Memory Architectures for Exaflop Computing Systems

A dissertation presented by

Milan Pavlović
to The Department of Computer Architecture

in fulfillments of the requirements
for the degree of Doctor of Philosophy
in the subject of Computer Science

Advisor — Alex Ramirez
Co-advisor — Petar Radojković

Universitat Politècnica de Catalunya
Barcelona, Spain
December, 2015

To my parents / Mojim roditeljima

Abstract

Most computing systems are heavily dependent on their main memories, as their primary storage, or as an intermediate cache for slower storage systems (HDDs). The capacity of memory systems, as well as their performance, have a direct impact on overall computing capabilities of the system, and are also major contributors to its initial and operating costs.

Dynamic Random Access Memory (DRAM) technology has been dominating the main memory landscape since its beginnings in 1970s until today. However, due to DRAM's inherent limitations, its steady rate of development has saturated over the past decade, creating a disparity between CPU and main memory performance, known as the *memory wall*.

Modern parallel architectures, such as High-Performance Computing (HPC) clusters and manycore solutions, create even more stress on their memory systems. It is not trivial to estimate memory requirements that these systems will have in the future, and if DRAM technology would be able to meet them, or we would need to look for a novel memory solution.

This thesis attempts to give insight in the most important technological challenges that future memory systems need to address, in order to meet the ever growing requirements of users and their applications, in manycore and HPC context. We try to describe the limitations of DRAM, as the dominant technology in today's main memory systems, that may impede performance or increase cost of future systems. We discuss some of the emerging memory technologies, and by comparing them with DRAM, we try to estimate their potential usage in future memory systems. The thesis evaluates the requirements of manycore scientific applications, in terms of memory bandwidth and footprint, and estimates how these requirements may change in the future. With this evaulation in mind, we propose a hybrid memory solution that employs DRAM and PCM, as well as several page placement and page migration policies, to bridge the gap between fast and small DRAM and larger but slower non-volatile memory.

As the aforementioned evaluations required custom software solutions, we present tools we produced over the course of this PhD, which continue to be used in Heterogeneous Computer Architectures group in Barcelona Supercomputing Center. First, Limpio — a LIghtweight MPI instrumentatiOn framework, that provides an interface for low-overhead instrumentation and profiling of MPI applications with user-defined routines. Second, MemTraceMPI, a Valgrind tool, used to produce memory access traces of MPI applications, with several innovative concepts included (filter-cache, iteration tracing, compressed trace files).

Acknowledgments

The work presented in this thesis was done in Barcelona Supercomputing Center (BSC), Barcelona, Spain. BSC provided financial support and access to resources that were indispensable for our research. Our work was also supported by the Ministry of Science and Innovation of Spain (contract number TIN-2007-60625 and TIN2012-34557), Generalitat de Catalunya under grant 2014-SGR-1261 and 2014-SGR-1051, HiPEAC European Network of Excellence on High Performance and Embedded Architecture and Compilation, the TERAFLUX project (ICT-FP7-249013), Mont-Blanc project (European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 288777), and the grant SEV-2011-0067 of Severo Ochoa Program awarded by the Spanish Government.

List of publications

- **Milan Pavlovic**, Milan Radulovic, Alex Ramirez and Petar Radojkovic. Limpio — Lightweight MPI instrumentatiOn. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (ICPC)*, pages 303–306, May 2015.
- **Milan Pavlovic**, Nikola Puzovic and Alex Ramirez. Data placement in HPC architectures with heterogeneous off-chip memory. In *Proceedings of the 2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 193–200, Oct 2013.
- **Milan Pavlovic**, Yoav Etsion and Alex Ramirez. On the Memory System Requirements of Future Scientific Applications: Four Case-Studies. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 159–170, Nov, 2011
- **Milan Pavlovic**, Yoav Etsion and Alex Ramirez. Can Manycores Support the Memory Requirements of Scientific Applications? In *Proceedings of the 2010 1st Workshop on Applications for Multi- and Many-Core Processors (A4MMC)*, pages 65–76, Jun, 2010.
- **Milan Pavlovic** and Alex Ramirez. Power Management on Off-Chip DRAM. In *Proceedings of the 2009 Advanced Computer Architecture and Compilation for Embedded Systems (ACACES)*, Jul, 2009.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computing systems play significant role in every modern home or business, and they constantly improve their capabilities to meet growing requirements of users and their applications. Such systems, in any of their forms, either desktop, laptop, embedded, or servers, rely on memory systems as their primary storage, or as an intermediate cache for slower storage systems (HDDs). The capacity of memory systems, as well as their performance, have a direct impact on overall computing capabilities of the system, and are also major contributors to its initial and operating costs.

As it is the case with most of the components of any modern computer, the elements of a memory subsystem have seen quite a few technological changes throughout its development history. Although memory systems typically employ several different technologies, depending on overall system requirements or the role of a particular memory component, any technology they are built upon has both cost and physical limitations if we try to scale it down or push to some performance limit. The researchers always have to evaluate if the costs of improving or minimizing the existing technology become higher than the benefits it may bring. Whenever such situation happens, a fresh idea and an innovative technology is needed in order to answer the future requirements of the industry.

Dynamic Random Access Memory (DRAM) technology has been dominating the main memory landscape of most computing systems since its beginning in 1970s until today. Its relatively low cost per unit of stored data and high access speed have satisfied the most typical requirements of computing systems. The first DRAM devices had 1,024 bits of storage, and are known as 1Kb DRAMs. The technology improvements have secured the increase in storage capacity by a factor of 4 every three years, which was in line with the advances in CPUs. However, after the introduction of the 128Mb product in year 2000, a 2× increment became the rule, which, along with the uneven improvements in memory speed, created a disparity between CPU and main memory, known as the *memory wall* [87].

The term *High-Performance Computing (HPC)* is typically used for the practice of aggregating computational power of a large number of CPUs, in order to deliver performance that is far superior than any desktop workstation, for solving complex problems in science, engineering or business. It is often used interchangeably with the term *supercomputing*. Because of their important role in computational science, supercomputers are today considered the “third pillar” of science, behind theory and experimentation. They find their widest usage in computationally intensive tasks in various fields, such as quantum mechanics, weather forecasting, climate research, oil and gas exploration, molecular modeling, as well as complex physical simulations. In these disciplines, HPC can provide insights that would otherwise be impossible to obtain, by serving as a platform for simulations that replace expensive and sometimes dangerous experiments. The computational power of an HPC system, therefore, directly shapes the scope of scientific research, and the level of details of the simulated experiments. It is clear that creating a powerful HPC ecosystem is a very important task for any science-oriented society.

Computational performance of a supercomputer is expressed in floating point operations per second (FLOPS). From 1960s, when early supercomputers performed in the range of 1 MFLOPS, over 1980s when first 1 GFLOPS system is created, in 2008 we have reached a 1 PFLOPS barrier. The TOP500 project maintains a list of the 500 most powerful supercomputing systems in the world, along with their details. Currently, the first place on TOP500 list holds Tianhe-2 from National Supercomputing Center in Guangzhou, China, built in 2013, with peak performance of over 54 PFLOPS. Optimistic estimates predict that the first Exaflop system (10^{18} floating point operations per second) will be built by the end of this decade.

However, in order to advance to an Exaflop system, we must solve numerous scientific and technological challenges, such as reducing power requirements, coping with run-time errors, exploiting massive parallelism, and many more. Otherwise, the initial and operational cost of such a system may far outweigh its potential benefits. One of the most important design decisions in any HPC system is the architecture of the main memory system. As the memory wall becomes larger with time, many researchers see it as a major obstacle for reaching 1 EFLOPS milestone in supercomputing.

This thesis attempts to give insight in the most important technological challenges that future memory systems need to address, in order to meet the ever growing requirements of users and their applications, in manycore and HPC context. We try to describe the limitations of DRAM, as the dominant technology in today's main memory systems, that may impede performance or increase cost of future systems. We discuss some of the emerging memory technologies, and by comparing them with DRAM, we try to estimate their potential usage in future memory systems. The thesis evaluates the requirements of manycore scientific applications, in terms of memory bandwidth and footprint, and estimates how these requirements may change in the future. With this evaulation in mind, we propose a hybrid memory solution that employs DRAM and PCM, as well as several page placement and page migration policies, to bridge the gap between fast and small DRAM and larger but slower non-volatile memory.

The contributions of this thesis are the following:

- Evaluation of the memory system requirements of scientific HPC applications, running on MareNostrum supercomputer at Barcelona Supercomputing Center, and characterization of the memory performance requirements of future manycore architectures. Our analysis covers most important memory-related metrics, such as memory footprint and memory bandwidth, and reveals memory intensive segments in each application by examining their CPI stacks. It also quantifies the impact of the memory system on arithmetic performance of the applications. As a result, we conclude that the limitations in DRAM scalability will not have negative effects on manycore systems in the next several years.
- Proposal of an architecture with a hybrid memory design that places two technologically different memory modules in a flat address space. On such system, we evaluate several HPC workloads against different data placement

and migration policies, compare their performance by means of execution time and the number of non-volatile memory writes, and consider how it can be applied to the future HPC architectures. Our results show that the hybrid memory system with dynamic page migration and limited DRAM capacity, can achieve performance that is comparable to a hypothetical, hard to implement, DRAM-only system.

- Design and implementation of Limpio, a framework for profiling of MPI applications. Limpio overrides standard MPI functions, and executes instrumentation routines before and after the selected MPI calls. Users themselves can write and customize the instrumentation routines to fit the requirements of the analysis. Limpio can invoke external application profiling tools, and can switch between various tools in a single execution. It can also generate application traces of timestamped events that can be visualized by general-purpose visualization tools or libraries. Limpio is regularly used in Barcelona Supercomputing Center for instrumentation of large-scale HPC applications.

The thesis is organized as follows:

Chapter 2 gives a brief overview of DRAM as the dominant technology in today's main memory systems. It covers the fundamentals of DRAM, the advances that this technology experienced over time, limitations in its scalability, and its specific role in HPC. Furthermore, it presents a survey of most promising emerging memory technologies that have the opportunity to replace DRAM in future main memory systems. We compare them with DRAM, and discuss their advantages, disadvantages and their potential usage in memory systems. We also give closer insight in the opportunities that non-volatile memory technologies could bring to HPC.

Chapter 3 makes an evaluation of memory requirements of several scientific HPC applications, in terms of bandwidth and footprint. In order to estimate whether DRAM technology would be able to meet the needs of future manycores, we make a projection of these metrics for the next-generation systems.

Chapter 4 proposes a hybrid memory architecture, with DRAM and PCM, and evaluates different page placement and page migration policies, for balancing performance and lifetime of the memory system. The evaluation is performed using HPC workloads, as the proposed hybrid design can be employed in HPC systems.

Chapter 5 presents a tool that is designed and developed over the course of this PhD, and continues to be used in Heterogeneous Computer Architectures group in Barcelona Supercomputing Center. Limpio — a LIghtweight MPI instrumentatiOn framework, that provides an interface for low-overhead instrumentation and profiling of MPI applications with user-defined routines.

Chapter 6 summarizes the conclusions from all the contributions presented in this thesis.

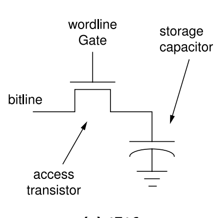
Chapter 2

Background

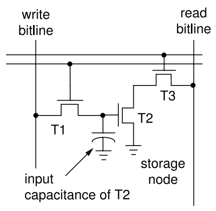
2.1 DRAM

DRAM (Dynamic Random Access Memory) is the most commonly used memory technology for main memories of modern computers. It earned its popularity with relatively simple production process, low cost per unit of storage, performance that could match the requirements of CPUs, and high reliability. Equally important, advances in semicondutor fabrication, that, with the increase in transistor density, fueled scaling of microprocessors over the last several decades, could also be successfully applied to DRAM. This brought more bandwidth and more capacity with each new DRAM generation.

In this section we will take a closer look at the basics of DRAM technology, and explore the limitations that slowed down its evolution and created a disparity between CPU and main memory, known as the *memory wall* [87]. We will also explore the role of DRAM in HPC, and particular memory demands in modern supercomputing.



(a) 1T1C



(b) 3T1C

Figure 2.1: Structure of a DRAM cell

2.1.1 DRAM fundamentals

The main building block of any semiconductor memory is a *cell*, where we can store one bit. Depending on the technology employed, one cell can contain different number of components in various configurations, and exhibit different characteristics to the memory consumer. For example, typical SRAM cell is comprised of four or six transistors. Early DRAM cell employed three transistor, while the modern DRAM cell consists of one transistor and one capacitor. One bit of information is stored as a charge, either in a powered transistor, or in a capacitor.

2.1.1.1 DRAM cell and array structure

For storing one bit of information, modern DRAM devices utilize a cell structure that comprises one transistor and one capacitor (1T1C). A circuit diagram of such cell is shown on Figure 2.1a. A cell is switched on by applying voltage on the gate of the access transistor (wordline). After that, the voltage that represents the bit that needs to be stored, from the bitline starts to charge the storage capacitor. When the storage capacitor is charged, it can hold the stored charge after the access transistor is switched off by removing the voltage from the wordline. However, due to the capacitor leakage, stored charge can be retained only for a limited time. For this reason, all DRAM cells must periodically be refreshed, in a process where stored data is read and written back. Following subsections will explain in more details the characteristics of a storage capacitor, and the process of DRAM refresh.

Reading the data from 1T1C DRAM cell discharges the storage capacitor and places the content of the cell onto a shared bitline. From there, the sense amplifiers

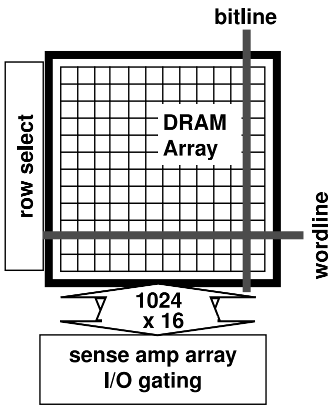


Figure 2.2: DRAM array

pick up the small voltage difference, and amplify it until a bit line is either on the lowest or on the highest voltage, representing logical '0' or '1'. In the process of reading, the original content of the storage capacitor is destroyed, so the information from the sense amplifiers needs to be written back to a memory cell.

First DRAM designs used a three-transistor and one-capacitor structure (3T1C), shown in Figure 2.1b, as its main storage cell. The transistors are separately used for read and write operations. When reading the data from the 3T1C DRAM cell the charge in a capacitor is preserved even after the stored bit is transferred to the bitline and the sense amplifiers, which makes the reading operation faster. However, this performance benefit was outweighed by a much smaller footprint of a 1T1C cell, so 3T1C structure is not used in modern DRAM devices.

A bank of storage cells is organized as a two-dimensional matrix, where each row of cells shares a common wordline, and cells on the same column are connected to the same bitline. A simplified illustration is shown in Figure 2.2. After a row address is decoded to select one row, all the storage capacitors that share the row's wordline discharge on their respective bitlines, and change their voltage levels. The sense amplifiers then resolve the resulting voltages into a digital value. One read operation moves the contents of the entire row to the sense amplifiers.

The capacitance of a storage capacitor is typically much smaller than the bitline capacitance. For this reason, sense amplifiers need to have a reference voltage in order to accurately detect the small changes on the bitline, and turn them to a logical '0' or '1'. It means that, instead of a single bitline, a pair of bitlines with matching capacitance is used for detecting the voltage difference.

2.1.1.2 Cell capacitor and DRAM refresh

In a manufacturing technology used in modern DRAMs, the storage cell capacitance is in the order of 30 fF. The DRAM access transistor's leakage current is in the order of 1 fA. A DRAM cell with these values of cell capacitance and leakage current, can hold electrical charge that is sufficient to resolve to the correct digital value for an limited period of time, ranging from hundreds of milliseconds to tens of seconds.

However, the intensity of the transistor leakage current is dependent on temperature, so the data retention times can be significantly different, not only from one cell to another at the same time and temperature, but also for the same DRAM cell at different moments. Memory systems must be designed so that none of the cells in a memory array loses its stored charge beacuse of the transistor leakage current. Therefore, all the cells in a DRAM device need to be refreshed before any single bit in the entire device loses its stored charge due to leakage. For modern DRAM devices, it means that the cells need to be refreshed once every 32 or 64 ms. Wherever DRAM design involves storage cells with low capacitance values or high leakage currents, the refresh interval needs to be reduced in order to guarantee proper data retention in cells in the device.

2.1.1.3 DIMM packaging

The early generations of computing systems allowed their users to expand the memory capacity by designing sockets where additional DRAM devices could be inserted. This was often a difficult and error prone process, as the pins on the devices might be bent or defective, faulty devices were difficult to identify, and it was physically possible to insert the device in the wrong orientation. The solution lied in creating memory modules that enclose a number of DRAM devices, and that have the standardized interface towards the rest of the architecture. Over the time, memory modules and their characteristics became the integral part of the memory system specification.

Various architectures of memory modules typically differ in the characteristics

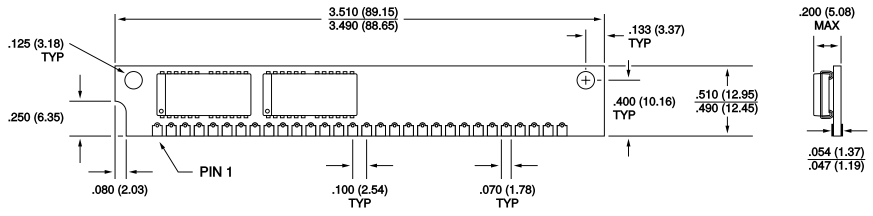


Figure 2.3: 30-pin SIMM diagram

of the DRAM devices they enclose, their count, the width of the data bus that interfaces the module with the rest of the system, and the signal frequency that they operate on. First standardized memory modules appeared in the end of 1980s, as a *Single In-line Memory Module (SIMM)*. SIMM provided 30-pin (Figure 2.3) or 72-pin (Figure 2.4) interface, with 8 or 9 (for 30-pin) and 32 to 36 (for 72-pin) signals lines on the data bus, along with the lines for power, ground, address, command and chip-select. The contacts on either side of the bottom of a SIMM module were electrically identical. The computing systems of that time typically employed four equivalent SIMMs to a wider memory interface, and used parity checking.

In the late 1990s, SIMMs were gradually replaced by *Dual In-line Memory Module (DIMM)*. DIMMs are physically larger than SIMMs, they provide wider data bus to the rest of the system, and the contacts on two sides of the bottom of a module are eletrically different, which allows for a denser routing of electrical signals. Moreover, larger real estate on a DIMM gave opportunity for a higher capacity devices, as well as room for some sophisticated control logic, that allowed buffering of address and control signals in *Registered DIMMs (RDIMMs)* (Figure 2.5). On the other hand, small-footprint personal computers and mobile devices, require a smaller alternative to a DIMM such as *Small Outline DIMM (SO-DIMM)* (Figure 2.6), while retaining throughput and capacity characteristics of traditional DIMM packagings.

However, a packaging process in a fixed format implies restricting available area and space for the storage elements of DRAM. Ease of installation and expandability that came with standard memory packaging is paid by a limited scalability, both in bandwidth and capacity. Memory architects are bound to produce innovative solutions for ever-growing memory demands, in a very constrained environment of a standard memory package. Otherwise, the production of non-standard memory modules would dramatically increase their cost and negatively impact the maintainability of a large scale computing system.

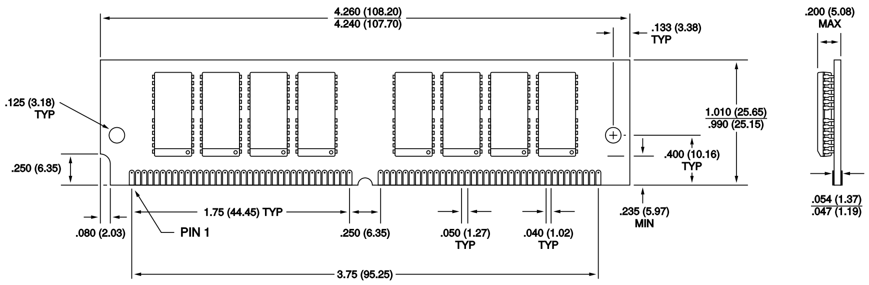


Figure 2.4: 72-pin SIMM diagram

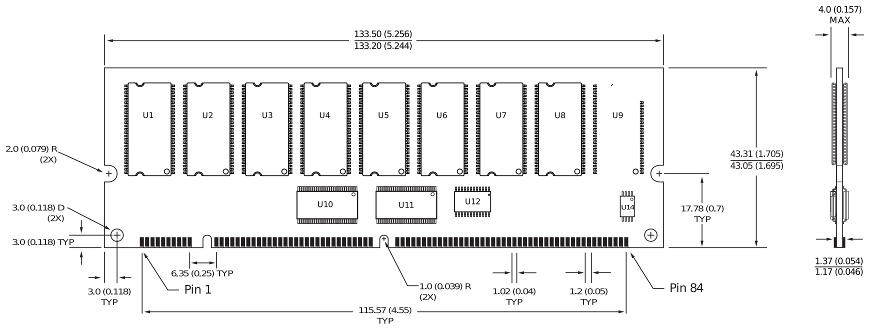


Figure 2.5: 168-pin Registered DIMM (RDIMM) diagram

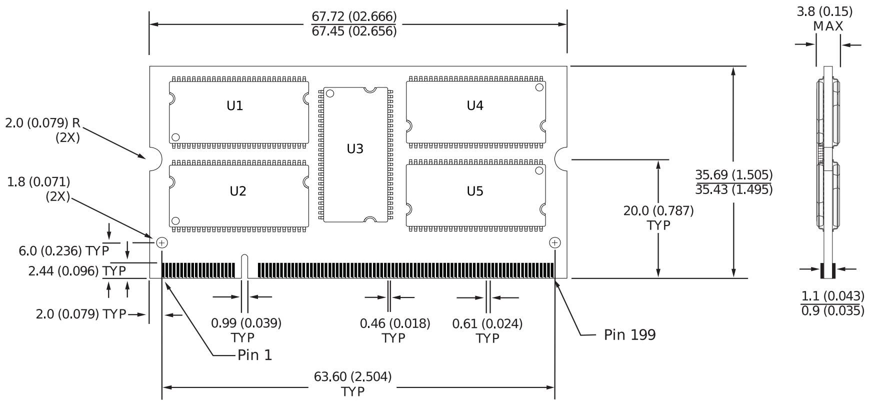


Figure 2.6: 200-pin Small Outline DIMM (SO-DIMM) diagram

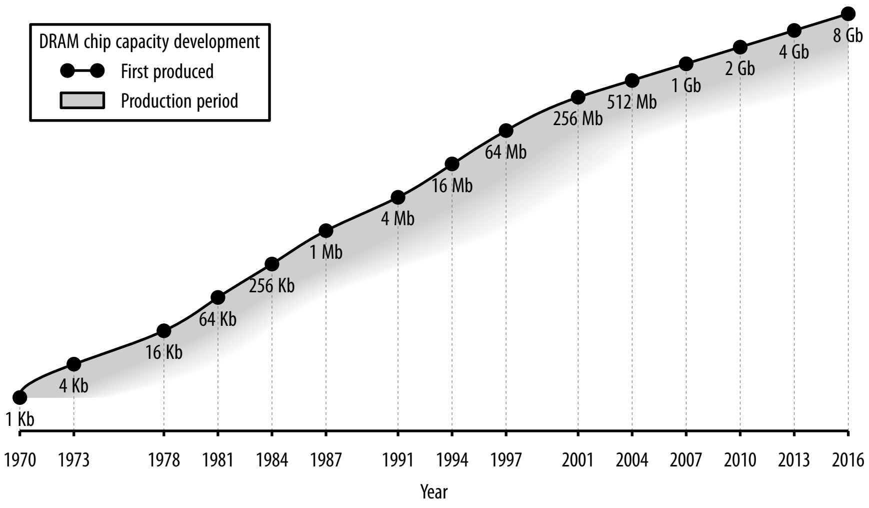


Figure 2.7: DRAM chip capacity development

2.1.2 DRAM development and history

In this section we will make an overview of the evolution of DRAM technology, since its beginnings in 1970. We try to analyze the main factors that led DRAM to quickly become the mainstream technology for main memories, and those that allowed it to remain in dominant position until today.

