. La consistencia atañe al orden relativo de las operaciones de memoria observado por distintos procesadores. Hacer que un sistema de memoria sea coherente es una tarea relativamente sencilla, pero asegurar que todos los procesadores del sistema observan las operaciones de memoria en el mismo orden es una tarea ardua. Por ello existen varios modelos de consistencia y cada sistema declara por cual se rige, de manera que los desarrolladores son conscientes de que pueden esperar del sistema y qué no.

Es preciso aclarar que en esta tesis nos centramos en la parte física del sistema de memoria. Todas las técnicas propuestas funcionan en un sistema con memoria virtual, pero ninguna requiere de los conceptos o mecanismos añadidos en las jerarquías de memoria virtual.

Los distintos niveles de la jerarquía sufren distintos problemas y ofrecen distintas posibilidades para atacarlos.

- El consumo y el rendimiento son problemas globales. Los niveles inferiores pueden usar su cercanía a la CPU para inspeccionar el estado de esta: registros, TLB, *etc.* y sacar partido de esta información. Los niveles mas altos, por otro lado, pueden sacar partido del relajamiento en las restricciones de latencia y de la mayor abstracción en el mapa de memoria para adaptar el tamaño y/o la geometrías de ésta.
- El soporte a la depuración es un aspecto que no se ata a ningún nivel en particular. Los niveles privados tienen cierto grado de información sobre compartición para garantizar la coherencia. Esta información puede ser de ayuda para analizar accesos y servir de ayuda en la depuración.
- La escalabilidad es un problema esencialmente de la memoria principal. Tradicionalmente los niveles inferiores están hechos de transistores, por lo que escalan con la tecnología de la CPU. En cambio la RAM usa otras tecnologías que no escalan tan bien.

B.1.2. Contribuciones de esta tesis

La jerarquía de memoria es un tema extenso, compuesto de muchos aspectos diferentes, todos ellos merecedores de atención. Para acotar el alcance de esta tesis nos hemos concentrado en tres de los problemas mencionados anteriormente, a saber, consumo de potencia, programabilidad/soporte a la depuración y escalabilidad/durabilidad. Para solventar estos problemas, hemos viajado desde la parte inferior de la jerarquía hacia arriba, ocupándonos en cada nivel del problema que nos parece más interesante de resolver o aliviar.

La primera parte de esta tesis se concentra en la frontera entre el primer nivel de cache y los registros, y en cómo reducir el consumo. La primera contribución es el *Filtro de Pila*: una estructura simple y pequeña que saca partido de la localidad espacial presentada por los accesos a la pila para filtrarlos usando una pseudo-cache de

nivel 0. Esta nueva estructura de correspondencia directa y tamaño reducido tiene, en términos energéticos, un coste bajo. Mostramos como en sistemas empotrados, un *Filtro de Pila* pequeño consigue un ahorro energético para nada despreciable con un bajo coste en términos de hardware requerido y bajo impacto en el rendimiento.

La segunda parte de esta tesis sube un poco en la jerarquía de niveles y trata de ayudar en el proceso de depuración de programas multihilo. La segunda contribución es un soporte hardware mínimo para detectar carreras de datos y un algoritmo, *Accessed Before* que usando dicho soporte puede detectar carreras de datos dinámicamente. Mostramos su fiabilidad y comparamos con otras propuestas tanto académicas como comerciales.

Para finalizar, la tercera parte de esta tesis se ocupa del uso de tecnologías resistivas en la memoria principal. En concreto, proponemos algunas técnicas e ideas para prolongar el ciclo de vida de sistemas de memoria basados en PCM.

Las contribuciones se pueden resumir como sigue:

- El *Filtro de Pila* es el primer esquema de filtrado semántico centrado en consumo. Además, probamos que un pequeño Filtro, de sólo 32 palabras, proporciona buenos resultados, lo que permite poner el filtro debajo de la cache de datos, ahorrando consumo de ésta sin afectar al rendimiento.

- Nuestro algoritmo *Access Before* es la primera propuesta de detección de carreras dinámica desde una perspectiva híbrida hardware/software. Usando extensiones realistas de hardware somos capaces de acelerar la detección de carreras. Nuestra técnica resulta ser más rápida que herramientas comerciales, tanto de la comunidad de código abierto como de la industria.

- La tecnología PCM no está suficientemente madura para ser incorporada como tecnología de memoria principal aún, debido a su escasa durabilidad. Nuestra técnica basada en un algoritmo de compresión consigue prolongar la vida de sistemas basados en memoria PCM por encima de lo que se logra con otras técnicas.

B.1.3. Organización de la tesis

El resto de la tesis se organiza como sigue:

- El capítulo B.2 presenta el Filtro de Pila, nuestra propuesta para reducir el consumo de la cache de datos de primer nivel en el contexto de sistemas empotrados.

- En el capítulo B.3 discutimos el problema de la detección de carreras de datos y presentamos nuestra propuesta híbrida HW/SW para acelerar la detección de carreras de datos.

- En el capítulo B.4 realizamos un estudio a fondo del potencial de compresión de datos como medio de supervivencia a fallos de memoria y presentamos una técnica para extender la durabilidad de memorias PCM.
- El capítulo B.5 recoge las conclusiones y el trabajo futuro.

B.2. Eficiencia energética: Reducción del consumo de DL1 utilizando un Filtro de Pila

La cache de datos de primer nivel (DL1) es una de las estructuras más frecuentemente accedidas en el procesador, lo que junto a su tamaño la convierten en uno de los principales consumidores de potencia. Para reducir su consumo, en este capítulo proponemos una pequeña estructura de filtrado que explora las características especiales de los accesos a pila. Este filtro, que actúa como un nivel superior, no-inclusivo de la jerarquía de memoria de datos, consta de un conjunto de registros que mantienen los datos almacenados en la vecindad de la cima de la pila. Nuestras simulaciones muestran que usando un pequeño *Filtro a Pila* (*Stack Filter, SF*) de unos pocos registros, se puede ahorrar entre el 10 % y el 25 % de consumo de cache en media, con una variación del rendimiento despreciable.

B.2.1. Introducción

Las mejoras técnicas continuas en el campo de los microprocesadores hacen que la tendencia vaya hacia chips cada vez más sofisticados. Sin embargo este hecho trae consigo un incremento en el consumo de energía, y es conocido por todos los arquitectos de computadores que el objetivo principal de los diseños actualmente es conseguir un alto rendimiento y un bajo consumo simultáneamente. Es por esto por lo que muchos investigadores han centrado sus esfuerzos en reducir el consumo. El consumo se reparte entre distintas estructuras, incluyendo las caches, el banco de registros, el predictor de saltos, etc. Sin embargo las caches *on-chip* pueden acumular una parte significativa del consumo total del chip [4, 5].

Una alternativa para mitigar este efecto es particionar la cache en varias caches más pequeñas [6, 7, 8] con el ahorro en tiempo de acceso y consumo por acceso que ello conlleva. Otro diseño, conocido como cache de filtrado o *Filter Cache* [9](*FC*), intercambia rendimiento por consumo filtrando reverencias a la cache a través de una cache L1 inusualmente pequeña. Una cache de segundo nivel (L2) de características similares a la típica L1 se sitúa en el siguiente nivel de la FC para minimizar la pérdida de rendimiento. Otra alternativa diferente llamada cache de vías seleccionables (*Selective Cache Ways*) [10] permite desactivar un subconjunto de las vías de una cache asociativa por conjuntos durante periodos de baja actividad en la cache, mientras que la cache puede hacerse operativa al completo para periodos de uso intenso de la cache. Las caches de bucles (*Loop Caches*) [11] son otra propuesta para ahorrar consumo mediante una estructura de datos de correspondencia directa

y un controlador de cache de bucles. El controlador sabe de manera precisa, con antelación, si la próxima instrucción que solicite datos los encontrará en la cache de bucle o no. Por ello no hay pérdida de rendimiento. Otras soluciones distintas aprovechan el comportamiento particular de algunas referencias a memoria y proponen reemplazar la cache unificada convencional por varias caches especializadas. Cada una se encarga de distintos tipos de referencias a memoria según sus particulares características de localidad – [12, 13] son ejemplos de esta idea, ambos utilizando la localidad dada en los accesos a la pila –. Estas alternativas muestran la posibilidad de mejorar en términos de rendimiento los esquemas de almacenamiento cache. Es importante resaltar que todas estas técnicas son sólo unas pocas de todas las propuestas existentes en el campo del diseño de memorias cache.

En esta primera parte proponemos una aproximación distinta que también saca provecho de las características especiales de las referencias a la pila. La novedad reside en el hecho de que no empleamos una cache especializada para servir estos accesos; en vez de eso, usamos una estructura simple y de reducido tamaño que almacena unas pocas palabras en la vecindad del puntero a pila, y actúa como un filtro: Si el dato referenciado está dentro del rango guardado en este filtro, evitamos un acceso innecesario a DL1. En otro caso, realizamos el acceso como en un diseño convencional. De este modo, aunque las IPC no experimentan variación, somos capaces de reducir significativamente el consumo de la cache de datos, crítico para el sistema, con una cantidad despreciable de hardware. Aunque esta técnica está diseñada para un procesador de alto rendimiento con arquitectura ARM ([14]), que es el que usamos para la evaluación, es también aplicable a otros a otras arquitecturas (ej. x86 IA-32) y otros ámbitos de aplicación.

B.2.2. Comportamiento de los accesos a pila

Los programas habitualmente usan regiones privadas de memoria a forma de pila para almacenar datos temporalmente con diversos propósitos. Nuestra arquitectura objetivo tiene registros que se preservan tras llamadas a función. Después de que la llamada a una función termine, estos registros deben tener el mismo valor que cuando se realizó la llamada. Para conseguir esto, una llamada a función almacena en la cima de la pila (apila) aquellos registros a preservar que se modifican durante la ejecución de la función. Una vez la función termina, los registros son restaurados desde la pila (desapilados), y el procesamiento continua desde el punto en el que esta se invocó.

Además de estos registros preservados, cuando el compilador no consigue alojar todas las variables locales en registros, tiene que volcar algunas de ellas en memoria para hacer sitio a otras. Este volcado consiste en almacenar estas variables en la pila mientras no se usen, haciendo sitio para que otras variables puedan ser alojadas en registros. Más tarde, cuando las variables se necesiten de nuevo, su valor se cargará desde la pila a registros desplazando a otras llegado el caso.

Estas operaciones convierten a la pila en una estructura con una localidad temporal y espacial elevada, que potencialmente puede ser aprovechada para gestionar estos

accesos a memoria de manera más eficiente.

Desde un punto de vista conceptual, la pila es una estructura esencialmente software pero que hace uso de ciertos elementos hardware para su gestión. En muchas arquitecturas existe un registro particular para almacenar la dirección de la cima de la pila, o de la primera posición libre sobre ésta. Este registro se conoce comúnmente como puntero a pila (*sp*) e identifica la dirección virtual más pequeña[1] – o grande dependiendo del convenio empleado – que contiene datos válidos de la pila. Aparte del *sp*, en muchas arquitecturas la pila también puede ser accedida mediante el puntero al marco de activación (fp), o cualquier otro registro de propósito general, lo que flexibiliza su gestión/uso pero dificulta la identificación de este tipo de accesos.

Una de las primeras cuestiones que hemos abordado en este trabajo ha sido estudiar las características intrínsecas de los accesos a pila en la arquitectura de referencia (basada en ARM) para los conjuntos de programas de prueba MiBench y MediaBench. Los experimentos llevados a cabo (ver sección 2.2) muestran que el 34.16 % de las instrucciones ejecutadas son accesos a memoria. Además, las referencias a pila acumulan el 15.58 % del total de las referencias a memoria sumadas para todas las aplicaciones y en algunos programas incluso superan el 70 %. Por otra parte, en el sistema procesador-compilador estudiado, el direccionamiento relativo al *sp* es el método dominante de acceso a la pila, acumulando el 60.23 % de todos los accesos. El otro método más comúnmente usado es $r_i + inmediato$ donde r_i es un registro de propósito general. Del total de aplicaciones usadas, sólo en cuatro de ellas el uso de *sp* como registro base es menor del 60 %, y son 5 en las que esta cantidad supera el 75 % de las referencias.

Una característica común a la mayoría de las aplicaciones que no conviene pasar por alto es que las direcciones referenciadas pertenecen a un espacio de memoria pequeño, contiguo y localmente estable. Para CRC32, por ejemplo,durante una porción representativa de la fase de ejecución, el *sp* varía únicamente en un espacio de 33 palabras. En las fases inicial y final del programa este rango aumenta, pero la duración de estas fases es despreciable en comparación con la fase de ejecución. Si estudiamos los valores del *sp* durante la ejecución podemos observar que la desviación típica es menor de 50 palabras para todas las aplicaciones salvo *cjpeg* y *stringsearch*. Estas características nos han llevado a la conclusión de que el contenido de la pila puede ser almacenado eficientemente en una estructura más simple y con menor consumo.

Además, tal y como señalan en [15], el puntero a pila tiene otra característica importante: la localidad de accesos en la vecindad de de la cima de pila (TOS). Aunque el valor del *sp* se modifica dos veces cada vez que el programa llama a una función durante la ejecución (crecimiento y contracción), un cambio neutraliza al otro, y por tanto la cima mantiene el valor inicial después de ambos. Nuestro estudio muestra que la mayoría de los accesos se sitúan en la proximidad del puntero a pila. Este comportamiento sugiere que una estructura pequeña que mantenga unas cuantas palabras en la vecindad de la cima es suficiente para servir la mayoría de los accesos.

[1] En los programas compilados con GNU-gcc para ARM la pila crece hacia direcciones inferiores.

B.2.3. Diseño del Filtro de Pila

Otros autores han detectado también la especial localidad de los accesos a la pila, y han tratado de explotarla mediante el uso de caches. Nuestra propuesta [14, 16] difiere de el trabajo previo en tres aspectos:

1. Estamos implementando un filtro realmente reducido, entre 8 y 32 palabras.
2. Nuestra propuesta coloca el filtro antes, en términos de la jerarquía, de la cache DL1. Estoy tiene que hacerse bajo una restricción: los accesos a DL1 no pueden ser retardados. Por lo tanto, debemos evitar cualquier penalización extra en los fallos. Para tratar esta restricción, incluimos una nueva unidad funcional en la ruta de datos llamada *NeCK*, encargada de llevar a cabo la discriminación entre aciertos y fallos de manera temprana, y así evitar penalizaciones.
3. Nuestra propuesta no se especializa en direccionamiento basado en *sp*.