DRAM products are commonly recognized by their capacity, i.e. the number of bits per chip, also referred as "granularity". The first products had a capacity of 1,024 bits and were named 1K DRAMs. As briefly presented in Section 2.1.1.1, and illustrated on Figure 2.1b, first DRAM storage cell design consisted of three transistors, needed for controlling DRAM cell operations, and one capacitor, needed for storing charge representing bit of data.

Subsequent DRAM generations increased the amount of storage by a factor of 4. The development trend of DRAM since its beginnings is presented in Figure 2.7. Horizontal axis represents time, and vertical axis shows the capacity of a DRAM chip in logarithmic scale. The grey area on the figure represents the time period in which a particular DRAM chip was being produced.

In 1975, Gordon Moore reported that his prediction from 1965 of an annual doubling of the number of transistors on a chip remained true [57]. In fact, Moore also predicted that the doubling rate would slow down to a period less than two years. This turned to be remarkably correct in case of DRAMs, as their 4-fold increase in capacity occurred roughly every four years.

Moreover, Moore identified three main contributors for such a fast increase in the number of transistors per chip:

- Improvements in manufacturability leading to larger die sizes
- Innovation in cell layout for more efficiency
- Higher resolution lithography for increased density

When quantifying the relative contributions of each of these factors in case of DRAM, the industry produced a balance of these contributions with 50% due to lithography, 25% due to an increase in die size manufacturability, and 25% due to innovative reductions in cell size per bit. Similar distribution remained the same for the next few decades, fueling the predicted DRAM chip capacity increase.

This 4-fold increase in capacity every 3 years was sustained in the industry until the introduction of the 128 Mb product, after which a 2-fold increment became the rule. The main reason for this — lithography was left as the only contributor improving the DRAM chip capacity, as the increase in die size and the improvements in cell size came to their saturation.

The following section will explore the various obstacles that are alse considered as a major factor against DRAM scalability.

2.1.3 Obstacles for DRAM scalability

With the discussed fundamentals of DRAM we are able to get a better overview of the inherent technological obstacles that might slow down, or even completely stop DRAM scaling in future.

2.1.3.1 Bandwidth scaling

Today, the fastest configuration of the DDR3 DRAM technology achieves 1600 MT/s per 64-bit data channel and provides 12.8 GB/s per channel, while the sustained bandwidth is typically 20%—25% lower because of page and bank conflicts. Indicatively, modern architectures such as the Intel Nehalem-EX [43] employ 4 channels, whereas the IBM Power7 [34] employs twice as many DDR3 channels, peaking at 102.4 GB/s of data. Existing DRAM densities enable such configurations to directly connect a single chip to a few hundreds GBs of DRAM.

Achieving higher bandwidths with existing technology is feasible, but at a cost [31]. Increasing the bandwidth of the chip-to-memory bus typically requires either increasing memory channel frequencies, or its bit width. But increasing the frequencies hinders signal integrity, which implies shorter wires. The limited distance in turn, restricts the board area on which DIMMs can be placed, and thereby limits the number of DIMMs connected to a single CMP. On the other hand, increasing the number of channels (or channel width) requires a matching increase in the number of processor pins, which results in more expensive processors: more pins imply larger processors, which are more costly (fewer processors per wafer). In addition, operating multiple channels and DIMMs in parallel greatly increases the power consumption of the memory system.

Alternatively, emerging technologies such as 3D-stacking and multi-chip packaging can potentially overcome the bandwidth hurdle by placing multiple DRAM dies in a single package. It is estimated that the bandwidth between 3D-stacked DRAM layers will be 100× faster than off-chip bandwidth and can thereby reach the TB/s domain [85, 11, 52]. Moreover, such technologies effectively eliminate the pin-count and wire lengths problems. In contrast, these technologies offer a limited DRAM capacity, as they are limited in the number of DRAM components they can pack together due to thermal dissipation and constrainted physical area. This limitation is aggrevated when we account for the additional memory redundancy required to facilitate fault tolerance in integrated packaging technologies, as these do not allow to replace faulty DRAM components [33]. These restrictions, therefore, is expected to limit the overall on-chip DRAM capacity to a few dozens of MBs.

Relying to only one of the existing technologies thus means sacrificing either high bandwidth or high capacity. Importantly, the packaging limitations we discuss are impervious to the underlying memory technology. Even though we base our discussion on the characteristics of the prevalent DRAM technology, similar limitations apply to alternative emerging technologies.

2.1.3.2 Capacitor scaling

In order to scale down the area occupied by one memory cell, one needs to think about scaling down the area occupied by the cell's storage capacitor. On the other hand, the stored capacitance, typically proportional to the capacitor's size, needs to be sufficient to quickly enough set the sense amplifiers over the sensing threshold. It also needs to be high enough to reliably dominate over different sources of noise (leakage, switching disturbances, radiation-induced charge) on the bitline. 4Mb DRAM chips typically had their bitline capacitance in a range of 150–350 fF. These values remained roughly the same until today, because the gains in per-cell capacitance were mostly cancelled by increasing the number of cells per bitline. Therefore, the required storage capacitance is expected to remain from 30–40 fF, which becomes a real challenge in an area-constrained cell.

In order to maintain the capacitance of the memory cell's storage capacitor relative to the bitline capacitance, with the cell size reduction in successive DRAM generations, planar capacitors were replaced by trench and stacked capacitors in the mid-1980s by IBM, Texas Instruments, and Toshiba [7]. The three-dimensional structure of a trench capacitor enables decoupling the effective surface area of the capacitor from the area of the memory cell, which resulted in continuous scaling of the storage capacitor while maintaining its capacitance in the desired range.

For further scalability options, and higher capacitance per unit area yield, the manufacturers have introduced capacitor dielectrics with higher dielectric constants. Typically, DRAM storage capacitors up to the 0.15-μm generation have used the NO (nitrite-oxide) dielectric. After that Ta_2O_5 was introduced for the 0.12-μm generation, and further scaling to 0.1 μm required a material with a relative dielectric constant of over 20, such as barium strontium titanate (BSTO) [78].

With the minimum lithographic feature size of a memory cell scaled down, the series resistance of the storage capacitor can become a dominant factor in the total resistance. It can cause degradation in the transfer of charge between the bitline and the capacitor. Typically, the capacitor's series resistance should not be higher than 50 kΩ.

2.1.3.3 Refresh and power scaling

With increase in DRAM capacity, the time and energy spent in refresh operations become more significant. More memory cells need to be refreshed, and with the lower retention time of the storage capacitors, they need to get refreshed more often. Liu et al. [47] showed that with these constraints a hypothetical DRAM device of 64Gb would spend 46% of its time and 47% of energy for refresh operations. For comparison, modern 4Gb DRAM device spends 8% of the time and 15% of the energy on refresh.

However, due to fabrication inconsistencies, retention times of the storage capacitors may differ from one memory cell to another. Refresh time of a device is chosen conservatively, so that it is sufficient to guarantee the integrity of a cell with shortest retention time. As a consequence, most of the refresh operations are unnecessary for a vast majority of cells that can retain data for significantly longer time. This opens up the opportunity for mechanisms that keep track of *weak* cells that have shorter retention times [4, 39], in order to lower the refresh rate for the rest of the cells. Some solutions, like RADR [47], continuosly track for any changes in each cell's retention time and adapt its refresh rate. Others take a different approach, and either propose not using rows with weak cells at all, or mapping critical data to cells with longer retention times [79, 49].

That would bring significant benefits in both performance and energy consumption of a DRAM system.

2.2 Emerging memory technologies

As elaborated in the previous section, existing memory systems based on DRAM face inherent limitations that impede scaling of both memory bandwidth and capacity. Moreover, analyses of the modern HPC systems show that the main memory is one of the major contributors to the total energy consumption, and consequently to the operational cost. With these issues in mind, we have enough reasons to be concerned that DRAM would not be able to successfully meet all the requirements of future HPC systems and applications (we will more thoroughly analyze the requirements of future HPC applications in Chapter 3).

In the following sections we will define an ideal set of characteristics of the hypothetical memory technology that would supersede DRAM in HPC systems. Then, we will make a brief survey of the promising emerging memory technologies, with their main features and perspectives for future development. Finally, we will pay closer attention to the benefits that these new technologies bring specifically to HPC systems.

2.2.1 Superseding DRAM — the winning recipe

The ultimate memory technology that would replace any existing solution would have to be scalable to any capacity, to provide infinite bandwidth, without latency, and at no cost. As it is the case with most engineering problems, these idealistic requirements are not only unfeasible when considered separately, but also make opposing constraints among them.

- Ingredient #1 — non-volatility

  As power consumption of DRAM devices starts to dominate in overall power breakdown of the system, much of which comes from a stand-by or refresh power, non-volatility becomes one of the most desirable characteristics of future memory systems. In that sense, HPC could make further use of non-volatile memories by performing check-pointing without incurring any performance overhead.

- Ingredient #2 — cell density

  As explained previously, the area occupied by a single DRAM cell is largely determined by the size of the storage capacitor. Emerging memory technologies make use of different storage mechanisms which have the possibility of scaling to much smaller sizes. That increases the cell density, and allows larger capacity devices on the same chip area.

- Ingredient #3 — endurance

  Many non-volatile technologies exhibit limited endurance — each cell can be written limited number of times before the storage material becomes unreliable. Despite many wear-leveling mechanisms that are used to prolong the lifetime of a write-limited device, it is important to have a technology with decent endurance, especially in a memory intensive HPC environment.

- Ingredient #4 — read and write latency

  Latencies of read and write operations in DRAM are comparable. Due to storage material constraints, some non-volatile memories often have much higher write latency. Moreover, non-volatile memories generally have higher read and write latencies when compared to DRAM.

2.2.2 Emerging memory technologies — overview

Existing memory hierarchies rely upon absolute performance differences among constituent memory technologies, such that each layer in the hierarchy employs a memory technology that outperforms the level below it in every performance aspect, yet providing better power consumption and density characteristics than the layers above it. This *strict ordering* of memory technologies is, however, changing, as emerging memory technologies often have a combination of characteristics that does not directly imply its position in memory hierarchy.

This emerging complexity is manifested in the Table 2.1, which lists various performance and design characteristics: read/write latency, write endurance, storage density, power consumption, etc., for both traditional and emerging memory technologies.

2.2.3 PCM (Phase Change Memory)

PCM is regarded as an emerging technology, although the phase changing concept has its roots in 1962, when Pearson et al. described the switching mechanism present in As-Te glasses. However, after several more important research papers on that topic, the immature material quality and high power consumption of the device made further commercialization of PCM unfeasible. The research resumed in late 1990s, when the first modern PCM design was described [77], and the test chips fabricated in 2000 [25].

PCM stores data using a phase-change material that can be in one of two physical states: crystalline or amorphous. As a consequence, it provides superior density relative to DRAM, and it can be used to provide a much higher capacity for the memory system than DRAM can within the same budget. Furthermore, a PCM

	Traditional technologies				Emerging technologies			
	DRAM	**SRAM**	**Flash**		**FeRAM**	**MRAM**	**PCRAM**	**Memristor**
			NOR	**NAND**				
Knowledge level	mature		advanced		product		advanced	beginning
Cell elements	1T1C	6T	1T		1T1C	1T1R	1T1R	1M
Half pitch (F) (nm)	50	65	90	90	180	130	65	3-10
Smallest cell area (F^2)	6	140	10	5	22	45	16	4
Read time (ns)	<1	<0.3	<10	<50	<45	<20	<60	<50
Write/erase time (ns)	<0.5	<0.3	10^5	10^6	10	20	60	<250
Retention time (years)	secs	N/A	>10	>10	>10	>10	>10	>10
Write op. voltage (V)	2.5	1	12	15	0.9–3.3	1.5	3	<3
Read op. voltage (V)	1.8	1	2	2	0.9–3.3	1.5	3	<3
Write endurance	10^{16}	10^{16}	10^5	10^5	10^{14}	10^{16}	10^9	10^{15}
Write energy (fJ/bit)	5	0.7	10	10	30	1.5×10^5	6×10^3	<50
Density (Gbit/cm^2)	6.67	0.17	1.23	2.47	0.14	0.13	1.48	250
Voltage scaling	fairly scalable				yes	no	poor	promising
Highly scalable	technological barriers				poor		promising	

Table 2.1: Traditional and emerging memory technologies comparison

cell can be in different degrees of crystallization, thereby enabling more than one bit to be stored in each cell, further improving data density in PCM [9]. It can also be expected that PCM is going to scale better than DRAM [23]

On the other hand, PCM is much slower than DRAM, and it would make a memory system comprising exclusively of PCM, to have much increased memory access latency; thereby, adversely impacting system performance. More, PCM devices are likely to sustain signifiantly reduced number of writes compared to DRAM, therefore the write traffic to these devices must be reduced. Otherwise, the short lifetime may significantly limit the usefulness of PCM for commercial systems.

Qureshi et al. explore *Phase Change Memory (PCM)* as a way to bridge the latency access gap between DRAM and persistent storage, and create large capacity main memory system that would avoid costly hard disk accesses, and still provide sufficient bandwidth to the processors [68].

2.2.4 STT-MRAM

Research exploring the magneto-resistance caused by the spin polarized current can be tracked back in the ’90s [19][74][35]. Although, significant scientific efforts of optimizing and applying this phenomenon to create a novel non-volatile memory is a relatively new approach. Only around ten years ago, in 2005, Hosomi *et al.* [28] presented a non-volatile memory utilizing spin transfer torque magnetization switching for the first time. In the following years, there has been a notable dedication of academic scientists and memory manufactures researching this novel non-volatile memory technology.

The storage and programmability of STT-MRAM revolves around a Magnetic Tunneling Junction (MTJ). An MTJ is constituted by a thin tunneling dielectric being sandwiched between two ferro-magnetic layers. One of the layers has a fixed magnetization while the other layer’s magnetization can be flipped. As Figure 2.8(a) and (b) depict, if both of the magnetic layers have the same polarity, the MTJ exerts low resistance therefore representing a logical “0”; in case of opposite polarity of the magnetic layers, the MTJ has a high resistance and represents a logical “1”. In order to read a value stored in an MTJ, a low current is applied to it. The current senses the MTJ’s resistance state in order to determine the data stored in it. Likewise, a new value can be written to the MTJ through flipping the polarity of its free magnetic layer by passing a large amount of current through it [88].

Figure 2.8(c) illustrates a simplified STT-MRAM array [37][38][62] based on 1T-1MTJ cell. The cells are organized into rows and columns, similar to the conventional DRAM modules. The main difference is that, instead of the capacitor used in DRAM, one bit of data is stored in the MTJ. In this design, the word lines $\mathrm{WL}_{1..m}$ activate particular rows of the cell array, while the bit lines $\mathrm{BL}_{1..n}$ are used to perform read or write operation to the corresponding MTJs. The current required for these operations is driven by the source lines $\mathrm{SL}_{1..m}$. The main drawback of the 1T-1MTJ STT-MRAM cell is a long and energy-hungry write operation. This motivated further research on the STT-MRAM cell design, and recently the manufacturers revealed advanced 2T-2MTJ, 3T-2MTJ and 4T-2MTJ cells in a pursuit to mitigate this drawback [2][61][59].

STT-MRAM can be used to build byte-addressable memory devices with pin structure compatible to the conventional DRAM chips [37]. Therefore, existing DRAM modules can be seamlessly replaced with STT-MRAM modules, without

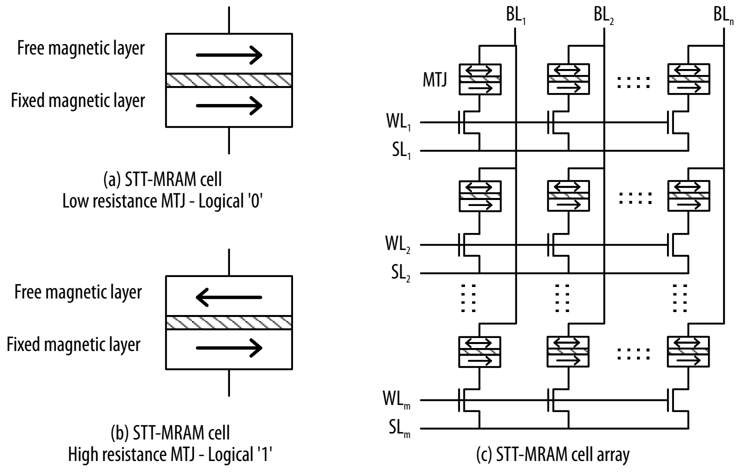


Figure 2.8: STT-MRAM cell and cell-array

requiring any modification in the rest of the system architecture. This may suggest an easier incorporation of STT-MRAM in the existing systems.

2.2.4.1 STT-MRAM design challenges and techniques

Process variations coming from magnetic devices and CMOS technology are imposing many important design challenges to STT-MRAM at a nanometer scale.

The variations in doping and geometry, originating from CMOS process instability, can significantly influence the resistance of transistors in STT-MRAM array, and thus impact its design. The NMOS selection transistor in an STT-MRAM cell works in the linear region when writing logical '1'. A seemingly insignificant variation of the NMOS device parameter or a small inaccuracy of gate-to-source voltage may cause a big difference in the switching current through the MTJ.

The variations in the process parameters of MTJ, as the key element of the information storage in STT-MRAM, directly influence the stability of the STT-MRAM cell. These variations can be caused by quantum mechanical tunneling of the oxide thickness resulting in a large spread in resistive states of the MTJ.

In order to mitigate large number of writes, which according to Zhou et al [90] contribute to over 70% of STT-MRAM dynamic energy, they proposed a read-before-write scheme that can identify unnecessary writes to memory. To do this, each column needs to be extended by a set of decision circuitry which compares the existing bit and the bit to be written, and decide if it is possible to avoid executing the write. The drawback of this method is further slowdown of write operations, as they need to be preceded by a read.

The number of writes to STT-MRAM cells can further be reduced by employing data inverting. Before writing new data to a block of cells, a specialized circuitry would calculate the Hamming distance (HD) between the existing data and the data to be written. Whenever calculated HD is greater than half of the block size, the data is inverted before storing, and a bit for determining the inversion status of the block is set.

The unbalanced write patterns to a memory system demand that the wear-leveling techniques need to be employed, in order to increase the endurance of the memory system. The most common wear-leveling techniques use special tables to store the number of accesses to each block of cells. Periodically, the most accessed blocks are logically swapped with the least accessed block, which increases storage overhead and latency, but improves on the endurance of the system.

2.2.5 Resistive Random Access Memory (R-RAM)

Resistive Random Access Memory (R-RAM), in its broad sense, represents any random access memory that stores data as a difference in resistance between two states that represent logic '0' or '1'. Generally, a single R-RAM cell has a metal-insulator-metal (MIM) structure, where the difference in resistance of the insulator layer, between high-resistance state (HRS) and low-resistance state (LRS), can be used to store one bit of data. There are several materials that can be used as an insulator layer for this purpose. So far, R-RAM has taken advantage of its high scalability, low read and write energy operations, and simple production, to serve as a replacement for more traditional data storage technologies (HDD and flash memory). Recent advances in cell structure made improvements in read and write latency, so these devices can see their potential use as the main memory. Similarly to PCM, each R-RAM cell can extend its data storage capacity by increasing the number of resistance states, and that way store two or four bits instead of only one.

2.2.5.1 R-RAM types

There are two basic types of R-RAM, which differ in physical and chemical processes in order to achieve changes in resistance of the insulator layer: *electrochemical metallization devices* and *valence change devices*.

Electrochemical metallization devices make use of *filaments*, conductive paths formed inside the insulator that connect the two terminals. Thypically, they occupy only a tiny fraction of the insulator layer, but are enough to change its resistivity. Filament is not a conductive material, like metal or semiconductor. Instead, it is a modification in a chemical structure in the dissolving ions within the insulator layer. It can be created or removed from the material by applying continuous voltage on the terminals.

A MIM structure typically consists of one inert (Au, Pt...) and one active (Cu, Al, Ni...) electrode, with the insulator between them. A continuous voltage applied on the electrodes moves ions from the active metal into the insulator space, which then interact with the dissolving ions in the insulator, to form a compound with conductive characteristics. When the insulator contains sufficient amount of conductive compounds, filaments can be formed from the active to the inert electrode. After that, they can conduct current across the insulator layer, effectively switching the material to a low-resistance state. Number of filaments in the material determines the density of the current. Resetting the material to the high-resistance state is done by applying opposite voltage, when the conductive compounds are dissolved back to their original state, thus removing filaments from the insulator, and that way increasing its resistivity.

On the other hand, a valence change device exploits only physical properties of the insulator, and uses charge trapping mechanism to add pseudo-states between the valence and the conduction band. Because of these pseudo-states the electrons are more likely to move to the conduction band. The charges migrate into the insulator by Fowler-Nordheim tunneling, are are then trapped by the defects in the insulator. That way, the Shotkky barrier at the metal-insulator junction is lowered, and the material can exhibit conductive properties at lower voltage levels, effectively representing different logic value.

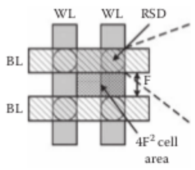
2.2.5.2 R-RAM array and cell structure

A straightforward and the most common design of a R-RAM array is the crossbar structure, used for the first time in a telecommunication signal routing system. Conceptually, it comprises two sets of wires with switches at their intersection points, so the signal routing is achieved by selecting a correct switch. In memory devices, such as R-RAM, the two sets of wires represent word lines (WL) and bit lines (BL), and the storage material is placed at cross points. That way, accesing a memory cell in a crossbar array can be done by supplying a read voltage to its WL and sensing the resulting current on its BL. If the sensed current is above the predefined threshold, the storage material exhibits a low-resistance state, and a logical '1' is read. Conversely, if the sensed current is below the threshold, the storage material is in a high-resistance state, and a logical '0' is read. To parallelize read operation on the whole WL, we can read the resulting current on all BLs simultaneously.

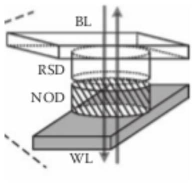
A single R-RAM cell can exploit its high R_{off}/R_{on} ratio to be designed without any switching device for increased array density — the single element connecting WL and BL is the storage material. A combination of a crossbar structure with a cell without a switching device is illustrated in Figure 2.9. Due to litography limitations, each side of the memory cell can be as low as $2F$ long, giving a single cell area of $4F^2$. Apart from such a high density, this design is characterized by a simple fabrication process, and the ability to construct multiple memory layers vertically, making 3D stacking a plausible solution.