Para el diseño del filtro de pila (SF) usamos un pequeño banco de registros para mantener N palabras en la vecindad de la cima de pila. Para evitar operaciones de memoria innecesarias, los datos se llevan al filtro bajo demanda, y son copiados al siguiente nivel (cache DL1) sólo si han sido modificados. El filtro tiene dos bits de estado por registro para reflejar la validez y el estado, *modificado* o *no modificado*. Tal y como sucede en las memorias cache, estos bits son activados si el registro contiene datos válidos, y si el dato ha sido modificado respectivamente, y están desactivados en otro caso. Las posiciones de memoria son asignadas a los registros de acuerdo con su distancia a la cima de pila. Cambios en el *sp* conllevan cambios en la distancia entre datos y cima. Para mantener la correspondencia entre la pila y el filtro, cuando el *sp* sufre una modificación de p palabras, los registros deben ser movidos p posiciones para compensar la variación de distancia en la pila. Hemos implementado estos movimientos mediante el uso de los registros como una estructura circular: El banco de registros tiene un puntero a la base, r_{base}, que apunta al registro que se corresponde con la cima. De esta manera, cuando el *sp* sufre una modificación de p palabras, las primeras p palabras del filtro son expulsadas, y copiadas al siguiente nivel si estaban modificadas, y después se actualiza el valor de r_{base}. Las p palabras se toman de la parte superior del filtro si la modificación es una contracción ($p < 0$), o de la parte inferior del filtro si es una expansión ($p > 0$).

Cuando hay una operación de memoria, el procesador debe comprobar si la dirección está cerca de la cima de la pila. Para ello, mientras que la ALU calcula la dirección efectiva, en paralelo, una unidad de propósito específico, la unidad NeCK, calcula si la palabra referenciada pertenece a la N-vecindad de la cima – definimos la N-vecindad de la cima como las posiciones que distan menos de N palabras de la cima–. El bit de validez del registro correspondiente también se comprueba en la misma fase de ejecución. Si ambas condiciones se satisfacen, en la siguiente etapa de la ejecución la palabra es tomada del filtro, en caso contrario la palabra es traída desde la cache, y, si pertenece a la N-vecindad, es copiada al SF. La mayor ventaja

de hacer el cálculo de la distancia y la comprobación de validez en la misma etapa que el cálculo de la dirección efectiva es que los fallos en el acceso al filtro no acarrean penalización de rendimiento (gracias a la detección temprana).

NeCK: Unidad de comprobación de vecindad

Esta unidad especial es responsable de decidir si la instrucción debe acceder al filtro o a la cache de datos. Hemos diseñado esta unidad para tener baja latencia, en el orden de un sumador completo de 32 bits, para evitar alargar el camino crítico. A continuación describimos el funcionamiento (para más detalle consultar las figuras de la sección 2.3):

- **Cálculo de la distancia (*dist*):** Para realizar esta operación hemos añadido en la ruta de datos la unidad NeCK. Esta unidad calcula la distancia (*dist*) entre *sp* y $dirEfectiva$ y comprueba si el dato cae en la N-vecindad de la cima mientras que la ruta de datos calcula la dirección efectiva. Esto se puede llevar a cabo en paralelo ya que el cálculo de $dirEfectiva$ se hace dos veces, una en la ruta de datos y otra en en la unidad NeCK, por lo que éste último no depende de la ruta de datos. Para pertenecer a la N-vecindad, *dist* ha de satisfacer $0 \leq dist < N$. Dado que N es una potencia de dos, todos los bits de *dist* salvo, a lo sumo, los $\log_2 N$ menos significativos deben ser 0. Para ello, un CSA de 32 bits de anchura calcula $[a, b] = \$r_i + inmediato - sp$ donde a y b satisfacen $dist = a + b$.

  A partir de ese punto, para que el bit j-ésimo de *dist* sea 1, $dist_j = a_j + b_j + c_j = 1$ debe ocurrir: (a) existe acarreo del bit previo y o bien $a_j = b_j = 0$ o bien $a_j = b_j = 1$, (b) no hay acarreo del bit previo y $a_j = \bar{b_j}$. Tanto en (a) como en (b) la palabra estaría fuera de la N-vecindad de la cima. Esta condición se puede relajar de la siguiente manera. Para c_j, en vez del acarreo de entrada el bit j-ésimo, podemos considerar $c_i = a_{i-1} \, OR \, b_{i-1}$ ya que si $a_{i-1} \, OR \, b_{i-1} = 1$ y $c_{out} = 0 \Rightarrow a_{i-1} \, XOR \, b_{i-1} \, XOR \, c_{i-1} = 1$ y así $ofs_{i-1} = 1$ por lo que la comprobación de (b) para el bit $i - 1$ fallará.

- **Detección de presencia:** Una vez que *dist* es conocido se ha de comprobar la presencia del dato en el filtro. Esto se hace mirando el bit de validez que controla el estado de los registros del filtro. Si la operación es un almacenamiento, el bit de validez puede ser ignorado, ya que la operación modificará todo el registro.

Modificaciones en la ruta de datos

La inclusión de nuestro filtro requiere ligeras modificaciones en la ruta de datos. Primero de todo, tenemos que añadir la unidad NeCK y el propio filtro. Dado que las modificaciones de *sp* necesitan actualizar el filtro, tenemos que añadir hardware para pausar el lanzamiento de instrucciones e inyectar las operaciones necesarias para gestionar la actualización de *sp*. Para terminar necesitamos añadir un camino para comunicar la ruta de datos de ejecución con el filtro y la DL1.

Gestión del Filtro de Pila

Una vez presentadas las extensiones y modificaciones del hardware necesarias, describimos brevemente cómo gestionar el *Filtro de Pila*, es decir, cómo realizar sus consultas y actualizaciones.

- **Consulta:** Inicialmente la unidad NeCK comprueba si la referencia es a la N-vecindad. De ser así, en la siguiente etapa, en el caso de instrucción de carga, si el bit de validez está activo se envía la palabra accedida desde el filtro. Si no está activo entonces la palabra se carga desde la cache de datos y se copia en el filtro, activando el bit de validez. En el caso de una instrucción de almacenamiento, si el bit de validez está activo o la operación es sobre una palabra alineada, se escribe la palabra en el filtro y se activan los bits de validez y modificación. Si la palabra correspondiente en el filtro no es válida y va a ser escrita parcialmente, primero se recupera el dato de la memoria cache y después realiza el almacenamiento en el filtro; tanto el bit de validez como el de modificación se activan a continuación.
- **Actualización de *sp*:** Cuando *sp* es modificado, el hardware tiene que rotar el filtro. Esta operación se hace en dos pasos:
  1. **Detección de la modificación:** Cuando una operación que va a modificar el valor de *sp* es lanzada, el valor de *sp* es copiado y se pausa el lanzamiento de las instrucciones subsiguientes. Una vez el nuevo valor de *sp* ha sido calculado, el filtro calcula la cantidad de palabras en las que es preciso rotar el filtro, y realiza esta rotación.
  2. **Rotación del Filtro de Pila:** El filtro calcula cuales de los registros serán desalojados e inyecta operaciones de almacenamiento de acuerdo con el valor de los bits de validez y modificación de las palabras desalojadas. Después desactiva los bits de validez y modificación de los registros desalojados. Para terminar, el registro base se actualiza inyectando una instrucción aritmética especial. Después de que las operaciones de almacenamiento hayan sido lanzadas y de que el registro base haya sido modificado el lanzamiento de instrucciones se reanuda. Nuestro modelo del filtro en el simulador tiene en cuenta estas pausas e inyección de instrucciones, y son tenidas en cuenta en los cálculos de consumo y rendimiento.

Otros aspectos a tener en cuenta

La arquitectura objetivo (ARM) tiene diferentes registros *sp* y pilas para los distintos modos de ejecución lo que en cierto modo simplifica la gestión del filtro en caso de excepción o llamada al sistema operativo. Los modos *User* y *System* comparten *sp* y pila y en ambos el filtro está activo. Cuando una excepción cambia a uno de los otros cinco modos (*IRQ*,*FIQ*,*Supervisor*,*Undefined* o *Abort*) el registro que actúa como *sp* cambia y el filtro se deshabilita. Esto no afecta la corrección del mismo,

ya que la pila es diferente. Es más, todas las pilas son disjuntas dos a dos y por lo tanto no cabe posibilidad de que haya una incoherencia entre el filtro y la cache de datos, y por ende no es necesario volcar el filtro a memoria cuando ocurre una excepción o una llamada al sistema.

Cuando hay un cambio de contexto en el procesador, el filtro ha de ser respaldado a memoria como cualquier otra estructura cache, pero para mantener la corrección de los datos este ha de ser respaldado a DL1 antes de que ésta escriba en memoria principal.

El último aspecto a tener en cuenta es la coherencia con el resto de la jerarquía de memoria. A primera vista la introducción de un nuevo nivel no inclusivo en la jerarquía de memoria puede parecer propenso a errores de coherencia. Dado que el filtro actúa únicamente sobre la pila, y que la pila es privada a cada thread, es fácil entender que no es éste el caso, por lo que mientras que los hilos respeten la privacidad de la pila de otros hilos, la pila y el filtro, nunca serán objeto de errores de coherencia. Por lo tanto la coherencia se resuelve de manera *gratuita* gracias a la privacidad de la pila. Pueden encontrarse más detalles acerca de la privacidad de la región de pila en [17].

B.2.4. Evaluación

El propósito de nuestra evaluación es mostrar que usando un Filtro de Pila (FP) de pequeño tamaño podemos reducir razonablemente el consumo, obteniendo un impacto mínimo sobre el rendimiento. En primer lugar, evaluamos el FP en el contexto de una arquitectura ARM de alto rendimiento y analizamos el influencia de su tamaño en los resultados obtenidos. Después comparamos el uso del filtro para dos arquitecturas distintas, a saber, ARM y x86 (IA-32). Para terminar comparamos el FP con la *Cache Filtro* [9] y la *cache basada en regiones* [20] para mostrar que, para tamaños pequeños, éste consigue un mayor ahorro con menor impacto en el rendimiento.

Análisis del Filtro

Hemos llevado a cabo pruebas para tres tamaños distintos de filtro: 8 palabras (32 bytes), 16 palabras (64 bytes) y 32 palabras (128 bytes). Para los tres tamaños la mitad de los registros se corresponden con registros bajo la cima y la otra mitad con registros sobre la cima. Es preciso señalar que las medidas obtenidas han sido promediadas empleando una media geométrica, ya que es menos sensible a los valores extremos.

Los resultados recogidos en la sección 2.6 demuestran que el FP es útil y consigue capturar gran parte la localidad de datos de la pila. Para un filtro de 32 palabras por ejemplo hay aplicaciones para las que la tasa de acierto – teniendo en cuenta sólo los accesos a la pila – es superior al 70 % y en media es 40 %. Estos números se reducen a 60 % y 32 % cuando el filtro tiene solo 16 palabras de longitud. Si

se calcula tasa de aciertos con respecto a todos los accesos a memoria, no sólo los de la pila, observamos que para algunos programas – por ejemplo *adpcm* – la tasa global se reduce drásticamente debido a su baja proporción de accesos a pila. Sin embargo para la mayor parte de programas la tasa de aciertos global supera el 25 % al emplear un filtro de 32 palabras, lo que se traduce en una reducción de 25 % en los accesos a DL1.

Como cabe esperar, esta reducción de accesos a DL1 acarrea una reducción del consumo de dicha estructura. El consumo de DL1 respecto al sistema base de referencia – sin filtro–, se reduce hasta en un 25 % para *cjpeg*, que es el mejor caso. Del mismo modo, el consumo de la DTLB se ve reducido en una proporción análoga, ya que en caso de acierto del filtro no se accede tampoco a esta estructura. Sólo cuando el filtro falla y es preciso acceder a la cache DL1, se requiere la consulta de la DTLB – al tratarse de una cache virtualmente accedida –. Como cabía esperar, para aquellas aplicaciones que hacen un escaso uso de la pila, el ahorro de energía es despreciable.

El FP está concebido para reducir el consumo de energía. Su latencia es la misma que la de un acceso a DL1, por lo que no cabe esperar mejora alguna en el rendimiento. A priori, tampoco cabe esperar un deterioro del mismo ya que la pronta detección de fallos evita ciclos de penalización. No obstante, hay ciertos efectos laterales que afectan al rendimiento y producen modificaciones que van desde la pérdida del 10 % en casos degenerados (*crc*) a una ligera ganancia del 1 % en otros casos. La causa de la pérdida de rendimiento observada en *crc* se debe a la pobre codificación de este benchmark, para el que el número de llamadas a función es excesivamente elevado, lo que genera un elevado número de actualizaciones del fp y por consiguiente de rotaciones del filtro. Es preciso destacar que, si en lugar de la implementación original de este benchmark se emplea una más razonable, la pérdida de rendimiento prácticamente desaparece. Las ligeras mejoras de rendimiento se deben al ligero incremento de la capacidad de almacenamiento *on-chip* producido por la inclusión del filtro, que podríamos considerar como una pseudo-cache DL0 de entre 32 y 128 bytes de almacenamiento no inclusivo. Este espacio extra evita algunos fallos de conflicto/capacidad entre datos de la pila y datos de otras regiones al tiempo que proporciona un camino alternativo de acceso a datos, con sus propios puertos de acceso.

La evaluación del FP se completa con un análisis del consumo global del procesador. El filtro, al tratarse de una pequeña estructura de correspondencia directa disipa mucha menos potencia que una cache, y por lo tanto un acceso al mismo consume mucho menos que un acceso a DL1, lo que permite un ahorro neto en el consumo de energía del conjunto del procesador. Cabe resaltar que con filtros de 16 y 32 palabras, incluso en el peor caso, el consumo siempre se ve reducido respecto al sistema base de referencia – sin filtro –. Como es fácil comprender el ahorro en el conjunto del procesador depende de la proporción consumida por la DL1 y la DTLB y del ahorro que el filtro origine en estas estructuras. A modo de ejemplo, un FP de 32 palabras consigue reducir casi en un 10 % el consumo en *cjpeg*, más del 6 % en *crc*, 5 % en *djpeg* y entre el 1.5 % y 4 % para *bitcnts*, *djpeg2k*, *dmpeg2*, *sha* y *tiffdither*. Para otros tamaños de FP el ahorro es menor.