However, crossbar array without switching elements experiences a side-effect known as a *sneak-path*, where a low resistance of three or more elements in a series can lead to a false reading on BL. This effect is illustrated in Figure 2.10. Because of this, much higher voltage needs to be applied to the selected cell than to the other cells in a sneak path, to ensure valid readings. Sneak paths also directly influence memory array scalability, as they can degrade the sensing margin in large crossbar structures. Similarly, this requires R-RAM storage materials to have relatively high resistance in LRS while keeping the difference between HRS and LRS large. For example, LRS should have a value in the range of killoohms, and HRS in the order of megaohms [84, 46].

When better access control is required, an active (transistor) or passive (diode) switching device can be included in each cell. In both cases switches can significantly reduce the leakage current, and at least decrease, if not eliminate, the impact



(a) Vertical



(b) Perspective

Figure 2.9: Evolution of the memory footprint

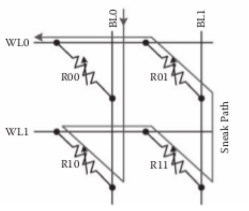


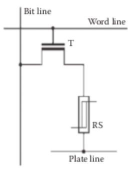
Figure 2.10: Sneak path

of sneak paths in an array. This makes the opportunity for creating larger R-RAM arrays, without strict limitations in resistance levels that cells without a switching device have. On the other hand, switching devices may increase the footprint of a memory cell, and negatively impact the density of the array.

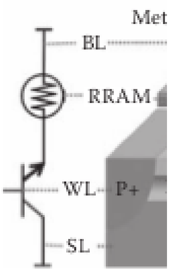
Figure 2.11a shows a design of a R-RAM cell using metal-oxide semiconductor field-effect transistor (MOSFET) as an active switching device. By isolating its gate from other terminals, we can use MOSFET transistor to switch the current on or off across the storage material. The memory cell area in this case is more than $10F^2$, which, compared to a cell without a switching device, is a significant increase. To make things worse, MOSFET layers cannot be integrated vertically, which prevents 3D technology from relieving problems with small density. Some advances in this aspect are expected from a thin-film transistor technology (TFT).

An improved solution can be achieved by replacing MOSFET with a bipolar junction transistor (BJT) (Figure 2.11b). By using vertical BJTs we can improve density in a single layer, and obtain $4F^2$ cell size [76]. However, stacking several layers in a 3D form remains a problem, because of the fabrication issues.

Instead of using a transistor as an active switching device, R-RAM cell can in-



(a) MOSFET



(b) BJT

Figure 2.11: MOSFET and BJT

clude a passive element, such as diode, non-ohmic device or chalcogenide material, integrated into a crossbar structure between the storage material and one of the terminals. Their non-linear electrical characteristics allow them to serve as a switch which is turned on only when the threshold voltage is exceeded. The alternatives between the aforementioned passive elements mainly differ in their mode of operation, and are used in either unipolar or bipolar R-RAM design, and usually have different switching threshold voltage. Besides that, they keep the single-cell footprint low (at $4F^2$), but because they never reach the ideal on/off characteristics, the effect of sneak paths is not fully eliminated. This constrains the array size to a certain extent, so the passive switching design is considered as a trade-off between active switching and the design without a switching device.

In order to make large R-RAM arrays without switching devices and sneak paths, Lee et al. [45] have proposed a complementary crossbar structure. In this design two R-RAM devices are connected together, where one has a role of a switch, and the other one of a memory cell. The resistance states of the two elements are complementary, that is, if one element is in LRS, the other one is in HRS. That way the total resistance of an unselected cell is kept high, and the sneak paths are eliminated. In order to read the value of a particular memory cell, one must first set its appropriate switch cell to a LRS, then read the total resistance, and finally reset the switch cell to its complementary state. Although this design can enable large R-RAM arrays, with a small footprint of a single cell, it suffers from slow reads and writes due to the multi-step operation process.

2.2.5.3 R-RAM opportunities and future

As shown in the previous sections R-RAM is a technology that has the potential to be scaled down to the nanometer level. Ho et al. [27] have shown a 9-nm R-RAM that can be used to build a crossbar array larger than 64×64, with a programming current in the order of milliamperes. Retention times and endurance have different values for various materials, but generally they are higher than 10 years and 10^6 number of writes.

2.2.6 Memristor

The traditional circuit theory recognizes three basic passive circuit elements — the resistor, the capacitor and the inductor. They have been the main building blocks of all electronic circuits that drive modern technology. Resistance, capacitance and inductance are used to define three equations which relate four fundamental circuit values — electric current (i), voltage (V), electrical charge (q) and magnetic flux (φ):

$$dV = R \times di \tag{2.1}$$

$$dq = C \times dV \tag{2.2}$$

$$d\varphi = L \times di \tag{2.3}$$

In 1971, Chua [14] set a theory where there exists a fourth basic circuit element, which he called *memristor*, and the value describing it — *memristance*. Memristance is, thus, defined as a relationship between magnetic flux and electrical charge:

$$d\varphi = M \times dq \tag{2.4}$$

The unit of memristance is ohm(Ω). However, unlike the resistor where the resistance is constant in time, and independent of applied voltage or current, memristance exhibits hysteretic behaviour dependent on the electric charge ($V(t) = M(q(t))I(t)$). Therefore, memristance can change its value by controlling the

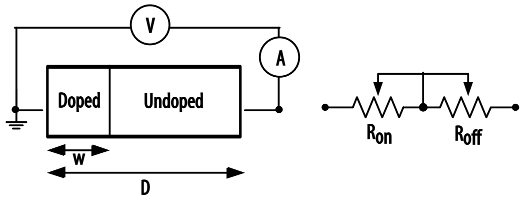


Figure 2.12: Structure of a memristor device, and its equivalent circuit model

amount of electric charge that passes through the device. Furthermore, the memristor is constant in time when no current is applied, which is the essence of non-volatility property.

Chua proved that memristor behavior cannot be synthesized using other three basic elements. However, his work described memristance only as a theoretical concept, without any evidence of its actual existance. In 2008, Williams et al. [75] claimed the actual discovery of memristor, by describing its structure, and confirming its behavior. Soon after that, various research teams, including Hewlett-Packard, SK Hynix and HR Laboratories, made their first experiments with a newly created element, enabling its application not only in memories, but also in computer logic and neuromorhic/neuroresistive architectures.

In Figure 2.12 we see the actual structure of a memristor device as a semiconductor thin film, placed between two metal contacts. The film comprises doped and undoped region of a total length D, and an internal state variable w, representing the length of the doped region. The two regions differ in their resistance — doped has lower resistance and undoped significantly higher. When voltage is applied to the device, the length w changes because of charged dopant drifting [89], effectively changing the device's total resistivity. When the doped region occupies the full length D, the device has the minimum resistivity of the value R_{on}. Similarly, if the undoped region fully extends, the device resistance reaches its maximum at a value of R_{off}. The mathematical model of a memristive device can, therefore, be described as:

$$R(w) = R_{on} \times \frac{w}{D} + R_{off} \times (1 - \frac{w}{D}) \tag{2.5}$$

In memory systems based on memristors data is stored using ions, instead of electrons, by defining the length of a doped region w relative to the length of the thin film D. When a charge is applied to a memory cell these ions are displaced a small distance, which causes a large difference in cell resistance. And because ions retain their position after the voltage is switched off, a memristor cell retains the stored information when the power is lost, giving it the non-volatility.

Although memristors are still in the research stage, with the commercial availability expected in 2018, it promises performance comparable to DRAM, with a price as low as Flash, and still demanding less energy for its operation than either of the aforementioned technologies. Most importantly, memristors show good scalability, not only by shrinking a cell to a $10nm$ scale, but also by stacking memristor grids in 3D structures, and by storing multiple bits per cell.

2.2.7 Opportunities in HPC for emerging memory technologies

2.2.7.1 DRAM refresh

One of the most obvious advantages that all non-volatile memory technologies have over DRAM, is absence of need for periodical refresh. In the first part of this chapter, we analyzed all the consequences of DRAM refresh, most-important being the degradation of system performance and increased energy consumption. With its specific constraints and requirements, it is important to underline the benefits we might see in HPC, once we remove DRAM refresh from the equation.

2.2.7.2 Check-pointing

A single HPC application process, during its lifetime, can unexpectedly fail and terminate. This can happen because of a software bug, or due to any transient or permanent failure of a hardware component in the system. As a result, the whole application may malfunction — the performance can degrade, the produced results may be incorrect, or the entire execution can terminate. Given that the re-execution of long-running HPC jobs can be costly and time-consuming, the reliability of the system is a very important requirement.

Large HPC systems provide fault tolerance by periodical checkpointing, where the current state of the system is taken as a snapshot, and saved to a persistent storage. If there happens a failure and the application terminates, it can be restarted from the last saved checkpoint, instead of restarting it from scratch [21].

However, in traditional HPC systems, check-pointing brings an additional cost, as it interferes with the regular execution, and stresses both I/O storage and interconnect network. It is estimated that between 15% and 45% of the operational time of current HPC systems is spent on check-pointing, restarting and partial recomputation of the work from the last checkpoint [16, 17]. As the size of the future HPC systems and applications increases, we can only expect that the time spent check-pointing would become a dominant component in total execution time. It is estimated that check-pointing time would go up to 65% of the total operational time of a 100,000-node cluster [22].

The systems that use one of the emerging memory technologies for the main memory, can exploit their non-volatility properties to avoid transferring large amounts of data through the network to a persistent storage. The memory system itself can safely store all the information needed for the recovery after the job re-execution. More, if we anticipate the situation where the whole server fails (due to the power failure), check-point data can be stored in a non-volatile memory of a nearby server. That way, we could increase the robustness of a system, without employing interconnect network and persistent storage, and avoid the overhead that this brings.

Reducing the overhead of check-pointing would also make room for advanced techniques in job scheduling. In order to make better utilization of the system, job scheduler can perform a check-point and interrupt the execution of a long-running low-priority job, to give the processing resources to a high-priority job. Once the prioritized jobs are completed, the original jobs can be restarted. This could also give benefits in system administration, as a system shutdown would not require emptying the job queue or terminating long-running jobs.

2.2.7.3 Memory errors

The reliability of a single DRAM cell directly determines the reliability of the memory system. For future HPC systems that require more memory, DRAM reliability becomes a very important issue.

First, if the probability of a single cell failure is constant, the overall memory reliability decreases with the consistent growth of its capacity [29]. Second, if DRAM cells keep getting smaller, they retain less amount of charge, which makes them increasingly susceptible to any external disturbance or data corruption [40]. Also, the distance between DRAM elements is already so small that electromagnetic coupling causes undesirable interactions between the adjacent cells. Finally, as the technology scales-down, the relative variation in process technology increases, leading to a higher number of cells that are exceptionally vulnerable to errors. Currently, memory errors are addressed by using advanced *chip-kill* error correction mechanism, which significantly improves reliability, but it also introduces notable power and storage overheads [32].

Non-volatile technologies mitigate the transient faults (caused by magnetic or electrical interference), that account for a significant portion of the overall memory faults. Therefore, they would not only improve the reliability of the system, but also reduce the complexity and overheads of the traditional error correction mechanisms.

On the other hand, neither of the emerging memory technologies is failure-safe. Their failures (stochastic in nature), can be efficiently controlled with proper ECC mechanisms [6]. Recent study of Pajouhi et al. [63] analyzes interaction between device parameters, bit-cell level parameters and different ECCs to optimize the robustness and energy-efficiency of STT-MRAM cache. An interesting follow up would be to extend this work on STT-MRAM main memory.

2.2.8 Emerging memory technologies — summary

The previous sections described several emerging memory technologies, and their characteristics in comparison with DRAM. Some of them are still in their research phase, or with existing technical obstacles needed to overcome before entering the mass production stage.

PCM, as the most mature of the emerging memory technologies, needs to focus on improving access times and endurance. From the architectural perspective there already exist several solutions that attempt to mitigate these shortcomings of PCM. Some of them are described in Chapter 4.

The applications for an STT-MRAM also require quick reads, which becomes a challenge because of small sense margins. Thermal fluctuations in manufacturing process can cause high write errors, which can impede STT-MRAM's potential, if left unresolved.

The endurance of R-RAM has been improved significantly, with enabling 10^{10} write cycles. One of the few things left to be addressed is their chip-level integration, which demands improvements in device uniformity and the design of a peripheral circuit.

Memristor is the newest of the emerging memory technologies, but in a short time has gained much attention not only because of its memory-related characteristics, but also for its application in logic circuits and neuromorhic/neuroresistive architectures. However, Hewlett-Packard research team that led the innovation regarding memristor, has recently announced the abandonment of memristors in their 'The Machine' project, aimed to cope with the flood of data from Internet of Things.

The successor of DRAM technology is, therefore, difficult to be predicted. Uncertainties in new technology development make predictions based on trend extrapolations highly unreliable.

Chapter 3

Memory requirements of HPC applications

In this chapter, we observe and characterize the memory behaviour, and specifically memory footprint, memory bandwidth and cache effectiveness, of several well-known parallel scientific applications running on a large processor cluster. Based on the analysis of their instrumented execution, we project some performance requirements from future memory systems serving large-scale chip multiprocessors (CMPs). In addition, we estimate the impact of memory system performance on the amount of instruction stalls, as well as on the real computational performance, using the number of floating point operations per second the applications perform.

Our projections show that the limitations of present memory technologies, either by means of capacity or bandwidth, will have a strong negative impact on scalability of memory systems for large CMPs. We conclude that future supercomputer systems require research on new alternative memory architectures, capable of offering both capacity and bandwidth beyond what current solutions provide.

3.1 Introduction

The inability to efficiently scale single-thread performance through frequency scaling and instruction level parallelism, has left on-chip parallelism as the only viable path for scaling performance, and vendors are already producing chip multiprocessors (CMPs) consisting of dozens of processing contexts on a single die [3, 42, 72].

But, placing multiple processing units on a single chip imposes greater stress on the memory system. While on-chip parallelism is effective at scaling the computational performance of a single chip (i.e. the number of arithmetic operations per second), it is unclear whether the memory system can scale to supply CMPs with sufficient data.

In this chapter, we characterize the memory behaviour of several well-known parallel scientific applications representing different scientific domain and selected based on their usage in supercomputing centers across Europe [73]. We predict the performance required from the memory system, to adequately serve large-scale CMPs. Based on the knowledge that we have today, and utilizing large processor clusters as our evaluation platform, we make a projection on memory requirements of future multicore systems, in terms of memory size, memory bandwidth and cache size.

Given the lack of parallel applications that can explicitly target future large-scale shared-memory CMPs, we base our predictions on the per-CPU memory requirements of distributed memory MPI applications. Although this methodology is imperfect (data may be replicated between nodes, which may result in pessimistic predictions when addressing shared-memory environments), we believe it provides a good indication of the requirements from a CMP memory system.

For the applications examined, we show that the per-core working set size typically consists of hundreds of MBs. In addition, we observe that per-core memory bandwidth reaches hundreds of MB/s in most cases, and that the bandwidths of both L1 and L2 caches are typically higher by an order of magnitude. A simple back-of-the-envelope calculation, therefore, suggests that a CMP consisting of 100 cores may require dozens of GBs of memory space, accessed at rates up to 100 GB/s.

Furthermore, we demonstrate the impact of memory system performance by analyzing its effect on instruction stalls, and by comparing theoretical and real arithmetic performance using the number of floating point operations per second that our benchmarks perform.

A common rule of thumb, used for designing the memory system of a supercomputer, dimensions memory size to 2 GB per core, and memory bandwidth to 0.5 bytes/FLOP. However, our results show that these estimates are much higher than real applications actually require.

The rest of this chapter is organized as follows. Section 3.2 highlights related publications on memory system analysis for multicores. Then, in Section 3.3 we describe our evaluation platform, applications under analysis, and performance evaluation methods. The following sections present our findings on memory footprint (Section 3.4), memory bandwidth (Section 3.5), as well as the impact of memory system on CPI stack (Section 3.6) and arithmetic performance (Section 3.7). Finally, we summarize our conclusions in Section 3.8.

3.2 Related work

Until recent years, the inaccessibility of large scale parallel machines have limited the ability of researchers to study the memory performance and requirements of parallel applications. As a result, common wisdom concerning these requirements mostly relies on characterizing established benchmarks suites, such as SPLASH-2 [86] or NAS [1].

At the architectural level, Burger et al. [12] point that many techniques used for tolerating memory latencies do so at an increased memory pin bandwidth, which will eventually become a critical bottleneck. They conclude that in the short term, more complex caching mechanisms can alleviate this bottleneck, but in the long term, off-chip accesses will become too expensive.

Liu et al. developed the memory intensity metric to evaluate the load imposed by parallel applications on off-chip memory bandwidth of CMPs [48]. Memory intensity was defined as the number of bytes transferred to and from the chip per executed instruction, thereby taking into account data locality that is captured by the on-chip cache. They show that, for a given parallel program, when the number of executing cores exceeds a certain threshold, performance will degrade due to the bandwidth problem.

Murphy et al. [58] quantitatively demonstrated the memory properties of real supercomputing applications, by comparing them with SPEC benchmark suite in terms of temporal locality, spatial locality and data intensiveness. They showed that the number of unique data items that the application consumes can have an impact on the performance of hierarchical memory systems much more than the average efficiency with which data is stored in the hierarchy.

Alam et al. [5] studied how different memory placement strategies affect overall system performance, by evaluating different AMD Opteron-based systems with up to 144 cores. They measured computational characteristics of the architecture and communication performance using low-level micro-bechmarks, a subset of NAS benchmark suite, and several MPI benchmarks. They have shown that optimal selection of MPI task and memory placement schemes can result in over 25% performance improvement.

Finally, Bhadauria et al. explored the effects of different hardware configuration on the perceived performance of the PARSEC benchmarks [10]. In order to evaluate how memory bandwidth affects the performance of PARSEC benchmarks, the authors reduced the frequency of the DRAM channels connected to a 4-way CMP, and concluded that memory bandwidth is not a limited resource in this configuration.

In contrast to the above, we focus on the potential of large-scale CMPs to serve as a supercomputing infrastructure, by characterizing well-known highly parallel scientific applications, and projecting how both current and emerging technologies can scale to meet their demands.

3.3 Methodology

Our analysis is based on a combination of tracing high-level events together with reading the hardware performance counters. The application execution is instrumented at the higher abstraction level: CPU bursts, synchronization and communication events. It produces a full timestamped trace of events, annotated with hardware performance counters and memory usage statistics associated to each CPU burst.

The target platform used for obtaining these measurements is the MareNostrum supercomputer [54], which consists of a cluster of JS21 blades (nodes), each hosting

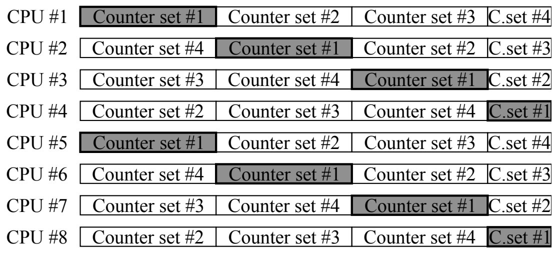

CPU #1 Counter set #1 Counter set #2 Counter set #3 C.set #4
CPU #2 Counter set #4 Counter set #1 Counter set #2 C.set #3
CPU #3 Counter set #3 Counter set #4 Counter set #1 C.set #2
CPU #4 Counter set #2 Counter set #3 Counter set #4 C.set #1
CPU #5 Counter set #1 Counter set #2 Counter set #3 C.set #4
CPU #6 Counter set #4 Counter set #1 Counter set #2 C.set #3
CPU #7 Counter set #3 Counter set #4 Counter set #1 C.set #2
CPU #8 Counter set #2 Counter set #3 Counter set #4 C.set #1

Figure 3.1: Spatial and temporal distribution of counter sets

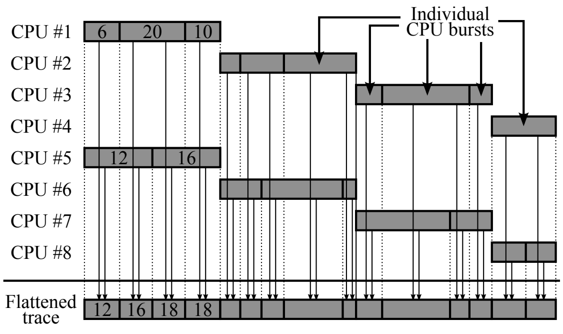


Figure 3.2: Flattening the sparse per-processor traces into a unified trace

4 IBM Power PC 970MP processors running at 2.3 GHz. Each node has 8 GB of RAM, shared among its 4 processors, and it is connected to a high-speed Myrinet type M3S-PCIXD-2-I port, as well as two GigaBit Ethernet ports.

In order to avoid contention on the nodes' RAM, the benchmarks were executed using only a single processor per node. Therefore, an application running on 64 processors actually had exclusive access to 64 nodes and 256 processors, of which 192 were idle (3 per node), so that each processor used by the application had 8 GB of memory and the full bandwidth at its disposal.

Probe execution of the selected set of applications using all four available processors per node, resulted in a decrease of average per-processor bandwidth by 10%–20%, compared to the execution using only one processor per node. This proved our assumption that the contention on one nodes' RAM, even for those applications that are less memory-intensive, would introduce error in measuring actual bandwidth requirements of each processor.

A very large set of performance counters on PowerPC 970MP allowed us to track in detail a wide spectrum of memory-related events. However, as it is the case with most of the modern processors, PowerPC 970 has only a limited set of physical counters that can track as many unique events at the same time. The set of counters that the processor tracks depends on the processor's active counter group. In total, there are ten counter groups, that non-exclusively cover fifty counters.

In order to have as many counters as possible available for further analysis, we configured our tracing mechanism to apply different counter groups to the processors involved in the execution in a cyclic fashion, where first processor would start tracking counters from first counter group, second processor from second counter group, and so on. To enhance further this spacial distribution of counter groups, we also configured each processor to switch its active counter group to the next one every five seconds. The execution itself typically took several minutes. Since, in all our experiments, we used minimum 16 processors, we were able to assign each of the ten counter groups to at least one processor.

An example of a distribution of counter sets is presented in Figure 3.1, where 8 CPUs are involved, with 4 counter sets, and the selected counter being in set #1. At any given moment two of the processors are tracking the selected counter.

We calculated average value of each counter for every CPU burst throughout the whole execution time, that way producing a flattened trace as a unified timeline of all the collected counters. Finally, since the resulting time series consisted of millions of short periods, we implemented a simple bucketing algorithm that divided the whole execution time in 200 equal segments and averaged our target values over each of those segments.

Figure 3.2 depicts the trace flattening process, by showing only those execution segments that track the selected counter. Each of the segments consists of a series of individual CPU bursts that contain the values of the selected counter. Flattened trace is divided into segments that correspond to the CPU bursts. Counter value for each segment is then derived from the two CPU bursts as the sum of their selected counter values, and by taking into account respective burst duration.

Importantly, our characterization methodology is validated against the performance reported by well-known benchmarks such as LINPACK [20], the de-facto benchmark of floating-point performance for high-performance scientific computing.

We base our analysis on a set of four applications, selected according to Simpson et al.'s survey of high-performance applications used in supercomputing centers across Europe [73], and chosen to represent the dominant scientific domains in the supercomputing centers surveyed. More importantly, the analysis showed that each of the selected applications stress different aspects of the memory system. The selected applications include:

- *GADGET* (GAlaxies with Dark matter and Gas intEracT). A code for cosmological simulations of structure formation. It computes gravitational forces with a hierarchical tree algorithm, optionally in combination with a particle-mesh scheme for long-range gravitational forces. It is one of the most often used applications, representing the area of astronomy and cosmology. From the four applications that we used, GADGET had the highest requirements of memory size.
- *MILC* (MIMD Lattice Computation). A set of codes for doing simulations of four dimensional SU(3) lattice gauge theory, represents the area of particle physics, and in our analysis had the highest requirements of memory bandwidth.
- *WRF* (Weather Research and Forecasting). A next-generation mesocale numerical weather prediction system designed to serve both operational forecasting and atmospheric research needs. It is a well-known DEISA benchmark from the area of earth and climate. In our analysis, it is characterized as the application that is bound by the system's computational resources.
- *SOCORRO* — self-consistent electronic-structure calculations utilizing the Kohn-Sham formulation of density-functional theory. Calculations are performed using a plane wave basis and either norm-conserving pseudopotentials or projector augmented wave functions. This application mostly stresses the memory bandwidth.

Each of the applications was executed on 16, 32, 64 and 128 processors, with the exception of GADGET — whose memory footprint could not fit on 16 MareNostrum blades.

The input sets in all of the analyzed applications remained unchanged while scaling the number of processors. With the advance in multicore design, and scaling the number of processors, it is reasonable to expect that the input sets of the

Table 3.1: The per-processor and overall memory footprints measured for the benchmark applications.

Application	#CPUs	Footprint [GB] per CPU avg	Footprint [GB] per CPU max	Footprint [GB] total	Maximum footprint reduction	Total footprint increase
GADGET	32	1.27	1.80	57.69	-	-
	64	0.75	0.98	62.58	45.76%	8.49%
	128	0.49	0.68	86.85	30.61%	38.78%
MILC	16	0.57	0.61	9.71	-	-
	32	0.29	0.31	9.90	49.00%	1.99%
	64	0.15	0.16	10.28	48.10%	3.81%
	128	0.07	0.09	11.04	46.29%	7.41%
WRF	16	0.19	0.20	3.17	-	-
	32	0.12	0.12	3.90	38.39%	23.21%
	64	0.07	0.07	4.77	38.85%	22.30%
	128	0.05	0.05	6.78	28.94%	42.12%
SOCORRO	16	0.18	0.20	3.19	-	-
	32	0.11	0.12	3.96	37.82%	24.36%
	64	0.08	0.09	5.56	29.87%	40.25%
	128	0.06	0.07	8.68	21.93%	56.14%

applications being executed will also grow. It is clear that the memory requirements of such applications can only be higher than the ones that we evaluate, and that the memory technology limits can be reached sooner than projected in this work.

3.4 Memory footprint

In this section, we try to quantify the memory footprint of parallel applications, as a key factor determining the size requirements of both on-chip and off-chip memory in future CMPs.

Table 3.1 shows the average and maximum per-processor footprints for the applications under analysis. The total footprint is calculated as the number of processors multiplied by the maximum per-processor footprint. In case that the memory system does not satisfy the maximum footprint requirements of a given application, it would crash when left out of memory space. The table also shows the relative reduction in maximum per-processor footprint, as well as relative increase in total footprint, both compared to the execution with half as many processors.

Figures 3.3a to 3.3d describe the progression of the average per-processor memory footprint for the four applications. The vertical axis shows the memory footprint in GB, and the horizontal axis depicts the progression of normalized execution time.

As expected in a strong scalability case, when we divide the same working set across a large number of processors, the per-processor memory footprint decreases as the number of processors participating in the computation increases. However, doubling the number of processors does not halve the size of the per-processor memory footprint. For both WRF and SOCORRO, doubling the number of processors only reduces the per-processor memory footprint by 20–40%. Scaling is somewhat better for GADGET, for which scaling from 32 to 64 processors reduces the per-processor footprint by 45%, while scaling further to 128 processors reduces the footprint by only 30% more. Scaling for MILC is very good, as footprint reduction is very close to 50%.

This suboptimal reduction in per-processor memory footprint is partially an artefact of using distributed memory applications, and it is caused by the replication of data between processors. Many distributed algorithms that employ spatial metrics to partition the entire problem set into segments, replicate segment borders among processors assigned with neighbouring segments. For example, partitioning a large matrix into small sub-matrices will typically involve replication of the rows and columns on segment borders. Therefore, reducing the per-processor segment size will inevitably increase the portion of the segment that constitute as part of its border — that is replicated on the processor assigned with the neighbouring segment, and thus increase the percentage of application data that is replicated among the nodes. As a result, total memory footprint increases along with the number of processors, as shown in Table 3.1. Shared memory environment, on the other hand, would not experience an increase in the total memory footprint, but still, we would see more impact on the working set captured by caches and increase in cache coherency traffic.

Figure 3.4 extends the measurements through linear regression, and projects the total memory footprint for larger number of processors. The vertical axis on Figure 3.4 shows the estimated total footprint, and the horizontal represents the number of processors. Points on the graph represent the actual values measured from the executed application, while the lines show the linear regression of the total footprint for the particular application (the projection lines do not have a linear appearance due to the logarithmic axes).

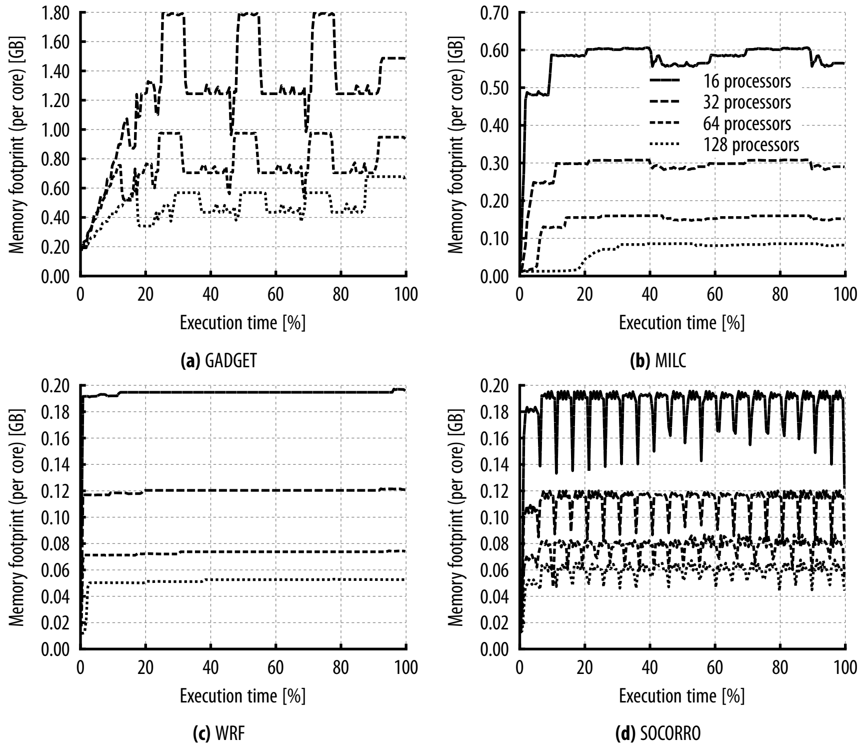


Figure 3.3: Evolution of the memory footprint

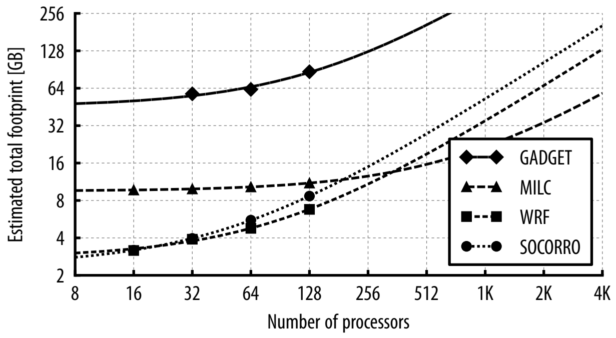


Figure 3.4: Projections of overall memory footprints for large-scale parallel systems, based on a linear regression model.

The amount of data replication identified in the discussed benchmarks also supports our projections about the usefulness of caching: even though each processor participating in a parallel computation needs to access data originally assigned to its neighbour, aggressive caching can capture such data-sharing patterns and prevent the data from going off-chip, thereby saving precious memory bandwidth. More details about this are provided in Section 3.5.2.

In summary, we see that future manycores consisting of more than 100 cores must be directly backed with a few dozens GBs of main memory in order to support scientific workloads.

3.5 Memory bandwidth

3.5.1 Bandwidth scales linearly

Consolidating multiple cores on a single chip imposes much higher bandwidth requirements on the shared components of the memory system — namely the off-chip memory and the shared caches. In order to predict those requirements, we measured the per-processor bandwidth consumed by each benchmark at three levels: the off-chip memory, L2 cache, and L1 cache.

Table 3.2 shows the average per processor bandwidth in each of the three levels: the off-chip memory, L2 cache, and L1 cache. It also shows the estimated total memory bandwidth required by all the processors involved in the execution combined.

Table 3.2: The memory bandwidth measured at different levels of the memory system for the benchmark applications.

Application	#CPUs	Average bandwidth per processor [GB/s]			Total memory bw. [GB/s]
		Memory	L2 cache	L1 cache	
GADGET	32	0.114	4.50	11.04	3.57
	64	0.100	3.66	10.96	6.26
	128	0.068	2.90	10.91	8.44
MILC	16	0.815	0.52	14.12	12.73
	32	0.598	0.80	13.47	18.70
	64	0.604	0.75	13.38	37.78
	128	0.617	0.77	13.29	77.18
WRF	16	0.117	1.55	7.35	1.82
	32	0.102	1.87	8.32	3.18
	64	0.091	1.70	8.87	5.69
	128	0.050	1.95	10.09	6.22
SOCORRO	16	0.331	4.14	15.35	5.18
	32	0.279	3.35	13.39	8.72
	64	0.212	2.83	12.08	13.23
	128	0.228	2.73	11.75	28.49

Figure 3.5 uses linear regression to present the estimated total memory bandwidth required by each application for a particular processor count. Horizontal axis presents the number of processors used for the execution, while the vertical axis presents the memory bandwidth. Points in the figure show the actual measured values, while the lines show the linear regression, that is calculated in order to estimate the total bandwidth requirements for a larger number of processors. Each series stands for one of the four benchmark applications used in our evaluation.

Both Figure 3.5 and Table 3.2 show that bandwidth requirements for MILC and

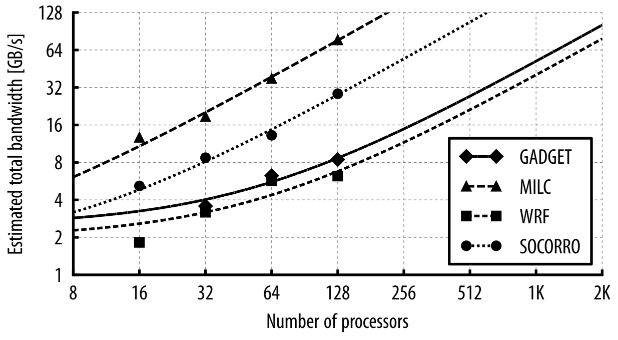


Figure 3.5: Projections of overall memory bandwidth for large-scale parallel systems, based on linear regression.

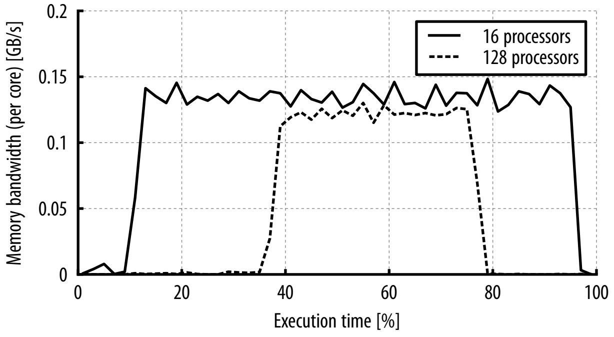


Figure 3.6: WRF memory bandwidth

SOCORRO grow almost linearly with the increase of the number of processors. MILC is the most demanding application in terms of bandwidth, and requires close to 80 GB/s when executed on 128 processors. SOCORRO behaves in a similar way, reaching almost 30 GB/s for 128 processor execution.

GADGET and WRF do not have such a steady growth, and for different reasons. GADGET tends to benefit from an increased cache effectiveness, due to the reduced working set per processor (as discussed in Section 3.4), and, therefore, less bandwidth is required from the main memory. The L1 cache bandwidth stays on the same level, which suggests that the total bandwidth requirements of each processor barely changes with the increase of the number of processors.

On the other hand, WRF experiences a different behaviour, as its initialization and finalization phase, which hardly produce any off-chip memory traffic, start to occupy a significant part of the total execution, with the increased number of processors. In contrast, its computation phase, which is very memory demanding, becomes relatively shorter and shorter. Therefore, the average bandwidth over the entire execution decreases, although it stays on the same level during the computation phase. Figure 3.6 demonstrates this phenomenon by showing how bandwidth progresses with time. Horizontal axis represents normalized execution time, and vertical axis off-chip memory bandwidth. For brevity, Figure 3.6 depicts memory bandwidth for 16 and 128 processors only.

In summary, we observe that future manycore systems consisting of more than 100 cores may easily require more than 100 GB/s of main memory bandwidth. Modern architectures such as Intel Nehalem-EX [43] or IBM Power7 [34] employ 4 and 8 DDR3 channels respectively, peaking at 102.4 GB/s of bandwidth. Knowing that the sustained bandwidth is typically 20%–25% lower due to page and bank conflicts, we conclude that such large-scale systems will need to provide higher bandwidth to support high-performance scientific computing.

3.5.2 Cache effectiveness

Compared with the off-chip bandwidth, the observed L2 cache bandwidth is typically an order of magnitude higher. This is understandable as the L2 cache hits filter bandwidth that would otherwise go off-chip. The same conclusion would apply for L1 versus L2 bandwidth. To better understand the effectiveness of the caches, as well as the effect of increased parallelism on caches, we have investigated the relation between L1 and L2 cache hit rates, and off-chip memory bandwidth.

In Table 3.3, we present L1 and L2 hit rates, as well as the effect they have on off-chip memory bandwidth per processor (also shown in Table 3.2).

We already mentioned, in Section 3.5.1, the effect of increased cache effectiveness for GADGET. It is interesting to observe, from Table 3.3, that L1 cache is the one that makes the difference, although its size of 32 KB compared to GADGET's extremely large working set of more than 0.5 GB per processor, seems relatively small. This could mean that GADGET has very regular memory access patterns, which target relatively small memory blocks, so that L1 cache can capture most of the memory accesses, and have increase in effectiveness as the size of the blocks decrease with higher number of processors.

MILC, on the other hand, is a bit less demanding when having memory footprint in mind, which makes the transition from 16 to 32 processors (and from 0.57 to 0.29 GB of memory footprint) very favourable for L2 cache. Reduction in per-processor working set, led to a better locality of access patterns, and that way better fitting in L2 cache. Huge increase in L2 cache hit rate, from 39.31% to 57.66%, made the impact on filtering a great deal of off-chip memory traffic.

Furtheremore, MILC's scalability results in an increase in the relative duration of the initialization phase, compared to the total execution time, as the number of processors increase (similar to WRF). Therefore, the stable average hit-rates of the different caches actually hide the fact that the different phases exhibit very distinct cache behavior, with the initialization phase enjoying an impressive L1 hit rate of over 99%, whereas the L1 hit-rate of computation phases decreases as the level of

Application	#CPUs	L1 cache hit rate	L2 cache hit rate	Mem. bw. [MB/s]
GADGET	32	93.67%	97.58%	114.26
	64	94.41%	97.40%	100.09
	128	95.09%	97.77%	67.53
MILC	16	97.13%	39.31%	814.56
	32	97.00%	57.66%	598.41
	64	96.43%	55.80%	604.44
	128	96.74%	56.07%	617.40
WRF	16	93.39%	93.15%	116.67
	32	93.71%	94.94%	101.90
	64	94.43%	95.02%	91.04
	128	94.99%	97.57%	49.74
SOCORRO	16	97.21%	92.74%	331.31
	32	97.38%	92.47%	279.16
	64	96.83%	93.20%	211.63
	128	97.02%	92.45%	227.91

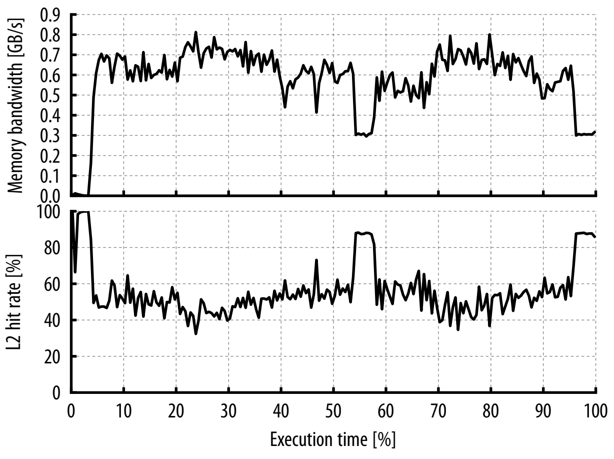
Table 3.3: Hit-rates measured for the different cache levels.

parallelism increases — from 97% for 32 processors, down to around 95% for 128 processors. These lower hit-rates in the computational phases, particularly when running on 128 processors, have a direct impact on the memory bandwidth (which increases from 598.41 to 617.40 MB/s) and memory related pipeline stalls.

WRF has particularly high hit rate for both L1 and L2 cache, and therefore, only a small fraction of all memory accesses ends up reaching off chip memory. Even a seemingly minor increase in cache effectiveness from 93% to 97% can lead to a big decrease in off-chip memory traffic. If we combine this fact with the effect of relative reduction of WRF's computation phase (discussed in Section 3.5.1, and shown in Figure 3.6), the resulting memory bandwidth per processor drops from 116.67 MB/s with 16 processors down to 49.74 MB/s with 128 processors.

Effective caching has a direct impact on memory bandwidth, as shown on Figure 3.7. The figure shows the relation between off-chip memory bandwidth (shown on top subplot), and L2 cache hit rate (shown on bottom subplot), of a MILC executed on 32 processors. Due to the lack of space, we omit similar cache effectiveness figures for other applications. With a constant memory access rate and a constant L1 hit rate, the two depicted metrics are inversely proportional. The figure clearly identifies the initialization phase and two iterations, with L2 cache hit rate peaking to more than 80% during the end of each iteration. As the hit rate increases, the

Figure 3.7: MILC 32p L2 cache effectiveness



off-chip memory goes from 0.6–0.7 GB/s down to ~0.3 GB/s.

In summary, caches prove to be effective bandwidth filters, and any further advances in their performance may delay reaching the point when bandwidth requirements of an application can no longer be supported by the existing memory technologies.

However, contrary to the memory footprint, maximum bandwidth requirements imposed by the running application do not need to be met in order for the application to run. Of course, it is preferred that the memory system provides the required bandwidth, but if not, the application would not crash. It is clear that the insufficient bandwidth would hurt the performance of the system, but we can only speculate about this performance degradation. For example, if memory provides only 50% of the bandwidth that one execution segment requires, and if we assume that this segment is completely memory bound (there are no stall cycles due to any other core resource but the memory), it would be safe to say that this execution segment would run half of its maximum speed.

CPI stack component					Index	Color
Total cycles	Completion stall cycles	others (Stall by BRU/CRU instruction, flush penalty (except LSU flush))			9	□
		Stall by FPU instruction	Stall by FPU basic latency		8	▨
			Stall by any form of FDIV/FSQRT instruction			
		Stall by FXU instruction	Stall by FXU basic latency		7	■
			Stall by any form of DIV/MTSPR/MFSPR			
		Stall by LSU instruction	Stall by basic latency		6	□
			Stall by D-cache miss		5	▨
			Stall by reject	Other reject	4	■
				Stall by translation (rejected by ERAT miss)		
	Completion table empty cycles	others (Flush penalty etc.)			3	□
		Branch redirection (branch misprediction) penalty				
		I-cache miss penalty			2	▨
	Completion cycles	overhead of cracking/microcoding and grouping restriction			1	■
		PowerPC base completion cycles				

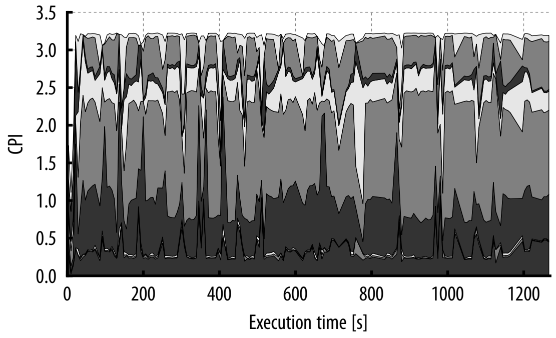
Table 3.4: CPI stack model

3.6 CPI stack

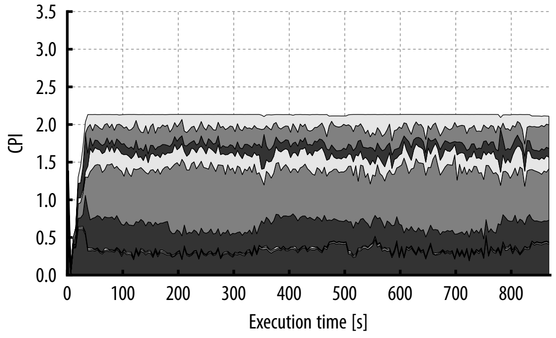
The *cycles per instruction* (CPI) metric is defined as the average number of processor cycles needed to complete one instruction. For any execution segment, it is calculated as a total of elapsed cycles divided by the number of completed instructions. A low CPI value means that the system resources are better utilized, and the architecture operates closer to its peak performance.

The PowerPC 970MP processor dispatches instructions to its back end in groups of five. Therefore, the theoretical execution throughput is five instructions per cycle, or conversely, $\frac{1}{5} = 0.2$ cycles per instruction. Several restrictions in PowerPC 970's dispatch queue, as well as insufficient instruction-level parallelism in real applications, prevent CPI from being as low as 0.2. Instead, our experiments show the CPI in real applications gets usually between 1 and 1.5, sometimes even as high as 3 (5–15× slower than the peak throughput).