Influencia de la arquitectura

Esta parte del estudio experimental se centra en la influencia de la arquitectura en el uso y beneficio de un FP. Las arquitecturas consideradas son x86 (IA-32) y ARM. Para ambas arquitecturas la configuración del FP es la misma: 32 palabras, 16 sobre la cima, 16 bajo la cima.

Los resultados demuestran que la arquitectura x86 hace un mayor uso del filtro. Esto se debe a dos factores. El primero es que una máquina x86 tiene menos registros arquitectónicos que una máquina ARM, por lo que el proceso de asignación de registros del compilador tiene mayores dificultades en ubicar las variables en registros, y éstas han de ser almacenadas temporalmente en la pila. El segundo factor es el convenio de llamadas a función empleado en cada arquitectura. Debido al reducido número de registros arquitectónicos de x86, los diferentes convenios de llamadas suelen pasar los argumentos por la pila y sólo el valor de retorno, o un puntero a él se pasa por registro. Esto se transforma, de nuevo, en un mayor número de accesos a pila, que potencialmente pueden dar lugar a aciertos de FP, como lo demuestran los resultados experimentales: 8 de las 11 aplicaciones usadas para la evaluación tienen una tasa de acierto global superior al 50 % para una máquina x86. Esto nos lleva a afirmar que para procesadores de bajo consumo como el Intel Atom el beneficio de un FP puede ser incluso mayor que en el caso de una arquitectura ARM.

Comparativa con otras soluciones

La ultima parte de la evaluación experimental compara nuestra propuesta con la *cache basada en regiones* de Lee y Tyson[20], y la *cache de filtro* de Kin *et al.* [9]. En concreto comparamos un FP de 32 palabras, con una cache filtro de 32 palabras y con una cache basada en regiones de varios tamaños, empleando para todos ellos el mismo sistema de referencia.

Para un tamaño tan reducido, observamos que la cache filtro es incapaz de aprovechar la elevada localidad temporal, ya que al no discriminar por tipo de acceso genera un elevado número de fallos de conflicto que no son amortizados por el menor coste de los accesos a esta estructura. En el caso de la cache basada en regiones, el principal problema es que, al no disponer el sistema base de referencia de una cache de segundo nivel, los fallos en la cache de pila se sirven directamente desde memoria principal lo que perjudica en exceso el rendimiento y por ende el consumo.

De las 3 propuestas analizadas, la que ofrece un mejor compromiso entre rendimiento y consumo de energía, cuantificado mediante el EDP – producto del ratio de ahorro energético por el ratio de pérdida de rendimiento – es el FP, para el que este producto nunca es mayor que 1 a diferencia de la cache de filtro y la cache basada en regiones para las que se supera este valor en la mayoría de los programas analizados.

Conclusiones

Todos los programas tienen una pila software donde almacenar, entre otros, el estado de la máquina entre llamadas a función. Las características de esta pila incluyen una elevada localidad tanto espacial como temporal. Es más, cuanto más cercana a la cima es una palabra, más localidad muestra. Si bien las caches explotan esta localidad de manera natural, existe margen de mejora aprovechando la extraordinaria localidad de estos accesos a memoria. Varias propuestas previas han tratado de sacar partido para incrementar el rendimiento, mientras que nosotros nos hemos centrado en reducir el consumo de energía.

A lo largo de este capítulo hemos mostrado que un pequeño espacio de almacenamiento basado en registros es suficiente para capturar gran parte de las referencias a la pila, reduciendo significativamente la cantidad de accesos a DL1. Esta reducción implica menor consumo en la cache de datos. Además el filtro está basado en registros y es más pequeño, por lo que la energía por acceso es mucho menor. Por lo que respecta al rendimiento, nuestro filtro tiene la misma latencia que la DL1, y los fallos no conllevan ninguna penalización, por lo que apenas hay variaciones de rendimiento, como muestra la evaluación experimental. Por último, la evaluación de esta técnica tanto en ARM y x86 confirma que nuestra solución responde satisfactoriamente en varias arquitecturas.

B.3. Soporte a la depuración: Detección de Carreras de Datos con Soporte HW Minimal

El desarrollo de software concurrente es mucho más difícil que el desarrollo de código secuencial. Los programadores tienen que afrontar interacciones impredecibles entre los hilos debido a la falta de determinismo del sistema de memoria. Sin una disciplina exhaustiva de operaciones de sincronización, el intercalado de accesos a memoria queda fuera del control del programa, lo que puede dar lugar a errores los cuales las *carreras de datos* son los más comunes. Una carrera de datos ocurre entre dos operaciones de memoria de hilos distintos, siendo al menos una de ellas una escritura, si acceden a la misma posición de memoria sin estar ordenadas por una operación de sincronización. Este acceso no *ordenado* a datos compartidos puede producir fallos del sistema, o corrupción de datos silenciosa, por lo que es recomendable evitarlo. Para ello existen esencialmente dos alternativas: que el lenguaje de programación lo impida por construcción o bien hacer uso de herramientas que permita detectar estos accesos no ordenados, siendo este último nuestro caso.

Hasta la fecha se han propuesto una amplia cantidad de mecanismos para detectar y evitar carreras, incluyendo varias soluciones puramente hardware (HW) y puramente software (SW).

Las soluciones hardware son generalmente complejas. Requieren un soporte hardware extenso, como cambios en la jerarquía cache que incluyen añadir una cantidad

significativa de estado a cada bloque, extender los mensajes de coherencia para contener información adicional y modificar el estado del protocolo de coherencia para reaccionar a eventos de interés. Además, los requisitos de almacenamiento, que suele estar cerca de componentes clave del procesador, son prohibitivos. Por otro lado, las implementaciones software pueden ser usadas sin modificar la arquitectura, pero son típicamente demasiado lentas para estar siempre activas. Realizar el análisis de las operaciones de memoria en software es lento. Más aún, estos algoritmos requieren una gran cantidad de metainformación y una frecuente comunicación entre hilos .

En este capítulo proponemos una solución híbrida llamada *Accessed Before*, en la que el soporte hardware se reduce al mínimo, disminuyendo así la complejidad y mejorando el rendimiento respecto a propuestas anteriores. Para ello aumentamos el repertorio con una sencilla instrucción que recibe una dirección como entrada y devuelve el estado de coherencia del marco de bloque que la contiene. También proponemos un algoritmo que usa este soporte HW para detectar carreras de datos de manera eficiente. Nuestra solución saca partido de dos conceptos: (1) la información dinámica necesaria puede ser extraída del estado de coherencia que el hardware se encarga de mantener, (2) existe un tipo de carreras de datos dinámicas que son más fáciles de detectar pero para las cuales se puede demostrar que cubren el conjunto de todas las carreras estáticas, dado que el programa se ejecute un suficiente número de veces. Asimismo mostramos cómo alterar la planificación de la ejecución del programa para propiciar que las carreras estáticas den lugar a este tipo de carreras dinámicas, incrementando la fiabilidad el mecanismo de detección hasta un nivel comparable al algoritmo de referencia tradicional, *Happened-Before*, pero con un requisito espacial mucho menor y menos sobrecarga temporal.

B.3.1. Conocimiento previo

Terminología Nos referimos a las instrucciones usando la siguiente terminología, usual en este ámbito: *instrucción estática* se refiere a la parte del código unívocamente determinada por su dirección de memoria, *instrucción dinámica* es cada una de las instancias de una instrucción estática ejecutada por los distintos hilos. Análogamente, una *carrera estática* está formada por dos instrucciones estáticas que cuando se ejecutan producen una carrera de datos, y el término *carrera dinámica* lo usamos para cada aparición de una carrera en tiempo de ejecución. Cuando nombramos hilos, hablaremos de un *hilo local*, generalmente involucrado en una carrera, y de uno o más *hilos remotos* que interactúan con el hilo local a través de la memoria compartida. Si una carrera tiene lugar llamaremos al hilo local *hilo detector*, e *hilo delincuente* al hilo remoto involucrado. El término *era* describe al conjunto de instrucciones dinámicas que ejecuta un hilo dado entre dos operaciones de sincronización consecutivas. Para simplificar la discusión asumimos una correspondencia uno a uno entre hilos y procesadores con su cache privada asociada. La gestión de los casos en los que esto no se cumple se discute posteriormente en la sección B.3.4.

Nos centramos en hilos del estándar POSIX, *pthreads*, y los mecanismos de sincronización asociados a él, que incluyen creación y unión de hilos, entrada y salida de

secciones críticas (*mutex*), y espera y señalización de variables condicionales. Definimos el concepto de *fuente* de una operación de sincronización al hilo que inicia la operación mediante, por ejemplo, la salida de una sección crítica, o la señalización de una variable condicional. De igual modo, el *destino* de la sincronización es el hilo que termina la operación, entrando de nuevo en la sección, o reanudando la ejecución tras la recepción de la señal. En este contexto, cuando hablamos de dos eventos ordenados – sean estos cuales sean –, nos referimos al orden parcial *happened-before* definido por Lamport [43].

B.3.2. Detección de carreras de datos

Cuando se escribe código concurrente empleando un modelo de memoria compartida, los programadores usan operaciones sincronizaciones para restringir el orden en el que los distintos hilos acceden a los datos compartidos. Una carrera de datos ocurre cuando el programador omite operaciones de sincronización y permite que más de un hilo acceda a las variables compartidas de manera *no sincronizada*, siendo al menos uno de los accesos una escritura.

En lo que se refiere a detección de carreras hay dos filosofías dominantes: algoritmos basados en *happened-before* como [44, 45, 37] y algoritmos basados en disciplina de bloqueo como [41, 38]. La disciplina de bloqueo consiste en la monitorización de la adquisición y liberación de todos los mutex para asegurar, para cada posición de memoria, que al menos un mutex dado ha sido adquirido en todos los accesos a dicha posición de memoria. Nosotros nos centramos en algoritmos de *happened-before*, que se basan en la relación del mismo nombre definida por Lamport para ordenar – parcialmente – los accesos a memoria entre hilos: *hb-orden*. Este orden se establece en función del orden del programa dentro de cada hilo y en función de las operaciones de sincronizaciones entre hilos distintos. Una carrera de datos se produce cuando, dadas dos operaciones de memoria a la misma posición, no existe ninguna relación de hb-orden entre ellas. Esta definición puede extenderse a *eras*, como veremos a continuación, dando lugar a un algoritmo de detección de carreras basado en eras.

Una forma de implementar este algoritmo de detección basado en eras, es mediante la utilización de conjuntos de lecturas y escrituras. Para cada era, el hilo local almacena el conjunto de direcciones leídas y el direcciones de posiciones escritas. Si se cumple que (a) la intersección de cualquiera de estos dos conjuntos con el conjunto de escrituras de una era de un hilo remoto o (b) la intersección del conjunto de escrituras con el conjunto de lecturas de una era de un hilo remoto, y se cumple además que la era local y la era remota no están hb-ordenadas, entonces existe una carrera para cada una de las direcciones de memoria pertenecientes a la intersección de ambos conjuntos. Aunque intuitiva, esta implementación requiere una elevada cantidad de memoria y comunicación entre hilos, lo que le resta atractivo.

Detección basada en Happened-Before

Puesto que la relación *happened-before* tiene en cuenta el orden del programa dentro de cada hilo, su definición puede extenderse fácilmente a eras en lugar de operaciones individuales. La definición formal es la siguiente: (1) Si dos eras, e_1, e_2 provienen del mismo hilo y e_1 se ejecuta antes que e_2 entonces $e_1 \rightarrow e_2$; (2) Si la última operación de una era e_1 inicia una sincronización que tiene su destino en la primera operación de la era e_2, que puede ser ejecutada en otro hilo, entonces $e_1 \rightarrow e_2$; (3) La definición se cierra por transitividad. En esta relación el símbolo $\rightarrow$ se usa en vez de $<_{hb}$. Esta definición de orden entre eras se puede extender al orden entre cada par de operaciones de las mismas.

En la práctica, las eras de un mismo hilo se numeran en orden creciente y cada hilo mantiene un *reloj* vectorial en el que almacena para cada hilo –incluido el mismo – el valor numérico de la última era *observada* por él. Para determinar la relación *happened-before* entre dos eras, por lo tanto, basta con comparar sus relojes vectoriales. El algoritmo de detección que hemos descrito previamente realiza esta comparación antes de calcular la intersección en los conjuntos de lecturas y escrituras, para así evitar cálculos innecesarios.

Coherencia cache

Antes de presentar nuestra propuesta es conveniente realizar un breve recordatorio del mecanismo de coherencia HW sobre el que se construye.

Sin pérdida de generalidad asumimos un protocolo coherencia cache basado en invalidación de tipo MESI, que sin duda alguna es el más frecuente en los procesadores actuales.

Un bloque de cache está siempre en uno de los siguiente estados: M (modificado) si la única copia privada está en la cache local y su contenido difiere de memoria principal, – es responsabilidad de esta cache mandar su copia del bloque si alguna otra cache lo solicita; E (exclusivo) si la única copia privada esta en la cache local y no ha sido modificada; S (compartida) si el bloque está en la cache local sin haber sido modificado y puede que haya copias privadas en otras caches o I (inválido) si los contenidos han de ser solicitados de nuevo porque la copia local está obsoleta.

Antes de realizar la modificación de un bloque, la cache local ha de poner el bloque en estado M. En la mayor parte de los casos, esto se lleva a cabo enviando mensajes de invalidación a todos aquellas cache que contienen una copia válida de éste y esperando a las correspondientes respuestas. La única excepción es si el bloque está en estado E, en cuyo caso, la cache local puede pasar su estado a M sin notificar a otras caches. Antes de que una cache pueda leer el contenido de un bloque tiene que poseer una copia local en un estado distinto de I. Esto se consigue *elevando* una petición del bloque. Cuando el dato se recibe se guarda en la cache local, en estado E si no hay otras copias privadas, o en estado S si hay más caches compartiendo el dato. Consideramos una *degradación* cualquier transición en la dirección $M \rightarrow E, S \rightarrow I$.

Transición hilo local	Acceso hilo local	Acceso hilo remoto	Tipo de carrera
$M \rightarrow S$	escritura	lectura	E→L
$M \rightarrow I$	escritura	escritura	E→E
$M \rightarrow I$ $E \rightarrow I$ $S \rightarrow I$	lectura	escritura	L→E

Cuadro B.1: Tipos de degradaciones y carreras.