The CPI stack model, which is the breakdown of CPI value to the individual

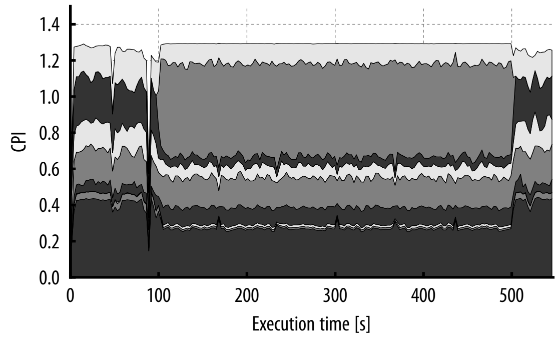


(a) MILC 16p

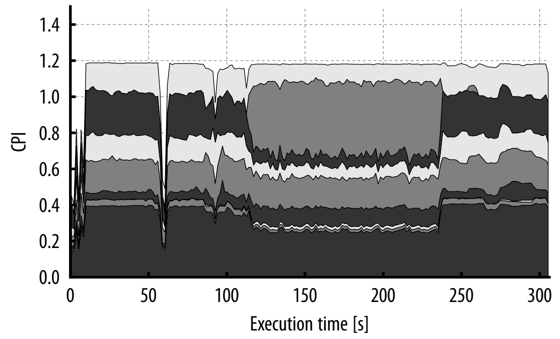


(b) MILC 32p

Figure 3.8: MILC CPI stack



(a) WRF 32p



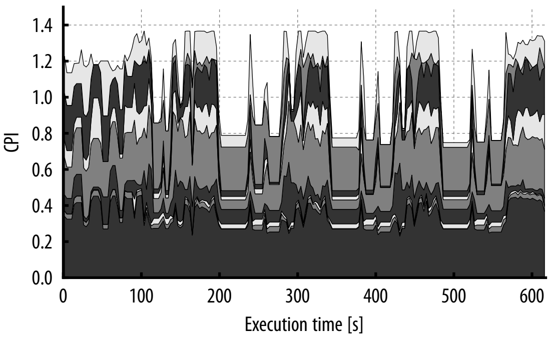
(b) WRF 128p

Figure 3.9: WRF CPI stack

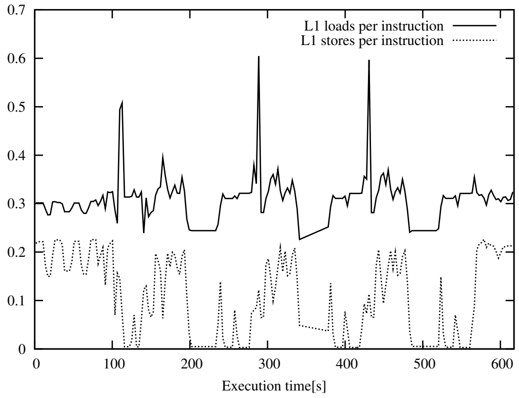
latencies contributed by different micro-architectural resources, can therefore be used to determine the key factors that impede performance. Each component in the stack describes the average number of cycles an instruction stalled on a particular core resource (like Load/Store unit, or Floating Point unit).

We construct the CPI stack using the PowerPC 970MP performance counters [55]. Some of them are specific to PowerPC 970MP architecture, so the CPI stack model for other architectures may contain minor differences. The individual stack components are described in Table 3.4. The table also includes both index and color used in the CPI stack figures presented in this section.

Figures 3.8–3.11 present the evolution of the CPI stack throughout the entire run of the benchmark applications (due to space constraints, we only show the full evolution of a subset of the applications and architectural configurations). The global



(a) GADGET 32p CPI stack



(b) GADGET 32p L1 cache traffic

Figure 3.10: GADGET 32p — CPI stack and L1 cache traffic

averages are shown in Figure 3.12. For example, MILC running on 16 processors (Figure 3.8a) requires 3.2 cycles to complete an instruction, of which an average 0.2 cycles are spent on committing the instruction, 2.3 cycles on stalls associated with the load/store unit (LSU), 0.5 cycles on the fixed-point and floating-point units (FXU and FPU, respectively), and 0.2 cycles on other stalls.

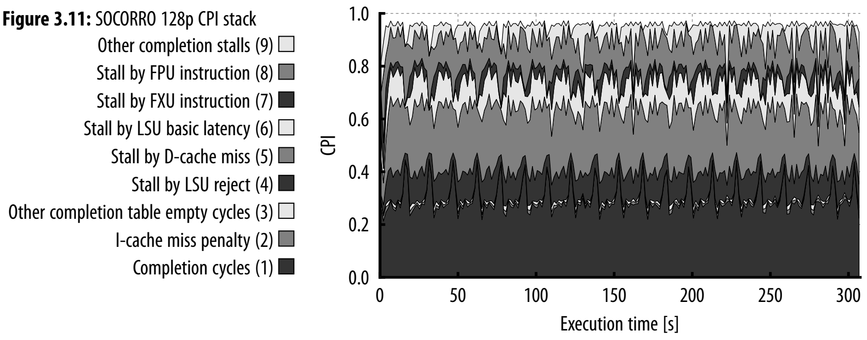
All the presented CPI stack figures show that both I-cache miss penalty and branch misprediction penalty have negligible impact on performance. This means that the I-cache is large enough for all the tested applications, so that it rarely experiences a miss. Also, the branch prediction shows efficiency, as expected for scientific applications [15]. Importantly, these findings are not affected by the increase in parallelism.

Figure 3.8 shows that scaling MILC from 16 to 32 processors dramatically decrease all LSU related stalls, from 2.3 cycles for 16 processors, to 1.5 cycles for 32 processors. The benefit is evident both in the overall CPI value, as well as the execution time. As shown in Section 3.5.2, this performance improvement is due to MILC's reduction in per-processor memory footprint. When running on 16 processors, MILC's working set exceeds the size of data cache. But the increase in the number of processors reduces the per-processor memory footprint such that it fits in the cache for the 32 processor configuration. With further increase in number of processors, we do not see such a dramatic improvement in CPI value, because the working set already fits in cache.

In the case of WRF, the evolution of the CPI stack clearly reveals the application's initialization, computation and finalization phases (same as those observed in Section 3.5.1), which are clearly distinguished by their FPU usage. The initialization and finalization phases hardly exhibit any FPU stalls. In contrast, the computation phase is highly FPU dependent as about half of its instructions' stall time is associated with the FPU. As the number of processors increases, WRF's scalability allows its computation phase to shorten. However, the initialization and finalization phases are evidently non-scalable and do not benefit from the increased parallelism. As such, their execution time does not change, and, therefore, begin to dominate the overall execution time. This implies that the application's overall scalability is limited, unless the input set size increases.

GADGET's most obvious CPI stack patterns, shown in Figure 3.10a, are significant fluctuations of the overall CPI value, revealing three periodic iterations. Parts of each iteration with low FPU usage have an overall high CPI value, whereas parts

Figure 3.11: SOCORRO 128p CPI stack



with high FPU usage have low CPI value. When correlating this with the frequency of memory accesses, presented in Figure 3.10b, it is noticeable that high CPI value corresponds to high memory traffic (more specifically high number of stores). This also justifies fairly large number of LSU stalls (LSU reject, D-cache miss and LSU basic latency) in this part of the iteration. Therefore, each of the three iterations can be divided into a communication phase (low FPU usage, lots of memory accesses, high bandwidth, high CPI), and a computation phase (high FPU usage, few memory accesses, low bandwidth, low CPI). We do not see much variation in GADGET graphs for 32, 64 or 128 processors.

Finally, SOCORRO is yet another example of an application that is memory bound, as we can observe in Figure 3.11. Even though its per-processor working set easily fits in the cache (Section 3.5.2), its periodic LSU and cache stalls imply that each of its iterations operates on a separate chunk of data. This data is transferred from the memory to the cache, processed, and stored back to memory. This scanning access pattern puts a significant load on the memory system, regardless of the cache size or number of processors.

Figure 3.12 presents the average CPI stack for all the applications tested. First, it appears that GADGET's computation pattern does not change when increasing the number of processors to 128, as indicated by its similar CPI stacks. For MILC, we observe a gradual increase in CPI (following the aforementioned drop when increasing the number of processors from 16 to 32), which is mostly attributed due to LSU related stalls. This is a consequence of a decrease in L1 cache effectiveness, and therefore, higher memory traffic (discussed in Section 3.5.2). WRF, on the other hand, enjoys a slight decreasing CPI trend, due to an increase in the relative duration of memory intensive phases, and a decrease of computationally intensive

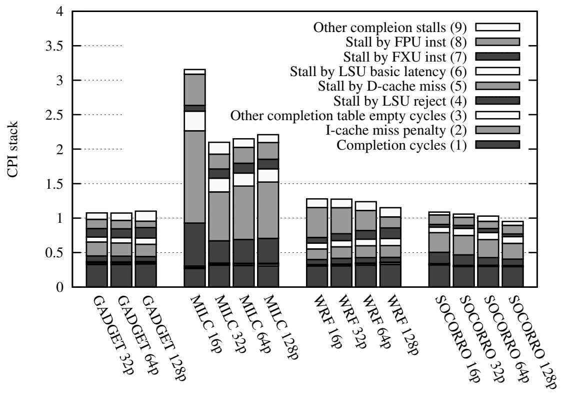


Figure 3.12: CPI stack average values

phase. Therefore, all LSU related stall values grow, while FPU related stall values decline. Finally, SOCORRO's CPI seems to benefit slightly with scaling number of processors, mostly due to a lower amount of LSU related stalls.

In summary, LSU related stalls seem to be the most dominant CPI stack component in the applications tested. The exceptions are GADGET and WRF, whose computation phases are limited by FPU stalls. It indicates that memory hierarchy could have an exceptionally high impact on performance of the future manycores. It is also clear that wider superscalar approach can have a limited impact, and the only ones that could see the benefit are FPU intensive applications.

CPI stack components not related to LSU can give us an indication on how scaling compute performance relates to scaling bandwidth requirements. That is, by shortening the time to process a block of data, we would increase the processing throughput, and, unless the cache hierarchy gets changed, increase the off-chip memory bandwidth demand. Therefore, making the same application analysis we have done in this work on a platform that outperforms PowerPC 970, would make the slope of our bandwidth projections on Figure 3.5 steeper, and reach memory bandwidth limits with fewer number of processors.

3.7 Arithmetic performance

In previous section we presented the impact of memory system on overall architectural performance, expressed as a rate of executed instructions. This section brings more focus on arithmetic performance, expressed as a rate of floating point operations (FLOPS) executed — probably the most important metric for determining supercomputer performance.

For PowerPC 970MP architecture, the total number of floating point operations in an execution segment is calculated as a sum of the number of operations in each floating point unit, and the number of fused multiply-adds. Each floating point unit can process one floating point and one fused multiply-add instruction in a single cycle. Since we have two floating point units per core, maximum number of floating point operations per cycle is 4, which would give a theoretical maximum of 9.2 GFLOPS on a machine running at 2.3 GHz.

All our calculations related to the arithmetic performance of a system, excluded execution segments that did not contain any floating point operations. This is done to estimate better efficiency of floating point units during computation phases, eliminate changes in relative duration of computation phases compared to initialization and finalization phases, and enable correlation of memory and arithmetic performance through bandwidth-to-performance ratio. Section 3.6 gives an indication on the execution segments that utilize the available floating point resources.

Figure 3.13 presents average and maximum floating-point performance of our applications under study. The figure shows that real applications achieve only a small fraction of the peak performance. This sub-optimal utilization of all the floating point resources that the architecture provides can be attributed to various data dependencies the inhibit ILP. This is consistent with the results for the highly optimized LINPACK benchmark, which only achieves an average 5.2 GFLOPS — only 57% of the peak 9.2 GFLOPS processor performance.

System designers often use processor's arithmetic performance measurements to dimension the required performance of other computer system components. Dimensioning memory bandwidth is often based on a ratio of maximum bandwidth and maximum theoretical rate of floating point operations. A common rule of thumb for obtaining optimal performance is to keep this ratio around 0.5 bytes per flop. This means that a processor capable of achieving 9.2 GFLOPS should rely on

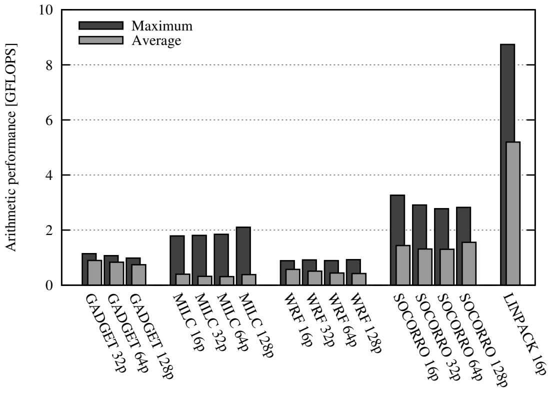


Figure 3.13: Arithmetic performance

memory that supplies 4.6 GB/s of bandwidth. However, our analysis show that this ratio is heavily overestimated, and that it should not be taken into account at all, mostly due to the underutilization of floating point resources in real applications.

Figure 3.14 shows bandwidth-performance ratios for all the applications under study, averaged over their time spent in computation phases. As we can see, most of the applications, with the exception of MILC, fit very well with projected bandwidth-performance ratio of 0.5 B/flop, that has been mentioned before. However, previous measurements of average arithmetic performance made clear that the reason for this ratio being relatively high is not high bandwidth requirements, but low arithmetic performance achieved. And indeed, in case of MILC, which performs only 0.3–0.4 GFLOPS bandwidth-performance ratio gets considerably higher than the other applications. On the other hand, LINPACK, with its average 5.2 GFLOPS, brings down the ratio to only 0.08 B/flop.

In conclusion, we observe that dimensioning memory bandwidth based on peak arithmetic performance may be inadequate, due to the overwhelming under-utilization of processor arithmetic performance in real applications. In this case, estimates of 4.6 GB/s of memory bandwidth per processor are considerably above the effective requirements. This is confirmed by the results presented in Section 3.5, which shows that none of the test applications exceeds bandwidth of 1 GB/s.

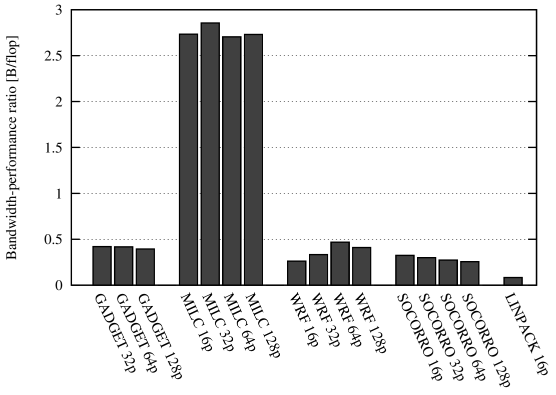


Figure 3.14: Bandwidth-performance ratio

3.8 Summary

The increasing multicore density is putting a proportionally higher stress on the memory system. It is unclear if current memory system architectures will be able to sustain the increasing number of on-chip cores.

In this chapter we evaluate the memory system requirements of HPC applications, running on the MareNostrum supercomputer at BSC, and characterize the memory performance requirements of future manycore designs.

We show that memory size requirements are actually closer to 0.5 GB per core, and memory bandwidth requirements are under 0.1 Bytes per peak flop. This is in contrast to the existing (undocumented) rule of thumb for designing the memory system of a supercomputer that call for 2GB of memory per core, and 0.5 bytes/flop peak bandwidth.

Next, we show that on-chip caches, originally placed to mitigate memory latencies, are very effective at filtering bandwidth to the off-chip memory, with on-chip bandwidth requirements being orders of magnitude higher.

Moreover, we evaluate the actual flops achieved by the applications, and show them to be only a fraction of the peak performance. It is only when computing bytes per real flops that the bandwidth requirements get close to the 0.5 mark, but it is due to the low flops, not to the high bandwidth.

In light of our findings, we expect that current memory architectures based on on-chip memory controllers and multiple parallel DDR channels should be able to sustain multicores for the next decade.

Chapter 4

Hybrid memory architectures

The performance of HPC applications is often bounded by the underlying memory system's performance. The trend of increasing the number of cores on a chip imposes even higher memory bandwidth and capacity requirements. The limitations of traditional memory technologies are pushing research in the direction of hybrid memory systems that, besides DRAM, include one or more modules based on some of the higher-density non-volatile memory technologies, where one of them will provide the required bandwidth, while the other will provide the required capacity for the application. This creates many challenges with data placement and migration policies between the modules of such hybrid memory system. In this chapter, we propose an architecture with a hybrid memory design that places two technologically different memory modules in a flat address space. On such system, we evaluate several HPC workloads against different data placement and migration policies, compare their performance by means of execution time and the number of non-volatile memory writes, and consider how it can be applied to the future HPC architectures. Our results show that the hybrid memory system with dynamic page migration and limited DRAM capacity, can achieve performance that is comparable to a hypothetical, hard to implement, DRAM-only system.

4.1 Introduction

A growing disparity in the rates of performance improvement between CPU and memory technologies has created a memory wall. So far, its negative impact has been relieved mostly by creating multi-level cache systems. At the same time, the increase of a single thread performance has reached a performance wall due to the inability to increase the operating frequency and to extract the instruction level parallelism. As a solution, the research community and the manufacturers have resorted to the use of multi-core systems in order to increase the performance of the chip.

A recent study has shown that the increase of the number of cores on a single chip puts great stress on the off-chip memory memory system: when executed on a 128-core system, HPC applications require 64 GB of capacity and may require up to 64 GB/s of off-chip memory bandwidth [66]. It is clear that current memory systems will not be able to sustain these requirements when the number of cores begins to increase.

SRAM and DRAM memories could provide the required bandwidth and capacity but at a great price. A large SRAM or DRAM memory would require a high amount of power: technology scaling brings significant increase in leakage power for both SRAM and DRAM and increases the power that is needed to refresh the cells in DRAM. Increasing the bandwidth of the the off-chip DRAM memory would require increasing the signalling frequency of the wires that connect the DRAM to the processor, limiting the length of this connection in order to preserve signal integrity. This would effectively restrict the area on the Printed Circuit Board (PCB) where DRAM chips could be placed, limiting the number of DIMMs that can be connected to a chip. Another alternative to increase bandwidth is to increase the number of channels (or channel width). This leads to the need for more pins on the processor, increasing the size and the cost of the processor itself, and would lead to the increase of the power consumption of the entire memory system. These problems (wire length and pin count) can be resolved by using 3D-stacked DRAM in the same package with the processor. 3D stacking can bring improvements in bandwidth but will offer only a limited amount of DRAM due to thermal dissipation and constrained area [51].

4.1.1 Overview of emerging memory technologies

To overcome these limitations, architects are looking into a number of emerging memory technologies that could replace DRAM as the off-chip memory.

Memristor-based memory technologies, like Spin-Transfer Torque Magnetoresistive RAM (STT-MRAM) [18] and Resistive RAM (RRAM) [83] store data as a resistance. They are non-volatile, power efficient and dense compared to standard SRAM and DRAM technologies. Read and write latencies of these memories are still larger than those of SRAM or DRAM, and RRAM cells can sustain a limited number of writes which limits their lifetime.

Another interesting memory technology is Phase Change Memory (PCM) that uses chalcogenide glass and exploits differences between its amorphous (high resistance) and crystalline (low resistance) states to store data. Existing products support only two states, but PCM allows memory cells to have multiple levels of resistance, enabling the storage of more than one bit per cell. The change between the states is achieved by applying high current to the memory cell, and requires more time than with STT-MRAM or DRAM (optimistic estimate claims ≈ 100 ns for writing a PCM cell [44]). Compared to other memory technologies, PCM offers excellent density but at a price of limited endurance (small number of writes into a single cell) and high energy that is needed to write the data.

Table 4.1 gives a comparison between SRAM and DRAM and the emerging memory technologies. Looking at the characteristics of the emerging technologies, we can see that there is no *silver bullet*: none of the technologies can provide fast access times combined with high density, high endurance and low power consumption. For example, when a PCM is used as a standalone main memory, the system is 1.6x slower and uses 2.2x more energy than a system with DRAM only [44]. To complicate things even more, non-volatile memories either have limited write endurance, or require high current for writing data and system architects should seek to minimize the number of writes to them. An overview of current research trends in hybrid memory systems is given in Section 4.4.

Out of all emerging memory technologies PCM is the one that is closest to production in large volumes [88], and in the rest of this chapter we focus on it. We analyze an architecture with hybrid off-chip memory system that combines small DRAM memory with large PCM memory module. We focus on HPC workloads

Table 4.1: Characteristics of current and emerging memory technologies.

Technology	Density	Read latency	Write latency	Endurance
SRAM	60–175 F^2	≈ 0.3 ns	≈ 0.3 ns	Very High
DRAM	4–15 F^2	≈ 1 ns	≈ 0.5 ns	Very High
PCM	6–20 F^2	≈ 60 ns	≈ 100 ns	Low
STT-MRAM	8–16 F^2	≈ 10 ns	≈ 10 ns	Very High
RRAM	1–4 F^2	≈ 50 ns	≈ 250 ns	High

that have high requirements from the memory system. This chapter makes the following contributions:

- Detailed modeling and simulation of a 128-core system with heterogeneous off-chip memory (DRAM and PCM).
- Evaluation of new page migration policies (LRU spill with empty page threshold, lifetime-aware back-migration)
- Analysis of hardware and software static and dynamic strategies for page placement in such a system from the aspect of both performance and the number of writes to the non-volatile memory.
- Analysis of trade-offs between performance and lifetime of non-volatile memories.

4.2 Proposal

4.2.1 System architecture

To stress the memory system as much as possible we focus on a large multi-core chip with 128 processors with L1 and L2 caches and two types of off-chip memory (Figure 4.1).

Each processor has a private L1 cache while L2 cache is shared and is distributed among cores. The system contains a fast and small DRAM memory, and a large PCM memory. Operating System is responsible for choosing the memory where a new page will be placed, and for making decisions about page migration from

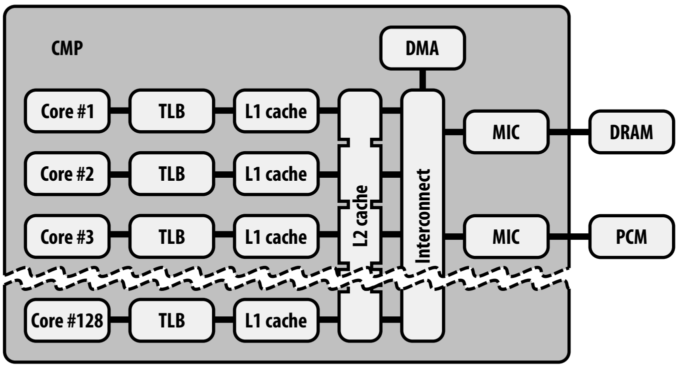


Figure 4.1: Target architecture

one memory to the other. The decisions about page placement and migration may be static or dynamic. In the latter case, OS may require information about pages that is stored in the Memory Management Unit (MMU), such as the number of the accesses to the page or the time the page was last accesses. The details about the page allocation and migration policies are given in the rest of this section.

MMU is responsible for translating virtual to physical addresses, and each processor contains a Translation Lookaside Buffer (TLB) to speed up the translation process. When a decision is reached to migrate the page between memories, MMU will use its DMA engine to perform the data movement. In order to prevent page allocation policies to negatively influence cache effectiveness by altering physical address access patterns, and that way significantly change the total number of requests that reach the main memory, we implemented a simple page coloring mechanism [36].

A migration starts by identifying which TLB holds the translation for a given page. The TLB and caches are then instructed to flush the cache lines that are part of that page. After possible writebacks are completed, DMA starts copying page contents from source to destination address. Finally, TLB is provided with the new translation of the logical page.

4.2.2 Static page placement

To better understand the effectiveness of the data placement policies that include migrations, we need to set our baseline using policies that allocate pages without altering their physical location during the application run.