B.3.3. Soporte hardware mínimo para la detección de datos

Nuestra propuesta de soporte hardware (mínimo) para acelerar la detección de carreras de datos consiste simplemente en exponer el estado de coherencia al software mediante una nueva instrucción, siendo este el encargado de grabar y usar la información del estado de coherencia para detectar las carreras. Para ello proponemos un nuevo algoritmo de detección al que hemos llamado "*Accessed Before*" (AccB), cuya idea clave es llevar la cuenta del último estado de coherencia observado de cada bloque de cache para detectar si ha sufrido pérdida de privilegios (o *degradación*) dentro de una era. Un bloque que pierde privilegios indica la posible existencia una carrera de datos: una cache remota ha accedido al bloque. Conviene resaltar que toda la información necesaria para este análisis es local al hilo, por lo que no se requiere comunicación entre hilos. No ocurre así en el algoritmo *Happened-Before* (HapB) descrito previamente, donde se requiere bastante comunicación.

Algoritmo Accessed Before, AccB

La idea que subyace de AccB es aprovechar el trafico de coherencia generado implícitamente para evitar comunicaciones extra para la detección de carreras. Recordemos que, cuando un hilo escribe una posición de memoria, todas las caches son notificadas para que invaliden su copia local del bloque. Además, si una posición de memoria está modificada en la cache local y un hilo remoto intenta leer dicha posición, la cache local recibe un mensaje para *degradar* el bloque a estado compartido y enviarlo al solicitante. Es posible como veremos a continuación detectar carreras gracias a estas transiciones de estado de coherencia.

El algoritmo es relativamente sencillo y trabaja por eras. Al inicio de una nueva era – tras una operación de sincronización – la primera vez que se accede localmente a cada variable, se inserta una línea en una tabla local que contiene la dirección de la variable y el estado de coherencia. Los subsiguientes accesos a la variable consultan el estado de coherencia en la cache, y lo comparan con el estado guardado en la tabla. En caso de degradación, se ha detectado una carrera y se informa. Al final de la era, para todas las variables de la tabla se compara su estado con el actual en la cache buscando degradaciones que delaten posibles carreras de datos, en cuyo caso también se informa. La Tabla B.1 recoge los distintos tipos de carreras inferidos según la transición producida en el estado de coherencia durante la era.

Evento de interés	Acción del algoritmo
Comienzo de la era	Vaciar la tabla local.
Antes del acceso a memoria	Comprobar el estado actual del bloque correspondiente con la entrada en la tabla local (si existe) para detectar degradaciones.
Después del acceso a memoria	Grabar el estado del bloque correspondiente en la tabla local.
Final de era	Comparar el estado de cada entrada en la tabla con el estado del correspondiente bloque en cache para detectar degradaciones.

Cuadro B.2: Eventos de interés y acciones asociadas.

El funcionamiento del algoritmo está resumido en la Tabla B.2 que muestra las acciones que lleva a cabo el software en respuesta a los principales eventos de interés. De nuevo, resaltamos que todas las acciones tomadas son locales a cada hilo. La única comunicación entre hilos ocurre a través del protocolo de coherencia cache, que está presente independientemente de nuestra técnica. Además, la información contenida en la tabla local hace referencia únicamente a una era, ya que no tomamos en consideración degradaciones que ocurren más allá del final de era. Éstas son dos importantes ventajas de nuestra solución en comparación con HapB, que requiere más comunicación y espacio de almacenamiento.

AccB aprovecha la comunicación implícita que lleva acabo el protocolo de coherencia cache para evitar introducir comunicación extra entre hilos. La idea es que si una variable es compartida por un conjunto de caches, los hilos asociados van a producir tráfico de coherencia entre sus respectivas caches. Es más, podemos definir un orden entre los cuatro estados de MESI: $M > E = S > I$. Este orden cuantifica el nivel de posesión de un hilo sobre el bloque. Cuando un hilo posee un bloque y lo comparte, el nivel de posesión se *degrada*: si un hilo tiene un bloque en estado M y en su cache y un hilo remoto lee el bloque o, peor aún, lo escribe, el estado se degrada a S en el primer caso, e I en el segundo. Es fácil inferir que que las degradaciones de estado están relacionadas con compartición y por tanto solo pueden ocurrir de manera segura, *i.e.* sin derivar en carreras de datos, si la operación que puso el bloque en el estado inicial y la operación remota que produjo la degradación están ordenadas por operaciones de sincronización.

La detección de carreras se reduce por lo tanto a la detección de degradaciones dentro de una era. Para llevar a cabo esta tarea necesitamos un poco de memoria (local) para guardar el estado en el que el bloque quedó después del último acceso (local). Más tarde hay que comparar este estado con su estado en la cache. Esta comparación se realiza sólo antes de cada acceso a la misma variable y una vez más al final de la era. Nótese que sólo hace falta monitorizar las variables compartidas, ya que las variables locales se alojan en la pila, que es privada a cada hilo. Además, una vez la era termina no hay motivo para mantener el estado previo de las variables accedidas en ella. Por este motivo, sólo mantenemos información sobre la era actual.

Asumimos que en el caso de producirse una carrera, el hilo local, i siempre accede

a la variable antes que el hilo delincuente, j. Podemos hacer esto sin pérdida de la generalidad, ya que como todos los hilos monitorizan sus variables, podemos considerar j como hilo local e i como hilo delincuente, en cuyo caso j sería quien detecta la carrera.

Fuentes de pérdida de precisión

Nuestro algoritmo presenta tres limitaciones que reducen su precisión: (1) falsa compartición, (2) desalojo de bloques de cache y (3) terminación anticipada de las eras. A continuación presentamos cada una de ellas indicando en cada caso la solución adoptada para hacerles frente.

Falsa compartición: Queremos reducir el soporte hardware al mínimo, por lo que no contemplamos reducir la granularidad de la información de coherencia más allá de la que aporta el propio protocolo de coherencia. La falsa compartición puede por lo tanto producir falsos positivos. En cualquier otro algoritmo de detección que use la misma granularidad se experimentará el mismo problema. Es más, nuestra solución puede extenderse a una granularidad más fina si fuese necesario, con un coste extra, lo cual es ortogonal a cualquier técnica software empleada para reducir la falsa compartición.

Desalojo de bloques de cache: Cuando un bloque es desalojado de la cache su estado de coherencia asociado se pierde. Como resultado, la cache que desaloja el bloque pierde la capacidad de detectar degradaciones, por lo que puede pasar por alto carreras de datos. De nuevo hemos optado por la solución más sencilla, y asumimos esta pérdida de información.

Terminación anticipada de eras (EEE): Una vez una era termina, la monitorización de variables termina también. Por lo tanto si posteriormente se producen degradaciones de un cierto bloque, no serán detectadas y el algoritmo fallará en la detección de la carrera (falso negativo). Al tratarse de una circunstancia más frecuente que las anteriores, hemos optado por incorporar ciertas mejoras a la implementación inicial, que veremos a continuación, para tratar estos casos.

B.3.4. Implementación

Capa Hardware

Nuestra propuesta consiste en extender el repertorio de instrucciones con una instrucción StateCheck (`StChk off(basereg),reg`) que devuelve el estado del bloque de cache de `off(basereg)`'s a través del registro `reg`. Si el bloque no está presente, la instrucción devuelve un valor especial *NoPresente* (NP) para distinguirlo del caso en el que esté presente en estado inválido. En caso de encontrarse en una transición, se devuelve el último estado válido.

Para implementar StateCheck tenemos que hacer modificaciones menores a (1) la lógica de control, (2) la ruta de datos de la cache, y (3) los controladores de cache.

En concreto, modificamos la lógica de control para que reconozca y decodifique la instrucción StateCheck y comunique a la cache que la información requerida no es el dato asociado a `off(basereg)`, si no el estado de coherencia. Modificamos la ruta de datos con un multiplexor que cree un camino para que el estado de coherencia pueda copiarse al banco de registros. Los controladores de cache por su parte requieren dos modificaciones: (a) si el bloque solicitado no está en la cache, se debe devolver el valor NP sin generar un fallo de cache y (b), cuando recibe una consulta del estado de un bloque el controlador de cache debe activar el bit de selección del multiplexor para permitir que el estado se transporte al procesador.

Capa Software

Estructuras de datos: Asociamos una tabla local a cada hilo para guardar la información de los accesos a variables compartidas realizados en cada era. Estas tablas son de tipo *hash*, indexadas mediante la dirección de la variable y son almacenadas en memoria principal. Cada entrada contiene el estado de coherencia guardado y la dirección de la última instrucción que accedió a dicha variable localmente.

Puntos de instrumentación: Usamos re-escritura dinámica de binarios para instrumentar cada operación de sincronización y cada operación de memoria sobre datos compartidos. Las sincronizaciones delimitan las eras. Insertamos un código que recorre la tabla local en busca de degradaciones antes de cada sincronización, seguido de un código que borra el contenido de la tabla en preparación para la nueva era.

El código de instrumentación de los accesos a memoria comprueba el estado actual de coherencia usando la instrucción StateCheck, y lo compara con el estado almacenado en la tabla. Si detecta una degradación, informa de la carrera, indicando la dirección de memoria implicada y la dirección de la instrucción que realizó el último acceso. Para terminar, actualiza el estado. Esta actualización se hace en base al estado actual y la operación llevada a cabo, lo cual es más seguro que ejecutar un segundo StateCheck para leer el estado, ya que debido al entrelazado de operaciones de memoria, una degradación puede tener lugar entre la primera instancia de StateCheck y la segunda, lo que produciría un fallo en la detección.

Optimizaciones

Hemos añadido tres optimizaciones, dos de las cuales mitigan el problema de la terminación precipitada de eras, y otra permite reducir la sobrecarga debida a la instrumentación.

Mejora de la cobertura: Redefinición de los límites de las eras. Para reducir el problema de la terminación anticipada de eras, cambiamos ligeramente la definición de los límites de una era para AccB y terminamos una era únicamente cuando el hilo local es la **fuente** de la sincronización, y no cuando es el destino. La idea es que cuando un hilo es fuente de una sincronización está comunicando al

resto de hilos que es seguro acceder a las variables compartidas que ha modificado en la era que termina. En cambio cuando es destino, el hilo está siendo notificado de que es seguro acceder a las variables de otro hilo pero puede que aún no sea seguro acceder a las suyas.

Mejora de la cobertura: Perturbación de la planificación. Para hacer frente al hecho de que AccB sólo pueda detectar carreras entre eras cuya ejecución se solapa en el tiempo, aplicamos perturbaciones estocásticas a la ejecución de los hilos para favorecer las variaciones en los solapamientos de eras. Para ello, cada vez que un hilo comienza una era decide al azar unirse o no a una barrera especial. Una vez que todos los hilos han decidido unirse – o no unirse –, o cuando la barrera caduca, los hilos bloqueados proceden a ejecutar su era. Al final de dicha era, los hilos implicados en la primera barrera se unen a una segunda. Una vez los hilos involucrados llegan a esta barrera o la barrera caduca, se hace la comprobación de degradaciones para todas las variables locales a la era de cada hilo, y a continuación prosiguen la ejecución. Hemos bautizado a esta técnica con el nombre *Barreras probabilísticas*.

Al contrario que CHESS [46] que sistemáticamente explora todas las intercalaciones de secciones críticas y lleva a cabo una detección de carreras en cada posible intercalación para conseguir una cobertura completa, AccB altera la ejecución extendiendo al azar la duración de algunas eras de manera que se solapen completamente, aliviando el problema de la terminación anticipada (EEE), un problema específico de AccB.

Reducción del impacto en el rendimiento con soporte hardware extra. Hasta el momento hemos mostrado como funciona AccB con un soporte HW mínimo, esto es, nada más exponiendo el estado de coherencia al software. Pero podemos llevarlo más allá añadiendo unos pocos bits de estado a cada bloque y usarlos para reducir la cantidad de accesos a la tabla local.

Además del estado de coherencia usual, almacenamos dos bits extra por bloque, el bit de lectura local (*lrd*) y el bit de escritura local (*lwr*), y un bit global a la cache, el bit de degradación (*dgd*). Estos bits se usan para guardar la naturaleza de los accesos al bloque desde el inicio de la era, y para registrar posibles degradaciones ocurridas en la cache desde el inicio de la era. También necesitamos una instrucción que ponga a cero estos bits para todas las líneas de la cache al inicio de una era. Para que estos bits tengan la semántica requerida tenemos que modificar el controlador de cache para activar los bits de *lrd* y *lwr* cuando proceda y para activa el bit de *dgd* cuando ocurra una degradación debido a un acceso remoto a una línea que tenga alguno de los bits *lrd* o *lwr* activos; y tenemos que extender la instrucción StateCheck para que devuelva el valor de estos bits junto con el estado de coherencia.

La capa software usa estos bits para evitar consultas y actualizaciones innecesarias de la tabla local, acelerando dramáticamente el esquema de detección, ya que si una variable ya ha experimentado un tipo de acceso en una era, subsiguientes accesos de la misma naturaleza no necesitan escribir de nuevo en la tabla. Asimismo, si el bit de *dgd* es cero, no tiene sentido consultar el estado guardado en la tabla, ya que sabemos que desde el inicio de la era no ha habido ninguna degradación.

A esta versión modificada la llamamos AccB ++.

Otros aspectos a tener en cuenta

A continuación proponemos soluciones para tratar situaciones como la migración de hilos, la ejecución especulativa y la compartición de memorias cache. La manera de resolver una migración es, o bien volcar las caches cuando hay una migración de hilos, lo cual supone una pérdida de información pero no introduce falsos positivos, o bien usar al inicio de cada era y en el momento de detección de una carrera, la instrucción CPUID. Si el resultado devuelto no es el mismo en ambos instantes, es que ha habido una migración desde el principio de la era y la información usada para detectar la carrera no es fiable, por lo que no debe informarse de esta. Para que la ejecución especulativa no sea un problema existen mecanismos como CHERRY [47], que permiten la ejecución especulativa de lecturas de memoria una vez la instrucción ha pasado el punto de no retorno. El problema de las caches compartidas se mitiga usando, para cada bloque, bits de lrd/lwr distintos para cada hilo que lo comparte. De este modo, el controlador de cache es capaz, tanto de detectar carreras que se producen entre los hilos que comparten cache, como de filtrar degradaciones que son seguras para los hilos involucrados. Es preciso resaltar a este respecto que una jerarquía multinivel no supone ningún problema. Basta con colocar el algoritmo en el nivel para el cual se mantiene la información de coherencia que es, en general, el nivel más cercano a memoria principal que no es común a todos los procesadores.

B.3.5. Evaluación

Para demostrar los beneficios de AccB, presentamos una comparativa resumida con HapB en términos de rendimiento, sobrecarga en espacio y precisión (ver sección 3.7 para más detalle).

AccB mejora el rendimiento en un 21 % en media, y AccB ++ siempre mejora el rendimiento e incluso llega a conseguir una aceleración de 6x en el mejor caso. Estas ganancias se deben a que, desde un punto de vista abstracto, AccB y AccB ++ ganan rendimiento porque reducen enormemente la cantidad de escrituras de la historia necesaria para el análisis. En el caso de AccB ++ es obvio que el soporte hardware adicional permite una reducción de las consultas de la tabla y de ahí su elevada mejora de rendimiento.

Por lo que respecta a la sobrecarga de almacenamiento, AccB utiliza, en media, sólo el 0.8 % de la memoria extra que requiere HapB. Si analizamos la cantidad de datos que requieren, promediando entre todas las aplicaciones, aunque AccB tenga más entradas por era que HapB debido, entre otros factores, a que las eras son más largas, AccB no mantiene historia de eras pasadas, por lo que la cantidad de entradas entre todas las tablas de todos los hilos es menor, lo que se traduce en una menor sobrecarga de memoria.

HapB es un algoritmo de detección completo, en el sentido de que si una carrera estática se manifiesta – es decir da lugar a una carrera dinámica – , este algoritmo la detecta. Su precisión por lo tanto es máxima y puede emplearse como referencia. Por su parte, AccB aunque no es completo, si que es capaz de detectar, para la mayoría de aplicaciones, todas las carreras manifestadas en ellas lo que demuestra su elevada precisión y la eficiencia de nuestra implementación.

B.3.6. Variación de la precisión

Al aumentar la granularidad con la cual se mantiene la información de coherencia, se introducen falsos positivos en el sistema. El estudio experimental que hemos realizado a este respecto muestra que, en estas circunstancias, todos los falsos positivos detectados por AccB son detectados por HapB. Es más, este último detecta incluso unos pocos más, por lo que podemos afirmar que más sensible a la granularidad de lo que es AccB.

B.3.7. Comparativa con detectores comerciales

Para completar la evaluación experimental, hemos comparado AccB con dos detectores comerciales, InspeXE y Helgrind. El resultado de la comparativa es, en lineas generales, que AccB produce una aceleración importante de la ejecución. No quizá suficiente para estar activo en máquinas en producción, pero acercándose cada vez mas a ello. Además, debido a la naturaleza de AccB, como no mantiene historia de eras pasadas, escala bien cuando aumentamos el tamaño de los datos de entrada de los programas y, como no tiene comunicación entre hilos, escala bien cuando aumentamos la cantidad de hilos en los que se divide la aplicación.

B.3.8. Conclusiones

La detección de carreras de datos de manera eficiente es algo deseable en el paradigma multihilo. Este capítulo ha introducido un algoritmo novedoso que se basa en una sencilla instrucción que expone el estado de coherencia al software. Nuestros experimentos muestran que esta técnica requiere notablemente menos espacio e impacta menos al rendimiento, obteniendo precisión casi completa. También proponemos pequeñas extensiones al hardware que reducen aún más el impacto al rendimiento, llevándolo a puntos en los que podría llegar a ser suficientemente rápido como para estar activo de manera continua en el sistema.

Nuestros experimentos también muestran que con este mínimo soporte hardware, una sencilla implementación del algoritmo de detección pone en un aprieto a dos aplicaciones comerciales que han sido usadas y mejoradas a lo largos de los años. Además, nuestra evaluación muestra buena escalabilidad: para la mayoría de las aplicaciones, la penalización en el rendimiento de AccB se mantiene estable o incluso

decrece – en media – al aumentar el numero de hilos o el tamaño de los datos de entrada.

B.4. Tecnologías emergentes de memoria

Han pasado ya varios años desde que se predijo el fin de DRAM como tecnología para la memoria principal, ya que tiene problemas de escalabilidad y de consumo más allá de los 30nm [3]. Estos dos problemas han motivado la investigación en nuevas tecnologías como PRAM, STT-MRAM [57] o FRAM [58], que son prometedoras como sustitutas de la DRAM debido a su mejor escalabilidad. De todas estas tecnologías PRAM ha sido la que ha concentrado más atención de los investigadores por ser la que más cerca está de comercializarse.

La DRAM almacena la información en forma de carga eléctrica – o de ausencia de carga – en un condensador. Los condensadores tienen que ser cargados o descargados en las escrituras. Además, las lecturas son destructivas: para leer una celda, se descarga y se comprueba si produce corriente eléctrica, lo que requiere que la celda sea re-escrita después de cada lectura. Además, el tiempo de retención de la carga no es muy largo, lo que requiere que las celdas sean refrescadas periódicamente. Esto tiene implicaciones en el consumo que sería deseable evitar. En cambio, PRAM es una tecnología *resistiva*. Una celda PRAM consiste en un material calcógeno que puede cambiar entre estado amorfo y cristalino rápidamente, en el orden de $\sim 1\mu s$ que equivale a unos 680 ciclos de CPU [59]. Este material está unido a un calentador que funde el material y a continuación lo enfría pudiendo producir estado amorfo o cristalino. Estos dos estados tienen propiedades físicas distintas, entre ellas la resistencia eléctrica, y por lo tanto, haciendo pasar una corriente a través del material podemos *leer* el estado.

Otra razón por la que PRAM está recibiendo tanta atención es porque es compatible con el proceso CMOS. Esto junto con la posibilidad de reducir el factor de escala más allá de los 20nm y el no necesitar un refresco periódico, ha hecho que muchos investigadores hayan tratado de solventar su principal debilidad: el corto tiempo de vida. Nuestro trabajo en este ámbito persigue este mismo objetivo.

B.4.1. Conocimiento previo

Antes de presentar el trabajo realizado es necesario explicar brevemente algunos conceptos sobre los que se asienta. Por ello procedemos a introducir someramente la tecnología *Phase Change Memory (PCM)* y conceptos básicos de teoría de códigos, haciendo especial énfasis en la compresión, y en concreto el concepto de entropía, clave en la motivación de la técnica propuesta.

Tecnología *Phase Change Memory*

. *Phase Change Memory (PCM)* es una tecnología de memoria que usa las propiedades eléctricas de un material como memoria del almacenaje [60]. Mas precisamente, usa el cambio de la resistencia eléctrica de materiales cuando cambian entre estado amorfo y cristalino. Esta idea fue concebida en la década de los sesenta, pero ha sido en los últimos años, con el uso de aleaciones de cristales calcógenos como $Ge_2Sb_2Te_5$ (GST) cuando ha ganado popularidad.

Una celda consta de dos electrodos entre los cuales se encuentran un elemento calentador y el material calcógeno. El calentador es un material que produce calor cuando es atravesado por una corriente elevando así la temperatura del material calcógeno.

El proceso de escritura consiste en calentar el material por encima del punto de fusión y enfriarlo de nuevo. Si queremos que el material quede en estado amorfo, de alta resistividad, ha de hacerse en un rápido pulso. En cambio si queremos que el material cristalice, reduciendo su resistividad, el pulso tiene que ser más largo, pero de menor intensidad, permitiendo que el material forme una estructura cuando aún está fundido. La forma de leer el estado de la celda consiste en atravesar esta por una pequeña corriente y medir su resistencia.

Las limitaciones de PRAM como reemplazo de DRAM son el tiempo de escritura y la reducida cantidad de escrituras que soporta. Las contracciones y expansiones continuas producidas por las operaciones de escritura resultan en un desacoplamiento del calentador y el GST, dejando la celda en un estado de fallo "bloqueado", momento a partir del cual la celda es aún legible, pero el valor contenido no podrá volver a cambiarse. Las celdas PCM de última generación soportan $10^7 \sim 10^9$ escrituras [61].

Por otro lado, la PRAM ofrece ventajas adicionales sobre la DRAM. Una de ellas es el hecho de que hay estados intermedios entre amorfo y cristalino, que son distinguibles, permitiendo almacenar más de un bit por celda. Otra característica es que al no ser una tecnología basada en cargas no se ve afectada por errores transitorios inducidos por partículas [2, 63].

Teoría de códigos

Los códigos de corrección de errores se llevan usando unos años en computadoras para sobrevivir a errores. Los códigos vienen caracterizados por cuantas palabras distintas los forman, cuántos símbolos del alfabeto se usan en cada palabra y cuantos errores pueden detectar/sobrevivir. Los códigos fueron concebidos para un contexto en el que, aunque se quiere minimizar la cantidad de datos enviados o almacenados, es posible permitirse guardar símbolos extra en los raros casos en los que es necesario. En nuestro contexto tenemos una limitación física sobre la cantidad de símbolos que puede haber en una palabra de código; si una palabra necesita más símbolos de los que podemos representar se genera una situación de fallo. Por lo tanto estos códigos no constituyen una solución válida para el problema que pretendemos afrontar.

La compresión es una parte importante de la teoría de códigos. La compresión es una aplicación de la teoría de códigos que recibe símbolos de un alfabeto y los transforma en símbolos que no son necesariamente del mismo alfabeto, de manera que las combinaciones más comunes del alfabeto de entrada son traducidas a palabras de corta longitud en el alfabeto de salida. Una técnica común conocida como Codificación de Huffman funciona literalmente así: toma los símbolos del código de entrada, los ordena por frecuencia y va asignando palabras del código de salida de manera que a mayor frecuencia de repetición, menor longitud. Otra posibilidad es codificar las palabras cuyos prefijos han sido ya observados en el proceso de codificación como una referencia a la última aparición del prefijo junto al resto de la cadena. El objetivo es conseguir una reducción en la cantidad de información necesaria para representar el caso frecuente.

Existen varias propuestas para el uso de compresión en distintos puntos de la jerarquía de memoria, pero el objetivo de estas técnicas es acelerar las transacciones del bus o la "expansión virtual" del espacio disponible. En nuestra propuesta la idea es usar los bits aún útiles del bloque para almacenar el resultado de la compresión del bloque y de este modo poder seguir usándolo. Esto requiere que la información pueda comprimirse a un tamaño igual o inferior a la cantidad de celdas útiles que quedan en el bloque. La *entropía* es una cuantificación de la cantidad de información contenida en una palabra, y da una idea de cuan bien los datos van a comprimirse, sin tener en cuenta el esquema de compresión usado, por lo que nos puede ser de una gran utilidad.

Entropía como medida de *compresibilidad*

En el área de teoría de la información la *entropía* es una manera de estimar cuanta información transmite un símbolo dado, una palabra, o un texto completo. La entropía se relaciona con la probabilidad de aparición de los símbolos. Por ejemplo, si tenemos el alfabeto $\mathcal{A} = \{0, 1\}$ y consideramos la palabra "00000010000100000010″, la probabilidad de que un símbolo de la palabra sea 0 es $\frac{17}{20}$ y la probabilidad de que sea uno es $\frac{3}{20}$, en este sentido, el enunciado *El símbolo* n^{esimo} *es un 1* aporta mucha más información que el enunciado *El símbolo* n^{esimo} *es un 0*, ya que 0 es el valor más probable. La existencia de este desequilibrio reduce la entropía, permitiendo que almacenásemos únicamente las posiciones de los unos para comprimir la palabra. Por el contrario, si consideramos la palabra "01100111001100110010″, $P(0) = P(1) = 0{,}5$ lo que hace que los símbolos sean impredecibles, por lo que la palabra es más difícil de comprimir.

Para aplicar nuestra técnica es necesario que la huella en memoria de los programas que ejecutamos sea *comprimible*, considerando los bloques de cache como palabras de 64 símbolos de longitud y siendo cada byte considerado como un símbolo distinto. Para ello tendremos en cuenta dos valores distintos: (1) la **entropía media**, que se calcula como la media de la entropía de todos los bloques que son expulsados de cache y respaldados en memoria; (2) la **entropía total** es la entropía del flujo de

Media	Total	Significado
baja	baja	Tanto los símbolos como el lenguaje son regulares (predecibles) ⇒ Altamente comprimible.
baja	alta	Aunque el lenguaje como tal no es regular, las palabra sí, y así la compresión a nivel de palabra funciona bien.
alta	baja	Las palabras no son comprimibles pero el lenguaje si ⇒ Es posible recodificar los símbolos de manera que, Las palabras constan de 64 símbolos, pero cada uno más pequeño.
alta	alta	Es poco probable que la compresión funcione bien.

Cuadro B.3: Significado de las diferentes combinaciones de valores de las entropías total y media.

información entre el último nivel de cache y la memoria principal. En otras palabras, la entropía media primero calcula la entropía de cada bloque, y luego agrega todos los valores usando la media, mientras que la entropía total agrega todos los bloques creando un texto y luego calcula la entropía de dicho texto.

La Tabla B.3 recoge el significado de estas dos medidas cuando se consideran juntas. El mejor caso es que la entropía media sea baja, porque eso significa que la compresión funcionaría bien y podemos aplicarla. Si la entropía media es alta pero la entropía total es baja, los programas pueden aplicar una fase de *cálculo de símbolos comunes* que genere una función $f : \mathcal{A} \rightarrow \cup_{n=0}^{N}\{0,1\}^n$ como puede ser la codificación de Huffman para recodificar los símbolos del alfabeto para aplicar codificación a nivel de símbolo. Aunque esto sea posible, el requisito hardware de este escenario es mayor.

Si aplicamos estas medidas a la suite SPEC2006 (ver sección 4.2) el resultado es que la entropía máxima de los bloques de un programa dado está, en un elevado porcentaje de los casos, por debajo de 5. Teniendo en cuenta que cada palabra tiene 6 símbolos, esto quiere decir que hay más de un 16,6 % de información redundante en cada palabra. Por lo que respecta a la entropía media, los números son prometedores incluso para las aplicaciones de punto flotante, que tienen mucha mayor variabilidad que las aplicaciones enteras. La entropía es menor de 2.5 en casi todos los casos, lo que hace pensar que las tasas de compresión son elevadas. Si nos fijamos en la entropía total, los valores son realmente bajos. Esto se debe a la gran cantidad de ceros que se escriben en memoria. En preciso señalar no obstante que estos valores cambian ligeramente según el tamaño que consideremos para el último nivel de cache. Esto se debe a que cuanto más grande es la cache de último nivel menor es la cantidad de expulsiones y por tanto la cantidad de escrituras de respaldo de cache a memoria. En estos escenarios la carencia de variedad estadística puede viciar los resultados hacia valores que no son consistentes con los resultados observados cuando varios procesadores compiten por la cache, lo que resulta en una reducción efectiva del espacio de cache del que disfruta cada procesador.