First touch policy allocates pages in order in which they are requested by the cores, first in DRAM, and after exhausting DRAM's capacity, in PCM. The effectiveness of this policy is hugely influenced by the application's access patterns and by the size of DRAM. Performance gains are expected only if the initial access to a hot page happens while there is available space in DRAM. Conversely, if DRAM space becomes polluted with less reused pages, allocated because of their early first access, a performance degradation is imminent [30].

In order to evaluate the full potential of any static allocation policy, and to create the most favorable static distribution of pages between DRAM and PCM, we create a profile of memory accesses for each application under study. A profile is a list of all the logical pages accessed by each core during the execution, with the number of accesses to each page. A page placement policy can then use the profile to "predict" the traffic intensity on a particular page, and decide about its placement. Of course, all the policies that rely on the profile have to pre-run the application, or at least one of its iterations, to generate the profile itself. In many situations, this is impossible or impractical, but the results can still serve as an idealistic baseline in comparison against some other non-profile policy.

Static profile-based policy ensures that the most accessed pages are allocated in DRAM, and least accessed in PCM. The allocation is done at the beginning of the execution by reading the profile, sorting the pages by their access count in descending order, and allocating them in first in DRAM, and then in PCM. This policy should eliminate some of the downsides of first touch policy. It should bring performance gains by correctly dividing hot and cold pages over fast and slow memory ranges. For the same reason, it should also contribute in decreasing number of writes to PCM, that way extending its lifetime.

To avoid writes to PCM even more, we can modify this policy to take into account only write access count from the profile, instead of the sum of reads and writes. That way, PCM will host pages that are least frequently written regardless of their read traffic. This reduction in number of writes to PCM comes at the expense of performance, as hot read-only pages placed in PCM might bottleneck the system. Nevertheless, depending on the priority between performance and lifetime, this may prove as a justifiable tradeoff.

4.2.3 Spill migration

Static allocation policies can not exploit any of the temporal characteristics of the memory access pattern, because the page's initial physical location remains unchanged throughout the execution. The principle of data caching, prefetching and many other migration mechanisms are targeted to alter the distance of a given piece of data from the processor, depending on its potential for temporal reuse.

For this, we propose a spill migration — a policy that performs data movement in one direction only, from DRAM to PCM. Unlike traditional memory hierarchy with cache memories, where data is initially located further from the processor, and then gradually migrated closer as it experiences more reuse, spill migration policy first allocates a page in fast memory (in our case DRAM), and later evicts it to PCM. These evictions are performed due to the limited DRAM capacity, in order to make room for the newly allocated pages. This policy does not account for migrating back those evicted pages that turn to be heavily used after the initial migration has happened — whatever is copied to PCM stays there.

LRU spill policy keeps track of last access time for each page in DRAM, and in case of eviction selects one that is least recently used. The rationale behind this policy lies in the assumption that all the data used by the application throughout its execution time can roughly be divided in two categories: first, data that is reused most of the time, and second, data that is reused in a limited period only, or very rarely reused. If we then assume that DRAM capacity is large enough to fit all the pages that host data from the first category, we can expect that, because of their reuse, they will never be selected for eviction from DRAM. Then, the evicted pages should theoretically be those that are rarely used, and those that completed their high reuse period. The policy expects that the newly allocated pages will experience a significant traffic, at least in the short period after their allocation. Therefore, the gain obtained by initially allocating them in DRAM will justify the cost of migrating them to PCM, if they become less frequently used later.

Eviction from DRAM can take significant time, especially in a system where in-

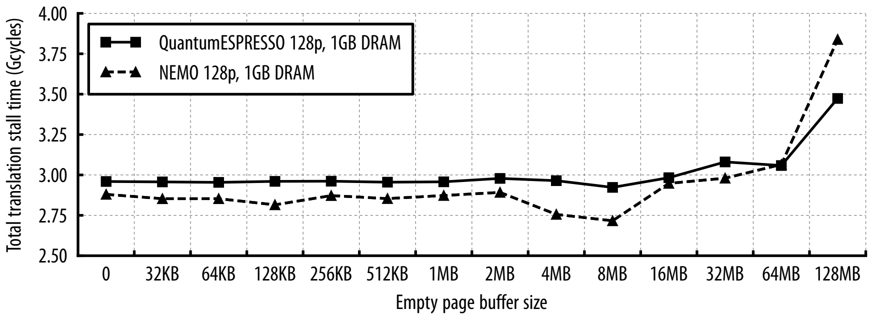


Figure 4.2: Empty page buffer size analysis

terconnect and memory can become congested by many simultaneous migrations, each triggered by a memory access from a different core. A naive eviction mechanism would start evicting when one of the TLBs makes a page table miss, and there are no more empty pages in DRAM's address space. It means that the translation of such memory access would have to stall until the eviction is finished, which leads to overall execution performance degradation.

To reduce the number of translation stalls, and to eliminate page eviction time from the translation critical path, we propose triggering the eviction once the number of empty pages in DRAM falls below a selected value — an *empty page threshold*. That way, corresponding TLB can immediately be provided with the translation, and continue processing the request, while the eviction is performed in the background. This can also enable more complex eviction mechanisms, and more detailed last access time analysis, once their operation time is taken off the translation critical path.

Selecting the value of empty page threshold is not trivial. Setting it too low would give TLBs a chance to quickly exhaust a small set of empty pages before any of the scheduled migrations finishes. Setting it too high would effectively reduce DRAM capacity, cause premature evictions, and significant interconnect traffic due to many "on-the-fly" migrations.

To confirm these claims, on Figure 4.2 we present overall translation stall time penalty for different empty page buffer size, in a system with 1GB of DRAM, running two different workloads. For low values we notice relatively stable level of total translation stall time. This indicates that the average eviction time is higher than the time in which TLBs exhaust a small supply of empty pages in DRAM. A scenario where one completed eviction unblocks one stalled translation, but shortly after that is followed by another similar translation-eviction pair makes the empty page buffer ineffective, and the changes in its size irrelevant.

On the other side, large empty page buffer allows the TLBs to invoke many more evictions before any of the translations stalls. However, once that happens penalty will be very high, due to a huge number of in-flight migrations and congested interconnect and memory. Measured total translation stall time for large empty page buffers shows that this tradeoff is not beneficial. Moreover, it is unaffordable sacrificing significant DRAM capacity to the empty page buffer, as it can cause other inneficiencies not related to the translation stall time.

Figure 4.2 shows that the two opposing causes for high overall translation time diminish their influence at around 8MB of empty page buffer size. The sweetspot effect is not overly dramatic, but enough to give us a reason to choose this value as fixed for the rest of our experiments.

Similarly to prioritizing writes in static profile-based policy, we also evaluate a modified LRU spill policy, where we select least recently written page for eviction, instead of least recently used. Again, we do that to explore if a decrease in PCM writes can outweigh potential performance degradation.

LRU spill policy suffers from similar drawbacks as the first touch policy — a decision to evict a page from DRAM, and make PCM its final and definitive host, is irreversible, and can be proven costly if suddenly its traffic increases at some later point. To better measure the amount of these wrong evictions and their negative effects, we made use of the profile information to more accurately "predict" expected traffic on a given page. Therefore, **spill profile-based policy** can either spare a page from eviction if its future traffic is high, or victimize it if it is low, regardless of its previous access count. That way, at any moment during the execution, this policy keeps in DRAM those pages that will have most accesses in the future, and use the profile to make ideal eviction decisions and achieve superior performance than LRU spill policy. Aside from being unpractical for use because of the need of a pre-run for generating a profile, this policy is technically hard to implement, because every eviction demands a comparison of each DRAM page statistics against the corresponding entry in the profile. However, as a "perfect spill policy" we can use it to estimate how well LRU spill policy performs.

Once again, for favoring PCM's lifetime instead of overall performance, we

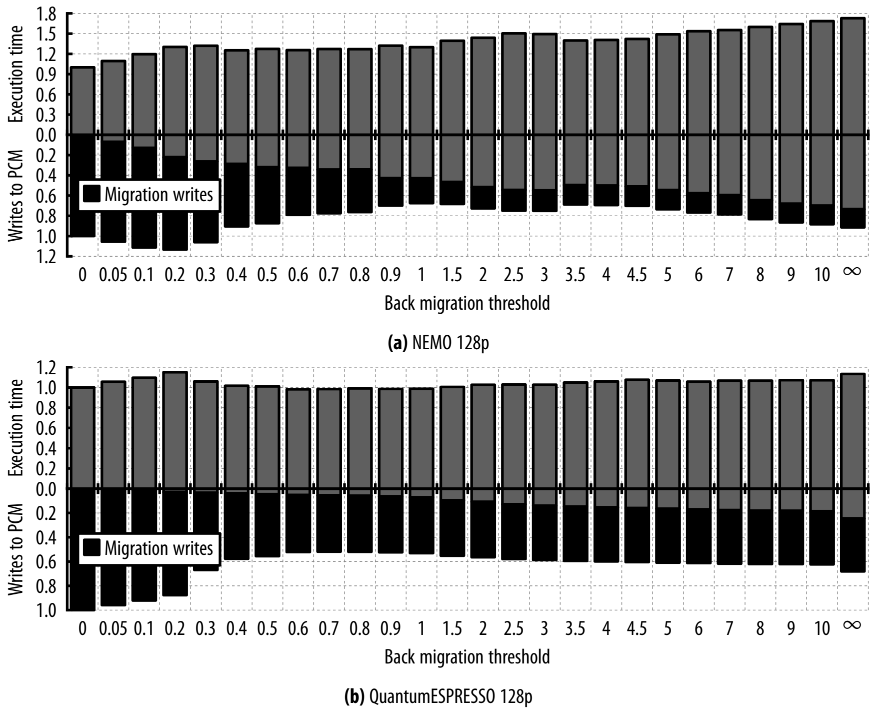


Figure 4.3: Back migration threshold analysis

evaluate modified spill profile-based policy, where we select a page for eviction based on its future write access count, instead of the total access count.

4.2.4 Dynamic page migration

Finally, a **Dynamic policy** introduces page migration in the other direction (from PCM to DRAM), denoted as back-migration. It is an extension of spill policy, so the same rules for eviction from DRAM still stand — whenever a page fault occurs, the system tries to allocate the page in the DRAM. In case an eviction is needed from DRAM, a page is selected in the same way as with LRU spill or spill profile-based policy.

The decision to back-migrate the page is made when the page is accessed in the PCM. When a page is first brought to the PCM we reset its access counter, regardless of how many times it was accessed in the DRAM. At the same time we keep track of the number of accesses for every page in the DRAM, as well as the average for all the pages ($n_{DRAMavg}$). When a page in PCM is accessed, we compare its access counter ($n_{accesses}$) with the average number of accesses to pages in DRAM. If (4.1) is satisfied, we migrate the page back to DRAM:

$$n_{accesses} > back\_migration\_threshold \times n_{DRAMavg} \tag{4.1}$$

Back migration threshold (BMT) is a value that controls the aggressiveness of migration triggering. If it is set to zero, a page is migrated as soon as it is touched in PCM, so the DRAM acts as a typical cache. In this case we expect good performance as the system tends to always move active pages to DRAM, but due to a large number of migrations, number of writes to PCM may go high. On the other hand, if BMT is set to infinity the page never gets migrated back, and then the policy is equivalent to LRU spill. In between those extremes we would like to search for values that give good performace and low number of PCM writes.

Figure 4.3 shows impact of changing BMT from 0 to infinity, on the performance and on the number of PCM writes, when executing two of our applications. Top part of the graph shows execution time normalized to an aggresive migration setup, when BMT is set to zero. Bottom part presents the number of writes to PCM normalized to the same setup. Since the aggresiveness of migrations directly influences number of writes to both memory modules, we separated PCM writes that are part of a migration, from those that are requested from the cores.

We can observe that in both cases the execution time is lowest when BMT is set to zero. This is expected, since in this case DRAM behaves like a cache, and the number of application accesses to PCM is close to zero. Same conclusions have been shown in similar architectures, when migrating pages to on-chip memories [82]. However, due to migrations from DRAM to PCM, the number of writes to PCM is high. As we increase BMT, two applications show different behaviour. However, we can spot a value for BMT of 1 as a rough minimum for the number of PCM writes, which also has a decent performance. In case of NEMO, performance degrades ~30%, but PCM writes decrease for the same value. In case of QuantumESPRESSO, performance stays on roughly the same level, while PCM writes decrease for almost 50%.

This gives us enough reason to further investigate two dynamic migration poli-

Table 4.2: Overview of data placement policies

Data placement policy	Profile based	Prioritize writes	Migrations		
			DRAM→PCM	PCM→DRAM	BMT
First touch	No	No	No	No	-
Static profile	Yes	No	No	No	-
Static profile(w)	Yes	Yes	No	No	-
Spill LRU	No	No	Yes	No	-
Spill LRU(w)	No	Yes	Yes	No	-
Spill profile	Yes	No	Yes	No	-
Spill profile(w)	Yes	Yes	Yes	No	-
Dyn perform	No	No	Yes	Yes	0.0
Dyn perform(w)	No	Yes	Yes	Yes	0.0
Dyn lifetime	No	No	Yes	Yes	1.0
Dyn lifetime(w)	No	Yes	Yes	Yes	1.0

cies: first, performance-oriented with BMT set to 0, and second, lifetime-oriented, with BMT set to 1. Similarly as with previous policies, we will also investigate the modification that takes into account only write accesses. That is, back-migration is considered only on a PCM write, and performed if the write count is greater than the average number of writes to the pages in DRAM, multiplied by BMT.

Table 4.2 summarizes all previously described data placement policies, and presents an overview of their respective key features — usage of the profile information, prioritization of writes, allowed migration directions between DRAM and PCM, and value of BMT (in case back migration is allowed).

4.3 Evaluation

Our investigation focuses on the changes in the execution time and in the number of writes to PCM as we gradually increase the size of DRAM, from 512MB up to the size that is larger than the total footprint of the application. We also try to investigate and explain performance differences as we configure memory management unit to allocate pages based on first-touch policy, profile-based policy and on dynamic policies. To evaluate the system, we use *TaskSim*, a trace-driven cycle-accurate CMP simulator validated against Cell BE [70].

Application	Domain	Footprint (GB)	Time
NEMO	Ocean modeling	6.54	≈ 2 s
CPMD	Computational chemistry	7.48	≈ 20 s
PEPC	Plasma physics	8.79	≈ 30 s
QuantumESPRESSO	Particle physics	19.75	≈ 40 s
GADGET	Astronomy and cosmology	57.43	≈ 1 m

Table 4.3: Overview of simulated applications, their total memory footprint and time of the simulated part.

The applications that we use for evaluation are listed in Table 4.3. They are production-level HPC codes that use MPI programming model, and we executed them with realistic input sets. These are representatives of scientific applications that are used in today's supercomputers [73].

To obtain the traces we implemented *MemTraceMPI*, a Valgrind [60] tool for tracing load and store instructions of an MPI application. It instruments all the executed instructions, outputting only information about the memory accesses: access type (load or store) and access size in bytes. This information allows for a detailed simulation of memory accesses. In order to simulate non-memory instructions, the tool records the number of instructions that are executed between two consecutive memory accesses. To preserve the dynamic nature of parallel application, *MemTraceMPI* detects calls to MPI functions and records their parameters. This allows us to simulate the communication and synchronization among MPI processes. Since the simulated system is a chip-multiprocessor, communication is performed using memory copies, and synchronization is achieved using standard SMP synchronization primitives.

The platform used for application tracing was MareNostrum, a supercomputer in Barcelona Supercomputing Center. It is a cluster of JS21 blades, each with 4 IBM Power PC 970MP 2.3 GHz processors. The 4 processors on a node shared 8 GB of RAM, and were connected to a high-speed Myrinet type M3S-PCIXD-2-I port, and two GigaBit Ethernet ports [81].

Large execution times of our applications led to hundreds of gigabytes of traces per application. To keep the trace files at a manageable size, and to maintain acceptable simulation time, we applied trace filtering against a simple configurable cache (512KB direct cache), included in *MemTraceMPI* tool. Only cache misses are written to the trace. Given that our simulated caches are going to have a larger size than the trace filter caches, we are certain that at least the same number of mem-

Table 4.4: Overview of simulated architecture parameters

L1	Capacity:	8 KB	**L2**	Capacity:	1 MB per core
	Word size:	8 B		Word size:	32 B
	Associativity:	4-way		Associativity:	32-way
	Latency:	2 cycles		Latency:	20 cycles
DRAM	CL-tRCD-tRP:	8-8-8	**PCM**	4× slower than DRAM	
TLB	Entries:	128	**MMU**	Page size:	8K

ory accesses will reach the main memory, as with unfiltered traces. This method has been previously applied in the work of Rico et al. [71] resulting with trace size reductions of up to 98%.

Even after filtering, simulation times were unacceptable (in the order of days). To further reduce the size of the traces, we exploited the fact that all applications iterate multiple times over the input set and included only a few iterations [13, 26].

4.3.1 Architecture parameters

Simulated architecture closely modeled architecture presented in Section 4.2.1, and Figure 4.1, with the most important parameters presented in the Table 4.4.

4.3.2 Results

Figure 4.4 compares execution times on a system with PCM memory only (t_{PCM}) and on a system with DRAM memory only (t_{DRAM}), both of which are of sufficient capacity to fit the entire application's working set. Vertical axis represents execution time normalized to t_{PCM}, and applications under study are aligned along the horizontal axis. We notice that only NEMO and CPMD experience changes in execution time that is equivalent to the performance difference between PCM and DRAM (4×). The rest of the applications put more stress on the interconnect than on the memory system. In those cases, the positive influence of the fast memory is diminished, as particularly is the case with PEPC, and, naturally, we cannot expect that a system with hybrid memory, regardless of the data placement policy, will give a performance improvement.

Figure 4.5 shows the main results of our simulations. Each subfigure presents one application, testing the effectiveness of our page placement policies, aligned on the horizontal axis. Top part of each subfigure shows *relative slowdown*, execution time normalized on a range from t_{DRAM} to t_{PCM} using (4.2).

$$relative\_slowdown = \frac{t_{exec} - t_{DRAM}}{t_{PCM} - t_{DRAM}} \tag{4.2}$$

Therefore, a relative slowdown with a value of 0 would represent a "perfect" data placement policy, that yields performance equal to the system with DRAM memory only. Conversely, a system with PCM memory only would give a relative slowdown of 1. On a hybrid memory system that we evaluate, using one of the proposed data placement policies we expect relative slowdown to be between 0 and 1.

Bottom part of each subfigure represents the number of writes to PCM, with a special segment showing writes caused by migrations, normalized to the total writes. Different bars enclosed by each data placement policy segment, depict changes in the aforementioned metrics as the size of DRAM in a system varies from minimal 512 MB, incrementing by a factor of 2 up to a size of the entire application footprint, where any placement policy would be obsolete, because all the data could fit in DRAM. On both vertical axes we imposed a cut-off at a value of 1, so that the low values remain clear. On any bar that is shortened because of this, we write its real value.

As expected, *First touch* policy, with its naive approach, experiences bad performance for all applications unless the size of the DRAM gets very large. Similar results stand for the number of PCM writes, with the only exception occurring when compared to the aggressive migration policies, that can direct huge write traffic to

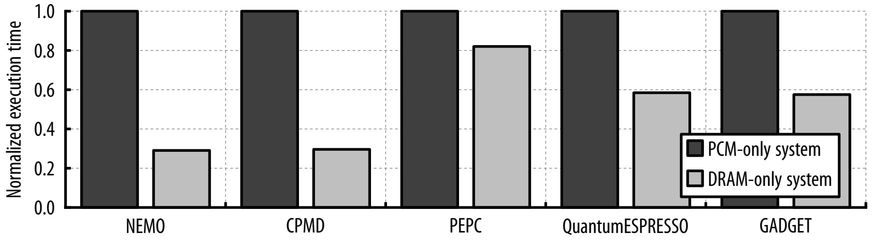


Figure 4.4: Performance comparison between PCM-only and DRAM-only system

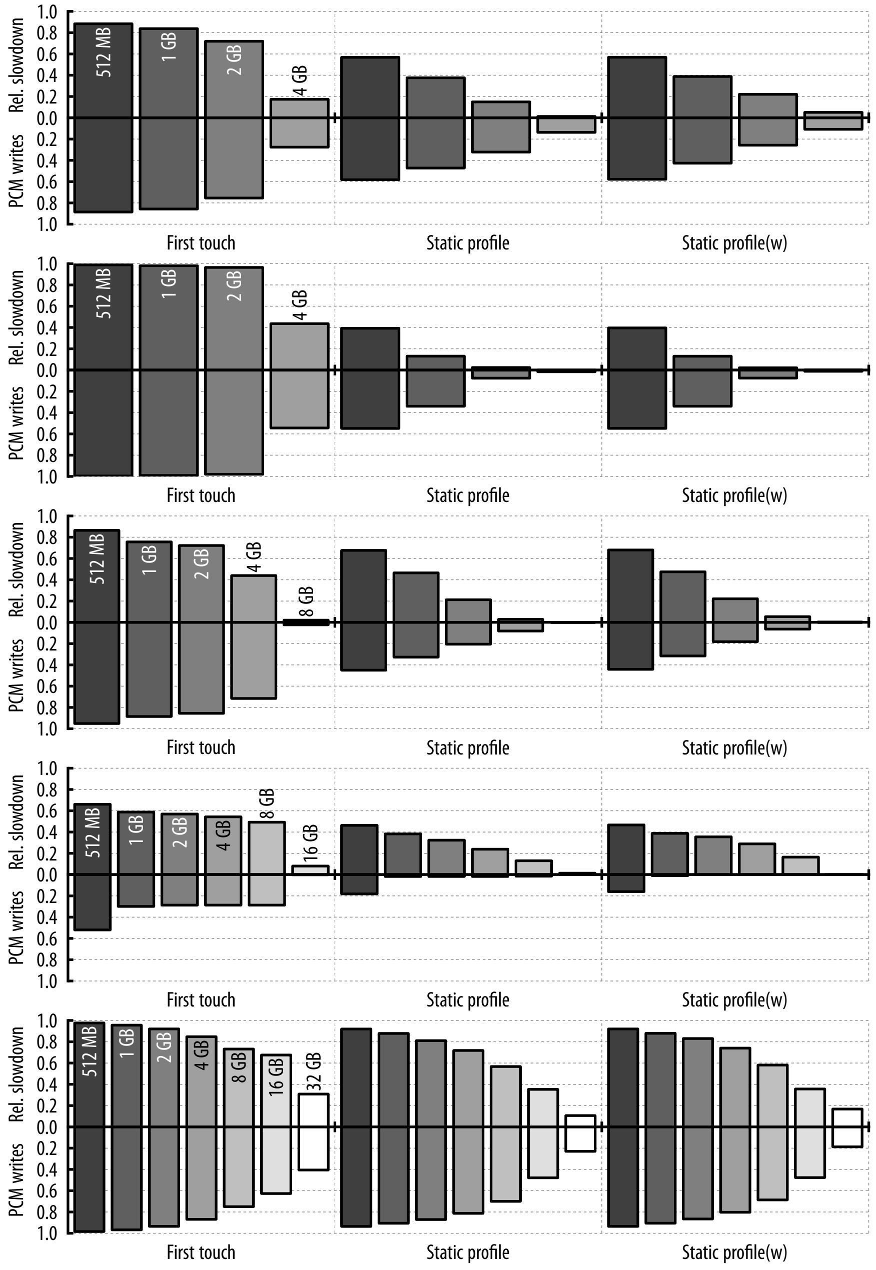


Figure 4.5: Overall performance and amount of PCM writes comparison of different data placement policies