Punteros de corrección de errores (ECP)

ECP [64] es una de las técnicas más exitosas de los último años para sobrevivir a fallos de las celdas PCM. Cuando se propuso mejoraba la ejecución de todas las demás técnicas existentes.

La idea detrás de ECP es usar pares $< p_i, r_i >$ formados por un puntero, p_i, y una celda de reemplazo, r_i, de manera que cuando una celda del bloque falla, se usa el puntero de uno de estos pares para indexarla, y se usa su celda de reemplazo en su lugar. Esta técnica ha sido diseñada con la restricción de que la sobrecarga en memoria no supere el 12.5 % lo que en nuestro caso dota a cada bloque de 6 pares puntero-reemplazo, permitiendo sobrevivir a 6 fallos.

El funcionamiento de este mecanismo sería el siguiente. Cuando un bloque se escribe en memoria principal, se realiza una lectura del valor escrito y se compara con el valor que se esperaba escribir. En caso de discrepancia se reserva un par $< p_i, r_i >$ y el índice de la celda discrepante se escribe en p_i. Todas las celdas de reemplazo se actualizan con el valor esperado. En caso de que el fallo se produzca en una celda de reemplazo, la solución es alojar otro puntero para ese mismo índice y escribir el valor.

Cuando un bloque se lee de memoria principal se aplican los cambios señalados por los punteros en orden de antigüedad, de manera que si una celda tiene más de un puntero de corrección el valor que se considera es el del par más recientemente alojado.

B.4.2. Nuestra técnica

En esta sección presentamos tres variantes de nuestra técnica basada en compresión: COMP, que comprime la información asociada a un bloque y la almacena en los bits funcionales del bloque para prolongar el tiempo de vida efectivo. $COMP_{CP}$ es una variación de COMP que usa un esquema de compresión que se adapta mucho mejor a este contexto. Finalmente introducimos CEPRAM, que usa emparejado de bloques de grano fino junto con la capacidad de ignorar símbolos para prolongar el tiempo de vida efectivo aún más.

A continuación introducimos los esquemas de compresión empleados, explicando en cada caso las mejoras que hemos incorporado (para una información más detallada referirse a la sección 4.3).

Compresión LZW sin tabla (NTZip)

LZW es un algoritmo tradicional de compresión de texto usado en muchos formatos y aplicaciones. Tomamos LZW como punto de partida por ser conocido y usado extensamente. LZW considera un diccionario inicial precargado con los símbolos del alfabeto, que en nuestro caso son los 256 valores que puede adoptar un byte.

A continuación el proceso es ir concatenando símbolos de la entrada hasta que la cadena resultante no aparece en el diccionario, tras lo cual, el índice en el diccionario del último prefijo que sí está en el diccionario se escribe en la salida con la anchura mínima necesaria, y a continuación esta concatenación es añadida al diccionario como nueva palabra, y consideramos la entrada a partir del último símbolo leído, incluido éste. El proceso se reitera hasta que se consume toda la entrada.

Por la naturaleza dinámica de la generación del diccionario, para la decodificación no hace falta transmitir el diccionario ya que puede inferirse *al vuelo* durante el proceso de decodificación.

LZW saca partido de las repeticiones de cadenas en el texto de entrada y las recodifica como símbolos de menor longitud.

Nuestra propuesta, NTZip es una variación de LZW en la que el diccionario empieza vacío, y se van añadiendo símbolos según aparecen. Esto requiere que los símbolos de salida tengan un bit extra para indicar si el resto del símbolo son 8 bits correspondientes a un símbolo de entrada que aún no está en el diccionario, o si el resto del símbolo se corresponde a un índice en el diccionario. A primera vista parece que esta modificación penaliza a los bloques con muchos símbolos distintos en la entrada, ya que en ese caso se requieren 9 bits por símbolo de salida, el prefijo y los 8 bits del nuevo símbolo que ha de ser insertado en el diccionario. Analicemos más en profundidad este caso: cuando comprimimos con LZW el primer símbolo es un índice en una tabla con 256 entradas – 8 bits de anchura – a continuación se inserta una nueva cadena en el diccionario y los subsiguientes símbolos pasan a tener 9 bits de anchura. En este caso por lo tanto NTZip sólo ocuparía un bit mas que LZW sobre una salida de 577 bits. La mayor ventaja de esta modificación es que el diccionario empieza con longitud 0 y puede llegar, a lo sumo, a 63 entradas, lo que hace que cuando una cadena se repite y es indexada usa entre 0 y 6 bits más uno de prefijo, en vez de los 9 necesarios para los símbolos en LZW.

El uso de NTZip requiere de bits de metainformación para describir el estado del bloque. Si queremos mantener la sobrecarga acotada por debajo del 12.5 % de memoria, tal y como ocurre con ECP_6 [64], SEC_{64} [68], $Pairing_8$ [69], $Wilkerson_4$ [70], sólo se pueden emplear a 64 bits extra por bloque.

La idea de COMP es usar ECP_6 hasta que ocurre el sexto fallo. Una vez llegamos a 7 fallos, en vez de descartar el bloque lo comprimimos. De los 64 bits de metainformación que se usaban en ECP, ahora usaremos 4 para marcar que el bloque está comprimido, y los otros 60 los dividimos en 7 punteros de 8 bits de longitud cada uno, lo cual permite direccionar 7 parejas de bits de las 256 que forman el bloque (512 bits $\Rightarrow$ 256 parejas). Con estos 7 punteros apuntamos las (hasta) 7 parejas que contienen al menos un bit fallido, lo cual nos permite ignorar esas posiciones, y tras la compresión de la información la almacenamos en aquellos bits no fallidos. El proceso se repite, con cada nuevo fallo, aumentamos el número de punteros reduciendo el tamaño de estos, lo cual requiere descartar grupos de bits cada vez más grandes, pero nos permite seguir describiendo qué partes del bloque hay que evitar al escribir la información comprimida. Este método nos permite sobrevivir hasta 10 fallos mientras la información contenida sea razonablemente comprimible. Una vez

se alcanza el undécimo error o en una escritura no se consigue alojar la información en los bits funcionales se descarta el bloque y con él la página completa.

Uno de las mayores limitaciones de este esquema es que, cuando hay 10 fallos de tamaño bit, debido a las restricciones de espacio en el almacenamiento de punteros, tenemos que descartar 10 bytes completos, reduciendo artificialmente el tamaño disponible para almacenar el dato comprimido.

NTZip con borrado

Como acabamos de indicar, el mayor problema de COMP es que se desperdicia mucho espacio por el tamaño limitado de los punteros. En esta sección proponemos no almacenar explícitamente qué celdas fallan. Dado que las celdas con fallo de bloqueo aún son legibles, se manifiestan en un error sólo el 50 % de las veces. El razonamiento de esto es que si asumimos que la probabilidad de que un bit sea 0 o que sea uno es la misma, la probabilidad de que una celda se intente escribir con un valor distinto al que tiene, manifestándose el fallo, es sólo del 50 %. Esto quiere decir que aunque tengamos n celdas con fallo, al escribir, en media, sólo $\frac{n}{2}$ se manifiesta en forma de error. Por ello proponemos NTZipBs, que es una modificación de NTZip en la que el índice 0 de la tabla se usa para codificar un símbolo de borrado. Durante la descompresión, si se encuentra un símbolo de borrado, el símbolo previo es ignorado, tanto en términos de salida como en términos de actualización del diccionario. De esta manera sólo emprendemos acciones de corrección en caso de que el fallo se manifieste.

El punto débil de esta solución es que la escritura se hace iterativa, ya que cada vez que se detecta un fallo en la escritura hay que insertar un símbolo de borrado a continuación y después continuar con la escritura del resto del bloque. En el peor de los casos, la escritura de un bloque con n fallos, si todos se manifiestan en errores, puede requerir hasta $n + 1$ escrituras consecutivas. Otro problema grande es que la decodificación de LZW, NTZip y NTZipBs no es paralelizable cuando el tamaño de bloque es tan pequeño, lo cual tiene un impacto negativo en el rendimiento. El proceso de codificación tampoco es paralelizable, pero al no estar en el camino crítico esto no es tan problemático.

C-Pack y C-PackBs

C-Pack [65] es un esquema de compresión de cache para máquinas de alto rendimiento. $COMP_{CP}$ es una variación de COMP que usa C-Pack en vez de NTZip como esquema de compresión. La evaluación de C-Pack presentada en [65] muestra que este algoritmo se adapta muy bien este contexto (bloques pequeños), sin afectar negativamente el rendimiento. Además en este mismo artículo se presentan los esquemas necesarios para realizar la compresión y descompresión en hardware, que pueden ser aprovechados en nuestro contexto. Tan sólo es preciso realizar una pequeña modificación en el decodificador para descartar los conjuntos de bits con fallos.

C-Pack considera símbolos de entrada de 32 bits, que clasifica según unos pocos patrones que incluyen que el símbolo sean todo ceros, que tenga la mitad superior ceros, que coincida total o parcialmente con símbolos previos o que sea un símbolo aún no observado.

En nuestro contexto el uso de C-Pack también requiere, al igual que sucede con NTZip, descartar las celdas bloqueadas. Por lo que tenemos la misma motivación para introducir un símbolo de borrado, en lo que hemos llamado C-PackBs.

En C-PackBs, el caracter de borrado $< bs >$ no es, estrictamente hablando, un borrado, si no más bien un símbolo de corrección que va seguido por una máscara de 5 bits, que lo convierten en un símbolo que ocupa un byte. Esta máscara indica qué bytes del símbolo anterior contienen un error y por tanto han de ser corregidos. A continuación del byte de $< bs >$ hay un byte por cada uno en la mascara. Estos bytes son los patrones de error que al ser operados – mediante una OR exclusiva – con el byte erróneo correspondiente dan lugar al símbolo correcto.

Prolongando aún más la vida mediante el emparejamiento de grano fino

La última propuesta de esta tesis es CEPRAM: compresión de memoria con emparejamiento a nivel de bloque para una extensión del tiempo de vida en memorias resistivas. La idea de CEPRAM es trabajar como $COMP_{CP}$ durante los 10 primeros fallos. CEPRAM lleva la cuenta de los bloques descartados. Cuando un bloque encuentra su undécimo fallo, si la lista de bloques descartados está vacía, el bloque y toda su página se descarta del sistema y se añade a la lista. Si por el contrario la lista no está vacía, se toma el primer bloque de la lista como bloque de "desbordamiento" para el bloque que está experimentando el fallo, el cual se marca con una nueva codificación en sus 4 bits de estado para denotar que es *Maestro* en la pareja, y el bloque de la lista se marca como *Esclavo*. De esta manera, cuando hay que escribir/leer datos del bloque, se comienza por el maestro, aplicando C-PackBs como esquema de compresión, y si el espacio no es necesario se continúa por el bloque de desbordamiento, permitiendo hasta 1024 bits de almacenamiento, algunos de los cuales pueden contener fallos.

Equilibrado del desgaste

Para CEPRAM usamos técnicas de equilibrado del desgaste como las descritas en [71] y [72]. Sin embargo, para nuestra técnica sólo rotamos la parte de datos, y no la de metadatos. La razón es que queremos que los 4 bits usados para codificar el estado del bloque sufran la menor cantidad de modificaciones posible. En nuestro diseño, si un bloque de memoria principal atraviesa cada uno de los estados, cada bit es escrito nada más que 3 veces, minimizando la probabilidad de que se produzca en ellos un fallo de bloqueo. Asimismo nos interesa que los punteros estén tan sanos como sea posible. Esta decisión resulta en más desgaste de los bits de datos a cambio de menos desgaste en los bits de metadatos.

Emparejado múltiple

CEPRAM sólo empareja con un bloque, pero podríamos diseñar un esquema en el que haya múltiples bloques de desbordamiento. En nuestra técnica no hemos considerado esa posibilidad ya que cuando la situación en la que es necesario añadir un segundo bloque de desbordamiento ocurre, el sistema esta cerca de la disfunción, por lo que no tiene sentido "prolongar la agonía".

Estimación del tiempo de vida

De acuerdo a nuestros estudios, el tamaño medio de un bloque comprimido con C-Pack son 320 bytes. Para cada fallo, en el peor caso, necesitamos 2 bytes para corregirlo, el símbolo de $< bs >$ y el patrón de error, es decir, 16 bits por fallo. Si un bloque tiene 512 bits y 320 tienen información nos quedan $512 - 320 = 192$ bits = 24 bytes, lo que nos permite corregir hasta 12 errores antes de descartar el bloque. En el caso de que tengamos el bloque emparejado esta cifra se eleva a 44 errores. Contando que la probabilidad de que un fallo se manifieste en un error es 0.5, esto puede corregir, en media, hasta casi 90 fallos por bloque. A continuación mostramos la evaluación experimental.

B.4.3. Evaluación

Nuestra evaluación experimental usando la suite SPEC2006 tiene 2 partes. La primera es la evaluación de NTZip/NTZipBs y C-Pack/C-PackBs como esquemas de compresión para este contexto y la segunda es evaluar el tiempo de vida de la memoria cuando se usa COMP,$COMP_{CP}$ y CEPRAM.

Evaluación de los esquemas de compresión

En general, todos los algoritmos de compresión hacen un buen trabajo a la hora de comprimir la huella de memoria de las aplicaciones de SPECINT. Comprimir la memoria de las aplicaciones de SPECFP es más difícil. Ninguno de los esquemas considerados está orientado a punto flotante. La mayor variabilidad intrínseca de los datos en punto flotante es la razón por la que muestran un rendimiento peor que el código entero. Sin embargo, en media, el 75 % de los bloques de las aplicaciones de punto flotante se comprimen por debajo de los 512 bits, y el 20 % del total son bloques formados por un byte repetido 64 veces.

En esta evaluación el tamaño de la cache de último nivel no afecta significativamente ni al porcentaje de bloques constantes escritos a memoria ni al porcentaje de bloques que crecen en tamaño al comprimirse.

Cabe resaltar que C-Pack mejora los resultados de NTZip para código de punto flotante, reduciendo la cantidad de bloques que desbordan el tamaño desde el 25 % hasta el 20 %, lo que lo hace aún más atractivo para este contexto.

Al considerar las versiones con el carácter de borrado en la implementación de CEPRAM, NTZipBs es capaz de sobrevivir al menos 27 fallos en cada pareja de bloques (13.5 fallos por bloque), llegando hasta 48 para dos aplicaciones. C-PackBs mejora el comportamiento, siendo capaz de sobrevivir al menos 30 fallos por pareja de bloques. Aunque el máximo es más pequeño, sólo 45, estas dos aplicaciones son las únicas en las que NTZipBs muestra mejores resultados que C-PackBs.

Comparativa con propuestas previas

Comparamos nuestra técnica con otras propuestas de la literatura, a saber ECC (Códigos de corrección de errores), DRM (replicación dinámica de memoria) y ECP (Punteros de corrección de errores).

En términos de errores teóricamente "sobrevividos" por página nuestra técnica multiplica casi por 10 la cantidad de errores "sobrevividos" de otras técnicas. Aunque esto no implica necesariamente un aumento en el tiempo de vida, ya que es un número sin correlación temporal si que es indicativo de un mayor potencial de supervivencia a fallos. Hay que tener en cuenta, además, cuánto se parece el comportamiento real al teórico, porque puede ocurrir que una página sea descartada porque un único bloque acumula un número elevado de fallos mientras que el resto están bastante sanos, lo cual resulta en un bajo aprovechamiento de la técnica. En este estudio DRM es la técnica que más partido saca a sus recursos, seguido por CEPRAM. A continuación está ECP que sólo aprovecha de manera efectiva el 50 % de los recursos de corrección de errores, siendo ECC el peor parado en esta comparativa, consiguiendo sólo el 5 % de aprovechamiento.

Si hacemos un análisis dinámico, simulando un sistema abstracto en el que aplicamos los modelos extraídos de la simulación de las aplicaciones, y representamos gráficamente el porcentaje de memoria útil respecto del tiempo (ver figura 4.10), observamos que: (1) ECP mejora al resto de técnicas; (2) CEPRAM mejora a ECP, si al igual que en la propuesta de DRM tomamos como referencia el momento en el que la cantidad de páginas útiles baja del 50 %, CEPRAM mejora, en el caso medio, un 23 % el tiempo de vida sobre ECP. Cabe destacar que por su naturaleza CEPRAM es más efectivo cuando la probabilidad de que una escritura modifique un bit es alta, porque en este caso el resto de esquemas sufren más, y en el mejor caso la extensión de vida supera el 50 %.

Terminamos nuestra evaluación con un razonamiento teórico. A medida que se producen más fallos, cada vez es más difícil corregirlos y cada vez es menor el tiempo entre fallos, por lo que llega un punto a partir del cual no tiene sentido seguir corrigiendo fallos, pues la rentabilidad de la inversión es demasiado baja. En nuestra evaluación teórica este punto se sitúa en torno a los 29 fallos, ya que una vez el fallo vigesimonoveno es corregido, la extensión de vida por fallo corregido baja por debajo del umbral del 0.5 %, lo cual es un margen excesivamente bajo. De acuerdo con este razonamiento, CEPRAM hace un buen trabajo desplazando la curva que muestra porcentaje de memoria útil respecto del tiempo hacia la derecha.

B.4.4. Conclusiones

Aunque las memorias resistivas están cada vez más cerca de la producción industrial, para poder sustituir a la tecnología DRAM a nivel de la memoria principal, es necesario resolver primero el problema que plantea el escaso tiempo de vida de sus celdas.

En este capítulo hemos presentado CEPRAM, un intento de hacer la memoria PRAM más duradera usando compresión de bloques de memoria. Nuestra técnica extiende ECP usando un esquema de compresión de alto rendimiento enfocado a bloques de cache convenientemente modificado para adaptarse a nuestro contexto. Esta propuesta consigue prolongar el tiempo de vida por encima de lo conseguido previamente. En particular, consigue garantizar la "supervivencia" de al menos la mitad de las páginas físicas durante un 25 % más de tiempo que ECP, lo que supone un 5 % más que un esquema ideal que soporte hasta 16 fallos por bloque.

Además de presentar la técnica, hacemos un análisis de por qué PCM no es el contexto idóneo para compresión y por qué no es una buena idea concentrar esfuerzos en mejorar el algoritmo de compresión usado, a no ser que se conciba la compresión de un modo distinto, por ejemplo centrándose en el caso peor en lugar de en el medio.

Finalmente mostramos un estudio de la mejora del tiempo de vida como función de la cantidad de errores sobreviidos por bloque en el que se dan ideas para decidir una cota razonable de la capacidad de recuperación de errores necesaria en función de la extensión de vida útil que queramos conseguir. Este estudio muestra que aún hay sitio para mejora más allá de CEPRAM, pese a que CEPRAM estira el tiempo de vida hasta un punto en el que prolongar aún más la vida tiene un elevado coste en términos de la cantidad de fallos que se debe ser capaz de corregir por bloque.

B.5. Conclusiones y trabajo futuro

El sistema de memoria es una parte importante de toda computadora, como refleja toda la investigación que se le ha dedicado desde el nacimiento de los computadores.

El sistema de memoria es un tema extenso, que plantea múltiples retos. Estos retos provienen de fuentes diversas. Algunas son intrínsecas, como el consumo o el rendimiento. Otras aparecen para suplir una necesidad del software surgida en algún momento, como por ejemplo, la necesidad del sistema operativo de aislar los mapas de memoria de procesos distintos, lo que dio origen a la memoria virtual, y presentó problemas de rendimiento que se resolvieron mediante el uso del TLB. Otros retos, en cambio, se deben a la tecnología. PCM es un buen ejemplo: a cambio de las ventajas que aporta en términos de escalabilidad, la adopción de esta nueva tecnología presenta nuevos problemas. También hay que resolver situaciones derivadas de la aplicación. Algunos aspectos de seguridad no han sido un problema – o al menos no tan importante – hasta que la tecnología nos dio la oportunidad de hacer, por ejemplo, pagos a través de internet u operaciones bancarias. Otro

ejemplo, seguir la pista de los accesos a memoria durante la depuración no era tan relevante antes de la adopción general de multiprocesadores de memoria compartida.

En vez de centrarnos en un problema, o en una parte específica de la memoria, este trabajo ha atacado tres grandes problemas, cada uno desde el punto de vista que hemos considerado más interesante.

Nuestro Filtro de Pila (*Stack Filter*, SF) sigue la filosofía de mantener un hardware simple para reducir consumo de potencia. Saca partido de la información proporcionada por uno de los registros de propósito general para saber que accesos a memoria residen en la pila, y aprovecha la excepcional localidad de estos accesos. Como demostramos esta localidad puede ser capturada con una estructura realmente simple, reduciendo por consiguiente el consumo de energía.

Igual que ocurre con el consumo, hay un gran interés en que las computadoras sean fiables tanto en términos de hardware como de software. Los errores de software llevan a situaciones que pueden dañar la reputación de la empresa que hay detrás de ese software, pueden ocasionar pérdida o corrupción de datos, problemas en la comunicación con sondas espaciales y muchas otras situaciones no deseadas. A medida que pasa el tiempo y los sistemas aumentan en tamaño y también aumentan las aplicaciones que corren sobre ellos, incrementando la dificultad de probar y verificar el sistema. Esta dificultad aumenta dramáticamente cuando un programa evoluciona de ejecutarse en un solo procesador a ejecutarse en un multiprocesador con hilos que se comunican. El comportamiento no determinista que introduce este cambio es una motivación para el desarrollo de herramientas que sigan la pista de las referencias a memoria para asegurar que la semántica pretendida por el programador se preserva.

Analizar los accesos a memoria para detectar errores de concurrencia no es tarea fácil, ya que suele suponer una elevada sobrecarga en tiempo, memoria y/o coste. Las técnicas hardware no tienen problemas de rendimiento, pero incrementan el coste y, al tener recursos limitados pierden precisión. Las técnicas software son normalmente demasiado lentas y consumen más memoria. Con AccB ofrecemos una alternativa híbrida para la detección de carreras de datos que, aunque no está madura para estar activa por defecto en códigos de producción, ofrece una sustancial mejora sobre las herramientas comerciales. Además, comparada con los mecanismos puramente hardware, aunque es más lenta, reduce su coste a un valor ínfimo al tiempo que ofrece una mayor precisión y flexibilidad. Un ejemplo de esta flexibilidad es la posibilidad de combinarla con técnicas como Aikido [89], orientadas a reducir las secciones instrumentadas al mínimo, lo que permitiría acelerar drásticamente la detección de carreras de datos.

La evolución de las computadoras no sólo ha acentuado la necesidad de herramientas para controlar la corrección de los programas, sino que también ha llevado al límite algunos componentes y tecnologías. Una de esas tecnologías es DRAM. Enfrentada al límite de escalado, los fabricantes necesitan un reemplazo para solucionar ese problema. La tecnología PCM es una candidata prometedora para sustituir DRAM como tecnología de memoria principal. Sin embargo acarrea nuevos retos para la industria y el mundo académico: hacer frente al escaso tiempo de vida de sus cel-

das. Nosotros hemos atacado este problema desde una nueva perspectiva. Hasta la fecha, nadie había usado compresión de datos como medio para dotar a un bloque de datos de la capacidad de ignorar símbolos. Con nuestra técnica somos capaces, después de un emparejado de bloques de grano fino, de sobrevivir hasta 30 manifestaciones de fallos por bloque, lo que en la práctica es un número de fallos elevado, ya que la probabilidad de un fallo de manifestarse es 0.5, en media. Hemos demostrado experimentalmente que CEPRAM prolonga la vida de dispositivos basados en PCM más allá que las propuestas existentes.Asimismo hemos intentado aportar suficiente información para justificar nuestro aserto de que, aunque hemos probado sus beneficios, este no es el contexto idóneo para aplicar compresión de datos tal y como se concibe hoy día, y que por lo tanto no merece la pena invertir esfuerzos adicionales en mejorar el algoritmo de compresión empleado. Por último, hemos realizado un análisis del punto a partir del cual los beneficios de poder sobrevivir a un fallo más no compensan sobre el esfuerzo empleado, y hemos mostraado que CEPRAM alcanza resultados próximos a este punto.

Nuestra conclusión final es que el sistema de memoria es un área amplia en la que hay muchos aspectos que tratar. Algunos son perpetuamente atractivos, otros son motivados por cambios en la tecnología o en la organización del sistema o surgen para suplir una necesidad del software. No importa cual es el origen de estas situaciones, todas son importantes y se les debe dedicar atención. Esta tesis se centra en tres problemas y ofrece soluciones que mejoran los últimos avances, aunque aún distan de ser consideradas soluciones definitivas y por lo tanto tienen margen de mejora.

B.5.1. Trabajo futuro

En esta tesis hemos aliviado los problemas en los que nos hemos centrado, sin embargo aún queda trabajo por hacer. Hay mucho margen de mejora, y de entre los objetivos a corto plazo para mejorar la situación actual queremos resaltar los siguientes:

- Llevar el filtrado semántico más allá de la cache DL1. Hemos mostrado el potencial de saber a que región de memoria referencia un acceso. Esto puede ser usado en otros niveles de la jerarquía. Aikido [89] muestra una manera para diferenciar páginas compartidas de páginas privadas. Esto puede ser usado para filtrar tráfico de coherencia declarando datos privados a hilos, datos compartidos de sólo lectura y datos compartidos coherentes. Esto puede hacerse para reducir las acciones de coherencia al conjunto coherente. El filtrado semántico puede también aplicarse a memoria principal PCM: Algunos segmentos de un proceso son más aptos para PCM (el texto, datos de sólo lectura) mientras que otros se adaptan mejor a DRAM (la pila), y partes distintas de la región dinámica pueden exhibir distintos comportamientos que los conviertan en candidatos para PCM o DRAM. Esta idea puede usarse para hacer una jerarquía híbrida con parte DRAM y parte PRAM.

- Los retos que acarrea PCM han sido atacados principalmente al nivel de la propia PCM. Pero toda la jerarquía puede ayudar, ya que son los recursos de la jerarquía y la manera en que se gestionan lo que determina dónde está cada dato en todo momento. Hay técnicas que se pueden hacer a nivel de registros [90], o del último nivel de cache [78]. Aún hay margen de mejora en el aspecto de gestión de las caches: política de reemplazo, de inclusión y otros aspectos que pueden reducir, directa o indirectamente, la cantidad de escrituras a memoria principal. Estos aspectos también afectan al comportamiento de PCM, pero dado que los modelos de consumo/latencia de PRAM varían tanto de los de DRAM, todos los efectos laterales han de ser considerados y equilibrados para adaptarse a las restricciones del nuevo modelo.

B.5.2. Publicaciones

Este trabajo ha dado fruto a las siguientes publicaciones:

- R. Gonzalez-Alberquilla, F. Castro, L. Piñuel and F. Tirado. Stack Oriented Data Cache Filtering. In *International Converence on Hardware/Software Codesign and System Synthesis*, pages 257-266. Grenoble, France, October 2009.
- R. Gonzalez-Alberquilla, F. Castro, L. Piñuel and F. Tirado. Stack Filter: Reducing L1 Data Cache Power Consumption. In *Journal on Systems Architecture* 56(12), pages 257-266. December 2010.
- R. Gonzalez-Alberquilla, K. Strauss, L. Ceze and L. Piñuel. Accelerating Data Race Detection with Minimal Hardware Support. In *Proceedings of the 17th international conference on parallel processing* Europar'11, pages 27-38. Bordeaux, France. September 2011.
- R. Gonzalez-Alberquilla, F. E. Frati, K. Strauss, L. Ceze and L. Piñuel. Data Race Detection with Minimal Hardware Support. To appear in *Computer Journal*, Oxford University Press.

List of Figures

List of Tables

Bibliography

[1] S. Naffziger, J. Warnock, and H. Knapp. Se2 when processors hit the power wall (or "when the cpu hits the fan"). In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 16 –17, feb. 2005.

[2] S.S. Mukherjee, J. Emer, and S.K. Reinhardt. The soft error problem: an architectural perspective. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 243 – 247, feb. 2005.