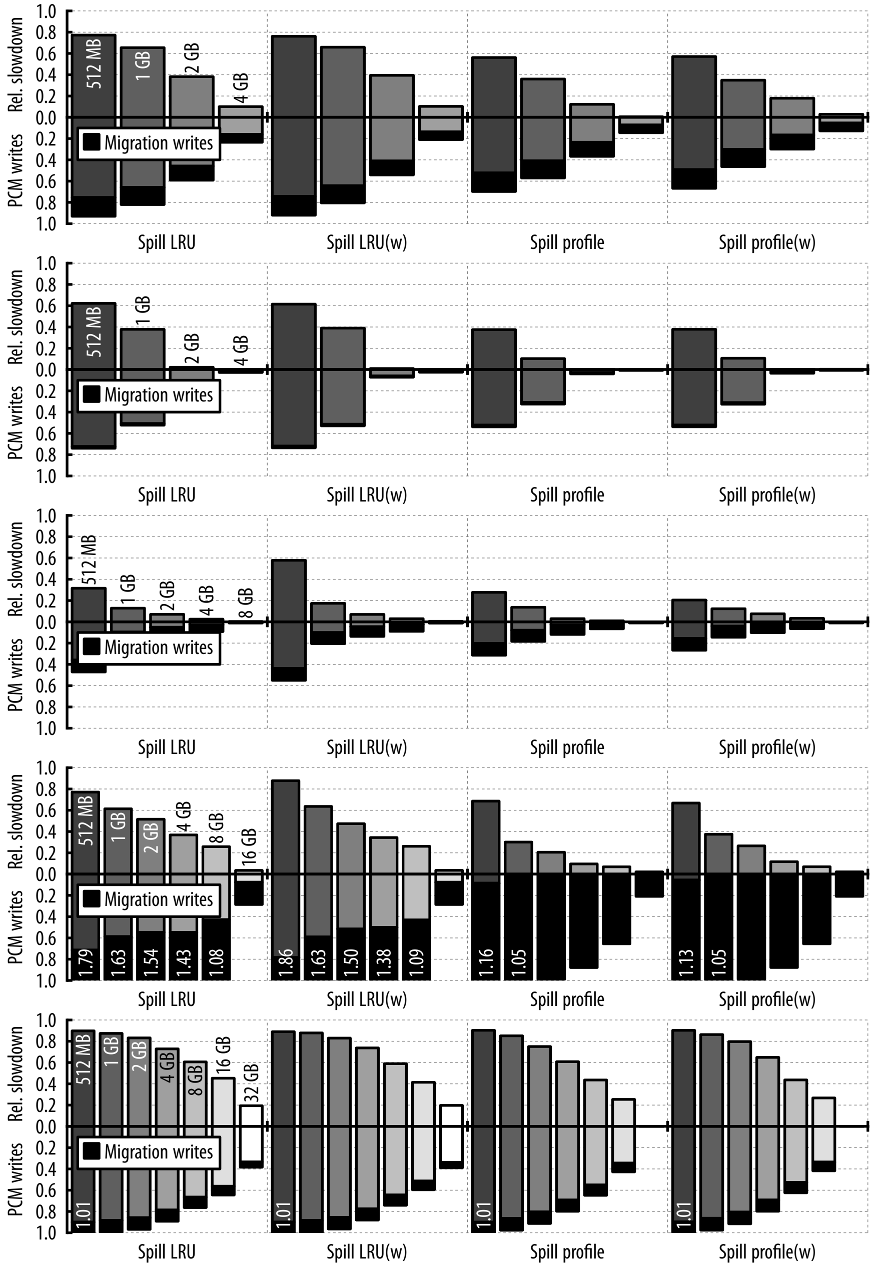


Figure 4.6: Overall performance and amount of PCM writes comparison of different data placement policies

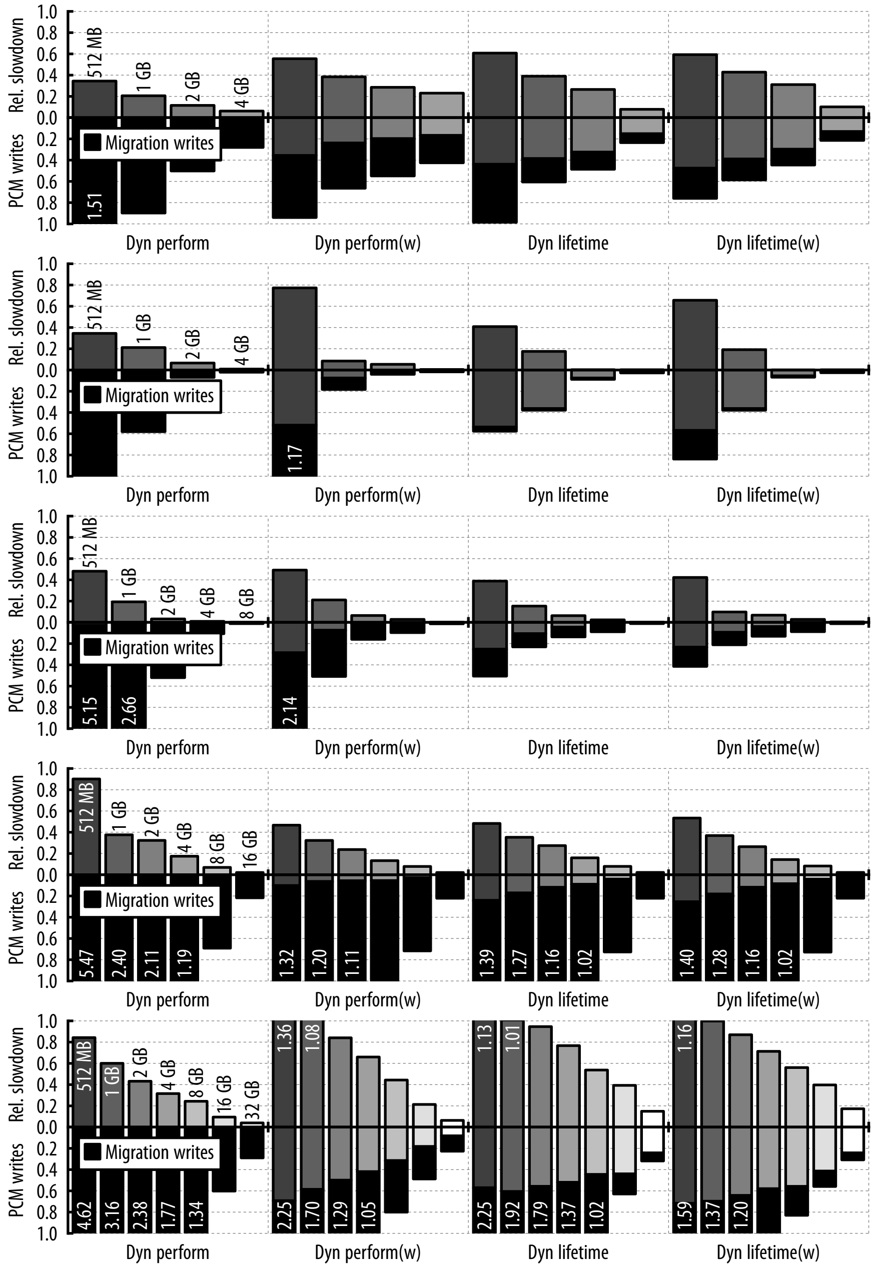


Figure 4.7: Overall performance and amount of PCM writes comparison of different data placement policies

PCM. The other two static policies perform better than the first touch, as they can exploit profile information for improving both of our relevant metrics. It is worth noticing that profile-based policy can reduce slowdown up to 40% using only the smallest DRAM size. Static profile policy that prioritizes writes (*Static profile(w)*) makes a very small, on chart almost invisible, tradeoff against regular static profile policy (*Static profile*), and a slight decrease in number of PCM writes pays with a small performance degradation.

LRU spill policy uses its migration capabilities to outperform first touch policy, but in most cases it is not better than static profile-based. Migrations that it executes do not provoke a significant number of extra writes to PCM, which is good, but also indicates that PCM write traffic is dominated by those pages that are previously evicted from DRAM, which is bad. Confirmation that selecting least recently used page is often a wrong decision comes when comparing it to spill profile policy, which performs in average 20% better, with less PCM writes. Similar to the static policies, spill profile-based policies show little difference if they prioritize writes.

Aggressive migration policy (*Dyn perform*) seems to dominate performance aspect of most applications, even for small DRAM sizes. However, its performance boosting capabilities can severely damage the lifetime of PCM, as the number of PCM writes can grow more than 5 times the baseline, in case of PEPC, QuantumESPRESSO and GADGET. We notice that almost 100% of the PCM writes, regardless of DRAM size, are a product of migrations. This time, modified aggressive dynamic policy (*Dyn perform(w)*) shows more obvious difference, mostly by significantly relieving PCM write traffic, without dramatically reducing performance.

Lifetime-oriented dynamic policy (*Dyn lifetime*) shows a nice balance between performance and number of writes, especially for the DRAM size of 1 GB. It is never too dominant in any aspect, but always among the best policies from what we evaluated. Performance-wise it shows an improvement of 20–60% over PCM-only system, and regarding PCM writes 40–60% (except QuantumESPRESSO and GADGET). It should be noted that lifetime-oriented policy has much more stable and predictable number of PCM writes than aggressive dynamic policy. When directly compared it can reduce PCM writes 20% to 10×, and only marginally degrade performance. Therefore, it might serve well in the wider spectrum of environments.

Dynamic policies, however, are not able to easily extract performance benefits with GADGET, the application with the largest footprint, unless DRAM is larger than 4 GB. When compared to any of the spill policies, it becomes clear that the dynamic policies cannot significantly reduce the amount of non-migration PCM writes. A closer insight in access profile information reveals that many pages in GADGET's working set experience similar traffic, and, therefore, cannot be easily divided into "hot" and "cold" segments, suitable for placing in DRAM or PCM, respectively. If DRAM is too small, this creates a ping-pong effect, where many pages are repeatedly migrated back and forth between DRAM and PCM.

4.4 Related work

Ramos et al. [69] proposed placing DRAM and PCM in a flat address space, with MC deciding about migrations between them. For facilitating this, they introduced a novel page ranking and migration policy. However, their target architecture did not include more than 8 cores, and they did not evaluate HPC applications for stressing memory bandwidth and capacity.

Qureshi et al. [68] placed DRAM in front of PCM as a cache with the intent to bridge the latency gap. Their work tackled PCM endurance issues by managing lazy-write organization, line-level writes, and wear-leveling. They show that a performance improvement of 3× is achievable with a DRAM buffer of only 3% size of PCM. By placing our memory modules in a flat address space instead, we try to increase the overall memory capacity, avoid negative impact of the DRAM cache on the workloads with low locality, and give the option of further upgrading the memory system with a module of different characteristics than DRAM or PCM.

Lee et al. [44] explored another hybrid memory organization with PCM and DRAM as a buffer, and concluded that PCM's long latencies, high energy, and finite endurance can be effectively mitigated. These effective buffer organizations and partial writes make PCM competitive with DRAM at current technology nodes. Moreover, proposed solutions are area neutral, which is a critical constraint in memory manufacturing.

A position paper from Hewlett-Packard [56] argues for the OS support in hybrid memory systems with the combination of DRAM and flash (or PCM) memory. The paper focuses on server workloads, but does not provide a detailed evaluation.

4.5 Summary

There is no *silver bullet* memory technology that allows high bandwidth, high density and low latency at the same time. Hence, many research efforts have focused on designing hybrid memory systems with different types of memory modules that in combination offer the characteristics that are needed. In this work, we looked at one such system that consists of a small and fast DRAM memory, and a large and slower PCM memory. We have focused on the problem of page placement and page migration in such system, and, to the best of our knowledge, we have performed first detailed analysis of several algorithms for performing this task. Using a set of High Performance Computing applications we have analyzed how the design parameters affect both performance of the system and the lifetime of the PCM memory.

Besides analyzing existing page placement algorithms, such as first touch or profile-based policies, we have also developed dynamic algorithms for placement and migration based on LRU spilling and back-migration using either aggressive (for performance) or more conservative (for PCM lifetime) policy.

Our analysis has started off with an expected result: if the only aim is to optimize for performance than the page placement and migration policy needs to be as aggressive as possible and needs to utilize DRAM memory as much as the capacity allows. Our results show that a system with only 1 GB of DRAM and an aggressive dynamic policy is only 20–60% slower than the system that has maximum DRAM capacity.

However, the performance of the aggressive dynamic policy comes at a price in the number of writes to PCM. These writes are costly, and they reduce the lifetime of the memory, so we have looked at other policies that aim to reduce their number, while keeping performance at an acceptable level. We have shown that these policies (called *Dynamic lifetime* in our analysis) when compared to the aggressive dynamic policy can reduce the number of PCM writes from 20% to 10×, while reducing performance up to 10%.

In conclusion, we have shown that page placement is a very important aspect of a system with hybrid memory, and the choice of the policy should be taken seriously and should depend on the objectives that the system architect has set.

Chapter 5

Limpio — LIghtweight MPI instrumentatiOn

5.1 Introduction

Computer architecture research often involves measuring various hardware-related statistics on a given platform during workload executions. The ability to easily setup experiments, and process collected results with little overhead, becomes very important when the researcher needs to repeat the experiments many (hundreds of) times in different execution environments. Thus, the choice of profiling tool, suitable for the problem under study, can be a significant research decision.

Profiling tools typically target two categories of users. First, software engineers, that are interested in application-related statistics, such as time consumption breakdown for each function in the code. They can use this information as an indicator on where to prioritize their code optimization efforts. Second, computer architects, that look for statistics about hardware performance counters, for having a better insight to the system from an architectural perspective. Their goal is to better understand *how well the hardware executes the workload*, instead of *how well the application is* *written.*

Both types of profiling tools collect their target statistics by introducing instrumentation routines in key points in the application. That typically requires manual or automatic modification of the code, and recompiling, or at least relinking application with the instrumentation library. While this is a trivial step in application development, hardware engineers are usually provided only with the application executable, without access to its source code, or details about the compilation process. Profiling can then only be done with tools that directly instrument the binary, or use wrapper interposition libraries.

Modern high-performance computing (HPC) clusters execute parallel applications typically written around an inter-process communication library, OpenMP or MPI. Therefore, communication function calls are the most obvious candidates for the instrumentation points. However, even on a 1000+ core machine, many applications can execute for hours or days, calling communication functions countless times. This raises two main concerns. First, instrumentation routines need to be lightweight, to avoid introducing overhead with respect to native execution, and that way distort time-related measurements, or have side-effects to cache or memory usage. Second, a timestamped trace of all communication events can become larger than what the processing tools can handle, or even larger than the available storage space on a shared cluster[1]. In such cases user might want to explicitly select the instrumentation points or choose to trace only a short execution segment.

Tools for post-mortem analysis, with their complexity, can impose a long learning curve to the beginners. Many times users find it easier to express the requirements of the analysis with short snippets of code, than to spend significant effort learning the features and coping with versatility of such tools.

In this appendix we present Limpio — LIghtweight MPI instrumentatiOn framework. Limpio enables the following design goals:

- Users can profile the application without recompiling. Instrumentation is done by dynamic linking.
- Limpio core is lightweight — it introduces no overhead to the native execution. Users themselves are responsible for the overhead of the instrumentation routines.

[1]Some of the applications we analyze produce traces of over 500 GB

- Limpio is customizable — users can write instrumentation routines highly specific to the analysis requirements.
- Limpio is extensible — users can invoke external tools from within instrumented calls.

5.2 Background

In order to parallelize numerical computation, scientific HPC applications divide and distribute input data over a large number of processes. Then, through a series of iterations and inter-process communication steps, they combine intermediate calculations into a final result. Therefore, scientific HPC applications naturally follow repetitive patterns, so characterizing their behavior in a few iterations of the main loop is equivalent to characterizing their entire execution [67]. Similarly, most of the processes execute the same algorithm on different data, so the behavior of a few processes can well represent the behavior of the entire application.

Most production MPI applications are by default dynamically linked against MPI library on a given system[2]. Although preinstalled shared libraries are typically found and loaded by the operating system, user can also create and preload his own library instead. By creating a wrapper for each standard library function, user creates a mechanism to inject his own instrumentation routines before and after the library call, without recompiling or relinking the original application, and without disrupting native application behavior and functionalities.

5.3 Architecture

Figure 5.1 presents a sequence diagram of an instrumented MPI call in the Limpio framework, and shows the interaction between the application, user-defined instrumentation routines, and standard MPI library. Limpio exposes two levels of interfaces — one to the application, and other to the user functions. From the application perspective, Limpio provides a set of wrapper functions for intercepting

[2]In order to use static linking, user needs to explicitly specify an appropriate flag when compiling the application, which is rarely done.

all MPI calls. In wrapper functions, Limpio first invokes a user-defined function that instruments the start of the MPI call. Then, the execution is passed to the corresponding function from the MPI library. Finally, after the MPI call completes, Limpio invokes a user function for instrumenting the end of the MPI call (see Figure 4.1).

Limpio, by itself, performs no application profiling. Instead, it allows users to create their own analysis tools by writing instrumentation routines. Limpio hooks the instrumentation routines with relevant MPI wrapper functions, and creates a shared library object, preloaded before the execution. The instrumentation routines, when invoked by the application MPI calls, measure statistics relevant to the user. Gathered data can be stored as a timestamped event trace for post-mortem analysis, or processed online (while the application runs) for producing a summary of statistics after the application completes.

```c
#include <stdio.h>
#include "limpio.h"

static void start(mpi_function_id_t mpi_function_id) {
   printf("Start %s\n", get_mpi_fn_name(mpi_function_id));
}
static void end(mpi_function_id_t mpi_function_id) {
   printf("End %s\n", get_mpi_fn_name(mpi_function_id));
}
__attribute__ ((constructor))
static void mpilog_init(void) {
   set_start_hook(&start);
   set_end_hook(&end);
}
```

Listing 5.1: Example Limpio tool that logs start and end of MPI calls

Listing 5.1 shows an example of a Limpio tool that logs start and end of MPI calls. Function `mpilog_init()`, invoked when the library is initialized, uses functions from Limpio framework to define MPI instrumentation routines — `start()` for instrumenting MPI call entry, and `end()` for MPI call exit. These routines print a message on the standard output, when invoked by the MPI call from the application. Similarly to `mpilog_init()`, there may exist `mpilog_fini()`, invoked after the application is completed, for outputting any statistics accumulated during the execution.

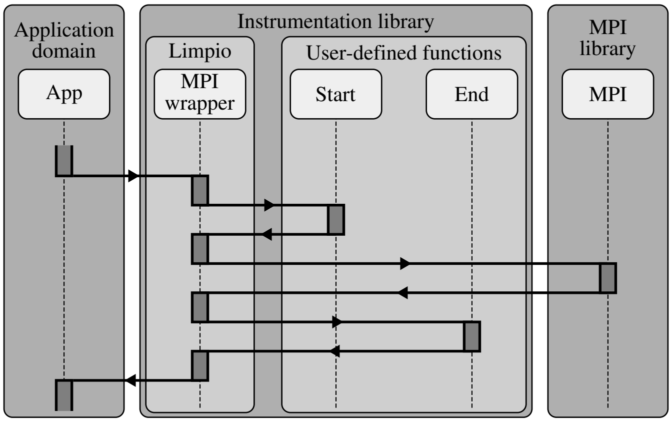


Figure 5.1: Limpio instrumentation — sequence diagram

5.4 Case studies

Limpio was developed for characterizing large-scale HPC applications. In Computer Architectures group at Barcelona Supercomputing Center, Limpio is regularly used to analyze production HPC applications running on MareNostrum [8] supercomputer. MareNostrum is one of six Tier-0 HPC systems in PRACE [64], containing 3,056 compute nodes, each with two Intel Sandy Bridge-EP E5-2670 sockets with eight cores operating at 2.6 GHz.

Access to MareNostrum allowed us to run experiments using up to several thousand cores. Because of a large number of application processes and long execution time of production HPC applications, it was essential to prepare the instrumentation tools for processing massive amounts of data. Limpio provided sufficient room for customization, and we used it to create several tools, each targeting different aspect of application characterization. In the following sections, we describe some of the Limpio tools, and illustrate their usage for profiling of ALYA application [65].

5.4.1 MPI profiler

MPI profiler tool produces basic statistics about MPI calls — their per-process and total number, and the accumulative time they consume relative to the total application execution. MPI profiler calculates these statistics using online processing — on each instrumented call, it increments the counter of the corresponding MPI function, and accumulates the duration of the call. After the execution, the tool generates a summary, as presented in Table 5.1, without producing any traces or intermediate results during the application run.

Table 5.1 gives a breakdown of MPI calls (columns) over application processes (rows). Each entry in the table shows how many times one MPI function is invoked by a particular process, as well as the aggregate duration of these calls relative to the total application execution time (shown in parentheses). Last column summarizes MPI communication for each process by showing total number of MPI calls, and their relative duration. Last row gives an overview of each MPI function usage combined over all the processes.

Profile of MPI calls is our first step in HPC application analysis. It gives an overview of the MPI calls that the application uses, and verifies that most of the processes have similar behavior from MPI communication perspective. Table 5.1 shows that ALYA has a master-worker architecture, where first process, substantially different from the rest, has a role of master, while other 255 processes serve as workers, and have similar MPI communication profiles.

The same analysis could well be performed with some of the existing tools, that can generate full timestamped MPI event traces, and, in a post-mortem analysis, summarize event count and duration. However, with that approach, HPC applications would generate very large MPI traces, which would make post-mortem analysis prohibitively slow, and could easily surpass the available disk quota on a shared HPC system. For this reason we chose online analysis to produce this kind of statistics.

	MPI_Send	MPI_Recv	MPI_Allreduce	MPI_Barrier	MPI_Bcast	MPI_Allgatherv	MPI_Gatherv	MPI_Sendrecv	Total
Process #1	7650(0.4%)	2314(0.1%)	74757(90.0%)	63(0.0%)	28(0.0%)	2(0.2%)	1(0.0%)	0(0.0%)	84819(92.9%)
Process #2	9(0.2%)	30(6.2%)	74757(30.3%)	63(0.0%)	28(0.0%)	2(0.1%)	1(0.0%)	140635(12.8%)	215529(51.4%)
Process #3	9(0.2%)	30(6.2%)	74757(41.2%)	63(0.0%)	28(0.0%)	2(0.1%)	1(0.0%)	84381(3.3%)	159275(52.9%)
Process #254	9(0.3%)	30(6.1%)	74757(40.0%)	63(0.0%)	28(0.1%)	2(0.1%)	1(0.0%)	56254(1.6%)	131148(49.7%)
Process #255	9(0.3%)	30(6.1%)	74757(40.1%)	63(0.0%)	28(0.1%)	2(0.1%)	1(0.0%)	56254(2.0%)	131148(50.2%)
Process #256	9(0.3%)	30(6.1%)	74757(38.9%)	63(0.0%)	28(0.1%)	2(0.1%)	1(0.0%)	112508(2.6%)	187402(49.6%)
Total	9967(0.3%)	9967(6.1%)	19137792(31.4%)	16128(0.0%)	7168(0.1%)	512(0.1%)	256(0.0%)	39489832(7.6%)	58672646(47.2%)

Table 5.1: MPI profile, ALYA application, 256 processes

5.4.2 Computation to communication ratio

Another important metric for analysis of HPC applications is computation to communication ratio. When inter-process communication dominates over effective computation time, especially on high number of processes, gains in parallel performance can diminish and be prevailed by the execution cost. Limpio MPI profiler measures relative MPI communication time — the time spent in all MPI library function calls, relative to the total execution time. Increase in relative MPI communication time, with the increase in the number of processes, indicates limited application scalability.