[3] K. Kim. Technology for sub-50nm dram and nand flash manufacturing. In *International Electron Devices Meeting 2005*, pages 323–326, Washington, December 2005.

[4] M. A. Viredaz and D. A. Wallach. Power evaluation of a handheld computer. *IEEE Micro*, 23(1):66–74, January 2003.

[5] J. Montanaro et al. A 160 Mhz, 32-b, 0.5W CMOS RISC Microprocessor. *Digital Technology Journal*, 9(1):49–62, 1997.

[6] K. Ghose and M. B. Kamble. Reducing Power in Superescalar Processors Caches using Su-banking, Multiple Line Buffers and Bit-Line Segmentation. In *International Symposium on Low-Power Electronics and Design*, pages 70–75, August 1999.

[7] C. L. Su and A. M. Despain. Cache Designs for Energy-Efficiency. In *Hawaii International Conference on Systems Sciences*, pages 306–314, January 1995.

[8] P. Racunas and Y. N. Patt. Partitioned First-Level Cache Design for Clustered Microarchitectures. In *International Conference on Supercomputing*, pages 22–31, San Francisco, California, June 2003.

[9] J. Kin, M. Gupta, and W. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. In *International Symposium on Microarchitecture*, pages 184–193, Research Triangle Park, North Carolina, December 1997.

[10] D. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. *Journal of Instruction-Level Paralelism*, 2, 2000.

[11] L. H. Lee, B. Moyer, and J. Arends. Instruction Fetch Energy Reduction Using Loop Caches for Embedded Applications with Small Tight Loops. In *International Symposium on Low-Power Electronics and Design*, pages 267–269, San Diego, California, August 1999.

[12] H. Lee, M. Smelyanskiy, C. Newburn, and G. Tyson. Stack Value File: Custom Microarchitecture for the Stack. In *International Symposium on High-Performance Computer Architecture*, pages 5–14, Monterey, Mexico, January 2001.

[13] S. Cho, P. Yew, and G. Lee. Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor. In *International Symposium on Computer Architecture*, pages 100–110, Atlanta, Georgia, May 1999.

[14] R. Gonzalez-Alberquilla, F. Castro, L. Pinuel, and F. Tirado. Stack Oriented Data Cache Filtering. In *International Conference on Hardware/Software Codesign and System Synthesis*, pages 257–266, Grenoble, France, October 2009.

[15] Ward and Halstead. *Computation Structures*. Kluwer Academics, 2002.

[16] R. Gonzalez-Alberquilla, F. Castro, L. Pinuel, and F. Tirado. Stack filter: Reducing l1 data cache power consumption. *J. Syst. Archit.*, 56(12):685–695, December 2010.

[17] C. S. Ballapuram, A. Sharif, and H. S. Lee. Exploiting Access Semantics and Program Behavior to Reduce Snoop Power in Chip Multiprocessors. In *ASPLOS 08*, pages 60–69, Seattle, Washington, March 2008.

[18] D. R. Ditzel and H. R. McLellan. Register Allocation for Free: The C Machine Stack Cache. 10(2):48–56, 1982.

[19] M. Huang, J. Renau, and J. Torrellas. L1 Data Cache Decomposition for Energy Efficiency. In *International Symposium on Low-Power Electronics and Design*, pages 10–15, Huntington Beach, California, August 2001.

[20] H. Lee and G. Tyson. Region-Based Caching: an Energy-Delay Efficient Memory Architecture for Embedded Processors. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 120–127, San Jose, California, November 2000.

[21] M. Geiger, S. McKee, and G. Tyson. Beyond Basic Region Caching: Specilalizing Cache Structures for High Performance and Energy Consevation. *hipeac*, 2005.

[22] A. Hemsath, R. Morton, and J. Sjodin. Implementing a Stack Cache. Technical report, Rice University, June 2007.

[23] Byung-Do Yang, Jae-Eun Lee, Jang-Su Kim, Junghyun Cho, Seung-Yun Lee, and Byoung-Gon Yu. A low power phase-change random access memory using a

data-comparison write scheme. In *IEEE International Symposium on Circuits and Systems, 2007. ISCAS 2007.*, pages 3014 –3017, may. 2007.

[24] C. Cascaval and D. Padua. Estimating Cache Misses and Locality using Stack Distances. In *International Conference on Supercomputing*, pages 150–159, San Francisco, California, 2003.

[25] Trevor Mudge, Todd Austin, Dirk Grunwald, et al. A SimpleScalar-Arm Power Modeling Project. http://www.eecs.umich.edu/~panalyzer.

[26] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 39(2):59–67, February 2002.

[27] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench:a Free, Commercially Representative Embedded Benchmark Suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, December 2001.

[28] Jason E. Fritts, Frederick W. Steiling, Joseph A. Tucek, and Wayne Wolf. Mediabench ii video: Expediting the next generation of video systems research. *Microprocess. Microsyst.*, 33(4):301–318, 2009.

[29] N. S. Kim, T. Austin, T. Mudge, and D. Grunwald. *Challenges for Architectural Level Power Modeling.* Kluwer Academics, 2002.

[30] http://www.hpl.hp.com/research/cacti/.

[31] Chi-Keung Luk, Robert Cohn, Robert Muth, Haris Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[32] J. Manson, W. Pugh, and S. Adve. The Java Memory Model. In *POPL*, 2005.

[33] C. Nelson and H.-J. Boehm. Concurrency Memory Model (final revision). C++ standards committee paper WG21/N2429=J16/07-0299, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2429.htm`, October 2007.

[34] Sang Lyul Min and Jong-Deok Choi. An Efficient Cache-based Access Anomaly Detection Scheme. In *ASPLOS*, 1991.

[35] Abdullah Muzahid, Darío Suárez, Shanxiang Qi, and Josep Torrellas. SigRace: Signature-Based Data Race Detection. In *ISCA*, 2009.

[36] Milos Prvulovic. CORD: Cost-effective (and nearly overhead-free) Order-Recording and Data race detection. In *HPCA*, 2006.

[37] Milos Prvulovic and Josep Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *ISCA'03: Proceedings of the 30th International Symposium on Computer Architecture*, 2003.

[38] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *HPCA*, 2007.

[39] Paruj Ratanaworabhan, Martin Burtscher, Darko Kirovski, Benjamin Zorn, Rahul Nagpal, and Karthik Pattabiraman. Detecting and Tolerating Asymmetric Races. In *PPoPP*, 2009.

[40] Michiel Ronsse and Koen De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2), 1999.

[41] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

[42] Yuan Yu, Top Rodeheffer, and Wei Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *SOSP*, 2005.

[43] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[44] Anne Dinning and Edith Schonberg. An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. In *PPoPP*, 1990.

[45] Dejan Perković and Peter J. Keleher. Online Data-Race Detection via Coherency Guarantees. In *OSDI*, 1996.

[46] T. Ball, S. Burckhardt, J. de Halleux, M. Musuvathi, and S. Qadeer. Deconstructing concurrency heisenbugs. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 403 –404, Vancouver, Canada, may 2009. IEEE Computer Society, Washington DC.

[47] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *MICRO*, 2002.

[48] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, 2009.

[49] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants. In *ASPLOS*, 2006.

[50] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ISCA*, 2008.

[51] Jochen Schimmel and Victor Pankratius. Exploiting cache traffic monitorinf for run-time race detection. In *Euro-Par'11: Proceedings of the 17th international conference on Parallel processing - Volume Part I*, EuroPar '11, pages 15–26, Bordeaux, France, 2011. Springer-Verlag, Berlin.

[52] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans Boehm. Conflict exceptions: Providing simple concurrent language semantics with precise hardware exceptions. In *ISCA*, 2010.

[53] Vijayanand Nagarajan and Rajiv Gupta. Ecmon: Exposing cache events for monitoring. In *ISCA*, 2009.

[54] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[55] Brandon Lucia and Luis Ceze. Finding concurrency bugs with context-aware communication graphs. In *Micro-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 553–563, New York, NY, USA, 2009. ACM.

[56] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, Toronto, Ontario, Canada, 2008. ACM, New York.

[57] Yiran Chen, Xiaobin Wang, Hai Li, H. Liu, and D.V. Dimitrov. Design margin exploration of spin-torque transfer ram (spram). In *Quality Electronic Design, 2008. ISQED 2008. 9th International Symposium on*, pages 684 –690, march 2008.

[58] A.W. Ruan, B. Hu, and Y.H. Zhai. A parasitic effect - free test scheme for ferroelectric random access memory (fram). In *Testing and Diagnosis, 2009. ICTD 2009. IEEE Circuits and Systems International Conference on*, pages 1 –4, april 2009.

[59] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *Proceedings of the 5th Bienial Conference on Innovative Data Systems Research (CIDR'11)*, jan. 2011.

[60] S. Lai. Current status of the phase change memory and its future. In *International Electron Devices Meeting 2003*, pages 255–258, Santa Clara, CA, December 2003.

[61] R. F.. Freitas and W. W. Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52(4.5):439–447, july 2008.

[62] T. Nirschl, J.B. Phipp, T.D. Happ, G.W. Burr, B. Rajendran, M.-H. Lee, A. Schrott, M. Yang, M. Breitwisch, C.-F. Chen, E. Joseph, M. Lamorey, R. Cheek, S.-H. Chen, S. Zaidi, S. Raoux, Y.C. Chen, Y. Zhu, R. Bergmann, H.-L. Lung, and C. Lam. Write strategies for 2 and 4-bit multi-level phase-change

memory. In *Electron Devices Meeting, 2007. IEDM 2007. IEEE International*, pages 461 –464, dec. 2007.

[63] J. F. Ziegler. Terrestrial cosmic rays. *IBM Journal of Research and Development*, 40(1):19 –39, jan. 1996.

[64] Stuart Schechter, Gabriel H. Loh, Karin Straus, and Doug Burger. Use ecp, not ecc, for hard failures in resistive memories. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, pages 141–152, New York, NY, USA, 2010. ACM.

[65] Xi Chen, Lei Yang, R.P. Dick, Li Shang, and H. Lekatsas. C-pack: A high-performance microprocessor cache compression algorithm. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 18(8):1196 –1208, aug. 2010.

[66] C.D. Benveniste, P.A. Franaszek, and J.T. Robinson. Cache-memory interfaces in compressed memory systems. *Computers, IEEE Transactions on*, 50(11):1106 –1116, nov 2001.

[67] Prateek Pujara and Aneesh Aggarwal. Restrictive compression techniques to increase level 1 cache capacity. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pages 327 – 333, oct. 2005.

[68] Rw Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 26(2):147–160, 1950.

[69] Engin Ipek et al. Dynamically replicated memory: building reliable systems from nanoscale resistive memories. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2010.

[70] Chris Wilkerson, Hongliang Gao, Alaa R. Alameldeen, Zeshan Chishti, Muhammad Khellah, and Shih-Lien Lu. Trading off cache capacity for reliability to enable low voltage operation. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 203–214, Washington, DC, USA, 2008. IEEE Computer Society.

[71] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–23, New York, NY, USA, 2009. ACM.

[72] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[73] Byung-Do Yang, Jae-Eun Lee, Jang-Su Kim, Junghyun Cho, Seung-Yun Lee, and Byoung-Gon Yu. A low power phase-change random access memory using a data-comparison write scheme. In *IEEE International Symposium on Circuits and Systems, 2007. ISCAS 2007.*, pages 3014 –3017, may. 2007.

[74] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 14–23, New York, NY, USA, 2009. ACM.

[75] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. *SIGARCH Comput. Archit. News*, 37(3):2–13, June 2009.

[76] Wangyuan Zhang and Tao Li. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 101–112, Washington, DC, USA, 2009. IEEE Computer Society.

[77] Sangyeun Cho and Hyunjin Lee. Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 347–357, New York, NY, USA, 2009. ACM.

[78] Miao Zhou, Yu Du, Bruce Childers, Rami Melhem, and Daniel Mossé. Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems. *ACM Trans. Archit. Code Optim.*, 8(4):53:1–53:21, January 2012.

[79] Nak Hee Seong, Dong Hyuk Woo, and Hsien-Hsin S. Lee. Security refresh: prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, pages 383–394, New York, NY, USA, 2010. ACM.

[80] Nak Hee Seong, Dong Hyuk Woo, V. Srinivasan, J.A. Rivers, and H.-H.S. Lee. Safer: Stuck-at-fault error recovery for memories. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 115 –124, dec. 2010.

[81] Doe Hyun Yoon, Naveen Muralimanohar, Jichuan Chang, Parthasarathy Ranganathan, Norman Jouppi, and Mattan Erez. Free-p: A practical end-to-end nonvolatile memory protection mechanism. *Micro, IEEE*, 32(3):79 –87, may-june 2012.

[82] Rami Melhem, Rakan Maddah, and Sangyeun Cho. Rdis: A recursively defined invertible set scheme to tolerate multiple stuck-at faults in resistive memory. In *Proceedings of the 42nd IEEE/IFIP International Conference on Dependable System and Networks (DSN)*, jun. 2012.

[83] Lei Jiang, Youtao Zhang, and Jun Yang. Enhancing phase change memory lifetime through fine-grained current regulation and voltage upscaling. In *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, ISLPED '11, pages 127–132, Piscataway, NJ, USA, 2011. IEEE Press.

[84] Moinuddin K. Qureshi. Pay-as-you-go: low-overhead hard-error correction for phase change memories. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 318–328, New York, NY, USA, 2011. ACM.

[85] Andrew Hay, Karin Strauss, Timothy Sherwood, Gabriel H. Loh, and Doug Burger. Preventing pcm banks from seizing too much power. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 186–195, New York, NY, USA, 2011. ACM.

[86] M.K. Qureshi, M.M. Franceschini, and L.A. Lastras-Montano. Improving read performance of phase change memories via write cancellation and write pausing. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –11, jan. 2010.

[87] The Standard Performance Evaluation Corporation. The spec cpu 2006 benchmark suite. http://www.specbench.org.

[88] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prulovic., L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. Sesc simulator. http://sesc.sourceforge.net.

[89] Marek Olszewski, Qin Zhao, David Koh, Jason Ansel, and Saman Amarasinghe. Aikido: accelerating shared data dynamic analyses. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 173–184, New York, NY, USA, 2012. ACM.

[90] Yazhi Huang, Tiantian Liu, and C.J. Xue. Register allocation for write activity minimization on non-volatile main memory. In *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, pages 129 –134, jan. 2011.