Figure 5.2 shows relative MPI communication time for ALYA application, for various number of processes. The figure separates MPI communication time of the master process, and the average of the worker processes. The results show how MPI communication starts being dominant for high number of processes, which decreases the relative time spent in computational segments. Therefore, the increase in parallelism is not paid off by the appropriate performance improvement (for a 256× increase in number of processes — from 4 to 1,024, we measure only 86× speedup).

This experiment required analysis of nine executions of the application (from 4 to 1,024 processes). By performing online analysis with Limpio MPI profiler, we avoided time-consuming process of collecting a large timestamped event trace and its post-processing. Moreover, optimized instrumentation routines introduced negligible overhead in the instrumented execution, which is important when measuring any time-related statistics.

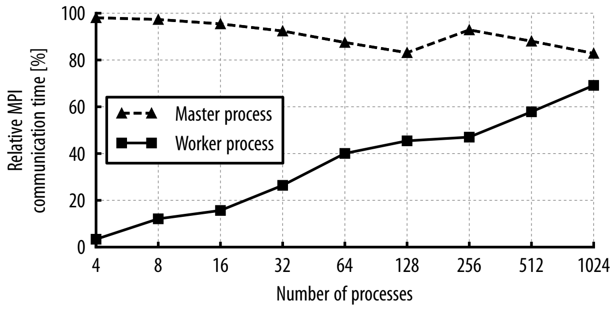


Figure 5.2: Relative MPI communication time, ALYA, 4–1,024 processes

5.4.3 Tracing and visualization

Profilers typically summarize their target statistics in a concise table, sacrificing data about their evolution and variability throughout the execution. Observing the changes in measured metrics over time gives insight in application-specific patterns, iterative behavior, correlation between measured metrics, or transient bottlenecks in the system. For that, we need a trace of timestamped events, triggered by an MPI call, where the current state of user-defined statistics is measured and saved.

In order to avoid producing large trace files, and to make their analysis or visualization feasible, Limpio provides a mechanism to selectively disable instrumentation in specific MPI function calls, by setting the environment variable `MPIINSTR_EXCLUDE`. Once the users obtain the MPI profile, they can disable instrumentation in the most frequent MPI calls. That way the granularity of the instrumentation points is coarsened, and the trace size is reduced. In our example, ALYA produced a profile (Table 5.1) where `MPI_Allreduce` and `MPI_Sendrecv` constitute the vast majority of all MPI calls. Disabling instrumentation of these two calls (Listing 5.2) reduces the trace to the size that can be handled by the tools for visualization or post-mortem analysis.

```
...
export MPIINSTR_EXCLUDE=MPI_Allreduce,MPI_Sendrecv
...
```

Listing 5.2: Excluding instrumentation of specific MPI functions

Traditional tools for trace visualization have to be compatible with the trace format generated by instrumentation tools. In Limpio, users can define the trace format in the instrumentation routines, and then visualize the trace using any external general-purpose visualization tool or library.

Figure 5.3 presents an example of a visualization of an MPI communication trace. It is obtained by running ALYA, instrumented and traced with Limpio (with most frequent MPI calls excluded), and rendered using Matplotlib. The figure confirms application repetitive behavior, with `MPI_Barrier` as the call that precedes and follows each iteration.

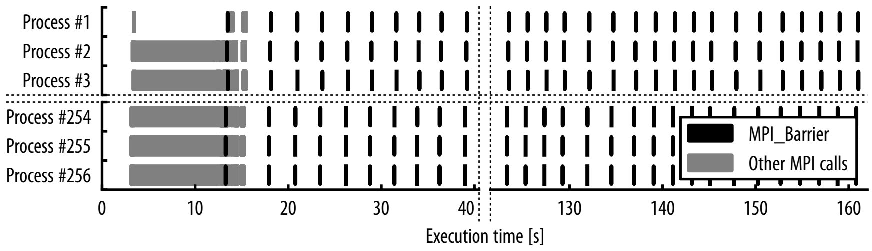


Figure 5.3: MPI call visualization, ALYA application, 256 processes

5.4.4 External instrumentation tools

The ability to recognize boundaries of application iterations, allows us to use Limpio for invoking external tools, and using them in a precisely defined segment of the application execution. For example, with the information extracted from a trace of timestamped events (Section 5.4.3), Limpio tool can be configured to detect iterations of the main loop, that are good representative of the overall application behavior. In that segment, it can invoke tools for instrumenting application at the instruction-level granularity, such as Pin [53] or Valgrind [60].

An example code for that functionality is given in Listing 5.3. In function `start()`, it is determined if the execution has reached target iteration (nth call of a target MPI function), and in that case, external instrumentation tool is attached to a running process. Similarly, in function `end()`, when the target iteration completes, Limpio sends a signal to the external tool, and detaches it from the process.

```
#include <signal.h>
...
static void start(mpi_function_id_t mpi_function_id) {
   if (is_instrumentation_segment_starting()) {
      sprintf(attach_instumentation_command, "%s -pid %ld",
         path_to_instrumentation_binary, (long)getpid());
      system(attach_command);
   }
}
static void end(mpi_function_id_t mpi_function_id) {
   if (is_instrumentation_segment_ending())
      kill(getpid(), SIGUSR1);
}
...
```

Listing 5.3: Attaching and detaching an external instrumentation tool

Combining Limpio with external instrumentation tools has multiple uses. First, instruction-level instrumentation usually introduces large performance overhead, and it is reasonable to apply it only on a short execution segment. Second, if external instrumentation is used for generating instruction or memory traces, their size can be controlled by modifying the length of the tracing segment.

5.5 Related work and tools

Tool mpiP [80] produces MPI function profiles similar to Limpio MPI profiler. It does not require recompiling, but it needs linking to the mpiP library. For profiling a segment of the execution, user needs to change application source code.

Score-P [41] provides an infrastructure for profiling, tracing, and online analysis of HPC applications. Instead of direct instrumentation, it can use sampling, but users need to recompile the application with Score-P instrumentation command.

Extrae [24] is a tool for tracing application performance data. Extrae can instrument applications based on a wide range of programming models (MPI, OpenMP, CUDA, OpenCL, etc.). With various interposition mechanisms for injecting probes into the target application, it provides many instrumentation options for HPC application analysis. Extrae uses clustering to detect iterative patterns and to choose the most interesting regions to present to the analyst [50]. However, Extrae cannot instrument a user-defined subset of MPI calls, nor it can be used for online analysis. Extrae is not designed for triggering calls to external profiling tools.

5.6 Summary

This appendix presents Limpio, a framework for profiling of MPI applications. Limpio overrides standard MPI functions, and executes instrumentation routines before and after the selected MPI calls. Users themselves can write and customize the instrumentation routines to fit the requirements of the analysis. Limpio can invoke external application profiling tools, and can switch between various tools in a single execution. It can also generate application traces of timestamped events that can be visualized by general-purpose visualization tools or libraries. Limpio is regularly used in Barcelona Supercomputing Center for instrumentation of large-scale HPC applications.

Chapter 6

Conclusions

Most computing systems are heavily dependent on their main memories, as their primary storage, or as an intermediate cache for slower storage systems (HDDs). The capacity of memory systems, as well as their performance, have a direct impact on overall computing capabilities of the system, and are also major contributors to its initial and operating costs.

DRAM is the mainstream technology for main memories of modern computers. Its dominant position is based on a simple production process, low cost per unit of storage, performance that could match the requirements of CPUs, and high reliability. Scaling trend of DRAM technology has slowed in the past decade, creating a disparity between the CPU and main memory performance, also known as *the memory wall*. It is estimated that DRAM improvements in the future will be even more impeded by inherent limitations in DRAM memory design.

Modern parallel architectures, such as High-Performance Computing (HPC) clusters and manycore solutions, create even more stress on their memory systems. It is not trivial to estimate memory requirements that these systems will have in the future, and if DRAM technology would be able to meet them, or we would need to look for a novel memory solution.

Emerging memory technologies try to overcome the limitations of DRAM, and answer the ever-growing demands of future computer applications. Many of them are still in the research phase, or with a fundamental drawbacks that makes them inferior to DRAM. However, their variety and plenty of space for their research can lead to the emergence of a DRAM successor. More, their usability in hybrid memory systems already improves on their overall value in the memory technology landscape.

In this thesis we attempted to give insight in the most important technological challenges that future memory systems need to address, in order to meet the requirements of future users and their applications, in manycore and HPC context. In Chapter 2 we described the limitations of DRAM, as the dominant technology in today's main memory systems, that may impede performance or increase cost of future systems. We discussed some of the emerging memory technologies, and by comparing them with DRAM, we tried to estimate their potential usage in future memory systems.

As its first contribution presented in Chapter 3, the thesis evaluates the requirements of manycore scientific applications, in terms of memory bandwidth and footprint, and estimates how these requirements may change in the future. As a result, we conclude that the limitations in DRAM scalability will not have negative effects on manycore systems in the next several years.

With this evaulation in mind, and as the second contribution presented in Chapter 4, we propose a hybrid memory solution that employs DRAM and PCM, as well as several page placement and page migration policies, to bridge the gap between fast and small DRAM and larger but slower non-volatile memory. Our results show that the hybrid memory system with dynamic page migration and limited DRAM capacity, can achieve performance that is comparable to a hypothetical, hard to implement, DRAM-only system.

Finally, our third contribution, presented in Chapter 5 describes a tool that is designed and developed over the course of this PhD, and continues to be used in Heterogeneous Computer Architectures group in Barcelona Supercomputing Center. Limpio — a LIghtweight MPI instrumentatiOn framework, that provides an interface for low-overhead instrumentation and profiling of MPI applications with user-defined routines.

Bibliography

[1] G. Abandah and E. Davidson. Configuration independent analysis for characterizing shared-memory applications. In *Intl. Parallel Processing Symp.*, pages 485–491, Mar 1998.

[2] K. Abe, H. Noguchi, E. Kitagawa, N. Shimomura, J. Ito, and S. Fujita. Novel Hybrid DRAM/MRAM Design for Reducing Power of High Performance Mobile CPU. In *IEEE International Electron Devices Meeting (IEDM)*, 2012.

[3] A. Agarwal, L. Bao, J. Brown, B. Edwards, M. Mattina, C.-C. Miao, C. Ramey, and D. Wentzlaff. Tile processor: Embedded multicore for networking and multimedia. In *Hot Chips*, Aug 2007.

[4] J.-H. Ahn, B.-H. Jeong, S.-H. Kim, S.-H. Chu, S.-K. Cho, H.-J. Lee, M.-H. Kim, S.-I. Park, S.-W. Shin, J.-H. Lee, B.-S. Han, J.-K. Hong, P. Moran, and Y.-T. Kim. Adaptive self refresh scheme for battery operated high-density mobile dram applications. In *Solid-State Circuits Conference, 2006. ASSCC 2006. IEEE Asian*, pages 319–322, Nov 2006.

[5] S. R. Alam, R. F. Barrett, J. A. Kuehn, P. C. Roth, and J. S. Vetter. Characterization of scientific workloads on systems with multi-core processors. In *IEEE Intl. Symp. on Workload Characterization*, pages 225–236, Oct 2006.

[6] C. Augustine, A. Raychowdhury, D. Somasekhar, J. Tschanz, K. Roy, and V. De. Numerical analysis of typical stt-mtj stacks for 1t-1r memory arrays. In *Elec-*

tron Devices Meeting (IEDM), 2010 IEEE International, pages 22.7.1–22.7.4, Dec 2010.

[7] D. Baglee and R. Parker. Trench capacitor for high density dynamic ram, July 10 1990. US Patent RE33,261.

[8] Barcelona Supercomputing Center. MareNostrum III System Architecture. http://www.bsc.es/marenostrum-support-services/mn3, 2013.

[9] F. Bedeschi, R. Fackenthal, C. Resta, E. Michele Donze, M. Jagasivamani, E. Buda, F. Pellizzer, D. Chow, A. Cabrini, G. M. Angelo Calvi, R. Faravelli, A. Fantini, G. Torelli, D. Mills, R. Gastaldi, and G. Casagrande. A multi-level-cell bipolar-selected phase-change memory. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, ISSCC '08, pages 428–430, 2008.

[10] M. Bhadauria, V. M. Weaver, and S. A. McKee. Understanding parsec performance on contemporary cmps. In *IEEE Intl. Symp. on Workload Characterization*, Oct 2009.

[11] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb. Die stacking (3D) microarchitecture. In *IEEE Micro*, pages 469–479, 2006.

[12] D. Burger, J. R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In *Intl. Symp. on Computer Architecture*, pages 78–89, 1996.

[13] M. Casas, R. M. Badia, and J. Labarta. Automatic phase detection and structure extraction of MPI applications. *International Journal of High Performance Computing Applications*, 24(3):335–360, 2010.

[14] L. Chua. Memristor — The missing circuit element. *Circuit Theory, IEEE Transactions on*, 18(5):507–519, Sep 1971.

[15] Z. Cvetanovic and R. E. Kessler. Performance analysis of the alpha 21264-based compaq es40 system. In *Intl. Symp. on Computer Architecture*, pages 192–202, 2000.

[16] J. T. Daly. ADTSC Nuclear Weapons Highlights: Facilitating High Throughput ASC Calculations. *Technical Report LALP-07-041, Los Alamos National Laboratory, Los Alamos, NM, USA*, 2007.

[17] J. T. Daly, L. A. Pritchett-Sheats, and S. E. Michalak. Application MTTFE vs. Platform MTTF: A Fresh Perspective on System Reliability and Application Throughput for Computations at Scale. In *Workshop on Resiliency in High Performance Computing*, 2008.

[18] Z. Diao, Z. Li, S. Wang, Y. Ding, A. Panchula, E. Chen, L.-C. Wang, and Y. Huai. Spin-transfer torque switching in magnetic tunnel junctions and spin-transfer torque random access memory. *Journal of Physics: Condensed Matter*, 19(16):165209, 2007.

[19] B. Dieny, V. S. Speriosu, S. S. P. Parkin, B. A. Gurney, D. R. Wilhoit, and D. Mauri. Giant magnetoresistive in soft ferromagnetic multilayers. *Phys. Rev. B*, 1991.

[20] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. Linpack. www.netlib.org/linpack.

[21] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.*, 2002.

[22] K. Ferreira, R. Riesen, R. Oldfield, J. Stearley, J. Laros, K. Pedretti, T. Kordenbrock, and R. Brightwell. Increasing Fault Resiliency in a Message-Passing Environment. *Sandia National Laboratories, Tech. Rep.*, 2009.

[23] R. F. Freitas and W. W. Wilcke. Storage-class memory: the next storage system technology. *IBM J. Res. Dev.*, 52(4):439–447, July 2008.

[24] H. Gelabert and G. Sánchez. Extrae User Guide Manual for version 2.2.0. *Barcelona Supercomputing Center (BSC)*, 2011.

[25] M. Gill, T. Lowrey, and J. Park. Ovonic unified memory - a high-performance nonvolatile memory technology for stand-alone memory and embedded applications. In *Solid-State Circuits Conference, 2002. Digest of Technical Papers. ISSCC. 2002 IEEE International*, volume 1, pages 202–459 vol.1, Feb 2002.

[26] J. Gonzalez, J. Gimenez, M. Casas, M. Moreto, A. Ramirez, J. Labarta, and M. Valero. Simulating whole supercomputer applications. *Micro, IEEE*, 31(3):32–45, 2011.

[27] C. Ho, C.-L. Hsu, C.-C. Chen, J.-T. Liu, C.-S. Wu, C.-C. Huang, C. Hu, and F.-L. Yang. 9nm half-pitch functional resistive memory cell with <1 ua programming current using thermally oxidized sub-stoichiometric wox film. In

Electron Devices Meeting (IEDM), 2010 IEEE International, pages 19.1.1–19.1.4, Dec 2010.

[28] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. A Novel Nonvolatile Memory with Spin Torque Transfer Magnetization Switching: Spin-RAM. In *IEEE International Electron Devices Meeting*, 2005.

[29] A. A. Hwang, I. Stefanovici, and B. Schroeder. Cosmic Rays DonâĂŹt Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[30] R. Iyer, H. Wang, and L. Bhuyan. Design and analysis of static memory management policies for CC-NUMA multiprocessors. *College Station, TX, USA, Tech. Rep*, 1998.

[31] B. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, Burlington, MA, USA, 2008.

[32] X. Jian, H. Duwe, J. Sartori, V. Sridharan, and R. Kumar. Low-power, Low-storage-overhead Chipkill Correct via Multi-line Error Correction. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[33] L. Jiang, R. Ye, and Q. Xu. Yield enhancement for 3D-Stacked memory by redundancy sharing across dies. In *IEEE/ACM Intl. Conf. on Computer-Aided Design*, pages 230–234, 2010.

[34] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. Power7: IBM's next-generation server processor. *IEEE Micro*, 30:7–15, 2010.

[35] J. A. Katine, F. J. Albert, R. A. Buhrman, E. B. Myers, and D. C. Ralph. Current-Driven Magnetization Reversal and Spin-Wave Excitations in Co /Cu /Co Pillars. *Phys. Rev. Lett.*, 2000.

[36] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 10(4):338–359, Nov. 1992.

[37] C. Kim, D. Kang, H. Kim, C. Park, D. SOHN, Y. Lee, S. Kang, H. Oh, and S. Cha. Magnetic Random Access Memory, 2013.

[38] H. Kim, S. Kang, D. SOHN, D. Kim, and K. Lee. Magneto-resistive memory device including source line voltage generator, 2013.

[39] J. Kim and M. Papaefthymiou. Dynamic memory design for low data-retention power. In D. Soudris, P. Pirsch, and E. Barke, editors, *Integrated Circuit Design*, volume 1918 of *Lecture Notes in Computer Science*, pages 207–216. Springer Berlin Heidelberg, 2000.

[40] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping Bits in Memory without Accessing them: An Experimental Study of DRAM Disturbance Errors. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014.

[41] A. Knüpfer, C. Rössel, D. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. TschÃijter, M. Wagner, B. Wesarg, and F. Wolf. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for HPC 2011*, pages 79–91. Springer Berlin Heidelberg, 2012.

[42] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25:21–29, 2005.

[43] S. Kottapalli and J. Baxter. Nehalem-EX CPU architecture. In *Hot Chips*, Aug 2009.

[44] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[45] M.-J. Lee, Y. Park, B.-S. Kang, S. eon Ahn, C. Lee, K. Kim, W. Xianyu, G. Stefanovich, J.-H. Lee, S.-J. Chung, Y.-H. Kim, C.-S. Lee, J.-B. Park, and I.-K. Yoo. 2-stack 1d-1r cross-point structure with oxide diodes as switch elements for high density resistance ram applications. In *Electron Devices Meeting, 2007. IEDM 2007. IEEE International*, pages 771–774, Dec 2007.

[46] D. Lewis and H.-H. Lee. Architectural evaluation of 3d stacked rram caches. In *3D System Integration, 2009. 3DIC 2009. IEEE International Conference on*, pages 1–4, Sept 2009.

[47] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu. Raidr: Retention-aware intelligent dram refresh. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[48] L. Liu, Z. Li, and A. H. Sameh. Analyzing memory access intensity in parallel programs on multicore. In *Supercomputing*, pages 359–367, 2008.

[49] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: Saving dram refresh-power through critical data partitioning. *SIGPLAN Not.*, 46(3):213–224, Mar. 2011.

[50] G. Llort, J. Gonzalez, H. Servat, J. Gimenez, and J. Labarta. On-line detection of large-scale parallel application's structure. In *International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–10. IEEE, 2010.

[51] G. Loh. 3D-Stacked Memory Architectures for Multi-core Processors. In *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pages 453–464, 2008.

[52] G. H. Loh. 3D-stacked memory architectures for multi-core processors. In *Intl. Symp. on Computer Architecture*, pages 453–464, 2008.

[53] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6), June 2005.

[54] The MareNostrum system architecture. http://www.bsc.es/plantillaA.php?cat_id=200.

[55] A. Mericas. Performance monitoring on the POWER5™ microprocessor. In *Performance Evaluation and Benchmarking*, pages 247–266. CRC Press, 2005.

[56] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for NVM+DRAM hybrid main memory. In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, pages 14–14, Berkeley, CA, USA, 2009. USENIX Association.

[57] G. E. Moore et al. Progress in digital integrated electronics. *IEDM Tech. Digest*, 11, 1975.

[58] R. C. Murphy and P. M. Kogge. On the memory access patterns of supercomputer applications: Benchmark selection and its implications. *IEEE Trans. Computers*, 2007.

[59] R. Nebashi, N. Sakimura, H. Honjo, S. Saito, Y. Ito, S. Miura, Y. Kato, K. Mori, Y. Ozaki, Y. Kobayashi, N. Ohshima, K. Kinoshita, T. Suzuki, K. Nagahara, N. Ishiwata, K. Suemitsu, S. Fukami, H. Hada, T. Sugibayashi, and N. Kasai. A 90nm 12ns 32Mb 2T1MTJ MRAM. In *IEEE International Solid-State Circuits Conference*, 2009.

[60] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.

[61] H. Noguchi, K. Kushida, K. Ikegami, K. Abe, E. Kitagawa, S. Kashiwada, C. Kamata, A. Kawasumi, H. Hara, and S. Fujita. A 250-MHz 256b-I/O 1-Mb STT-MRAM with Advanced Perpendicular MTJ Based Dual cell for Nonvolatile Magnetic Caches to Reduce Active Power of Processors. In *Symposium on VLSI Technology (VLSIT)*, 2013.

[62] H. Oh. Resistive Memory Device, System Including the Same and Method of Reading Data in the Same, 2014.

[63] Z. Pajouhi, X. Fong, and K. Roy. Device/Circuit/Architecture Co-design of Reliable STT-MRAM. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, 2015.

[64] Partnership for Advanced Computing in Europe (PRACE). Prace research infrastructure. http://www.prace-ri.eu.

[65] Partnership for Advanced Computing in Europe (PRACE). *Unified European Applications Benchmark Suite*, July 2013.

[66] M. Pavlovic, Y. Etsion, and A. Ramirez. On the memory system requirements of future scientific applications: Four case-studies. *IEEE Workload Characterization Symposium*, 0:159–170, 2011.

[67] R. Preissl, T. Kockerbauer, M. Schulz, D. Kranzlmuller, B. Supinski, and D. J. Quinlan. Detecting patterns in MPI communication traces. In *International Conference on Parallel Processing (ICPP)*. IEEE, 2008.

[68] M. K. Qureshi, V. Srinivasan, and J. 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.

[69] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 85–95, New York, NY, USA, 2011. ACM.

[70] A. Rico, F. Cabarcas, A. Quesada, M. Pavlovic, A. J. Vega, C. Villavieja, Y. Etsion, and A. Ramirez. Scalable simulation of decoupled accelerator architectures. *Universitat Politecnica de Catalunya, Tech. Rep. UPC-DACRR-2010-14*, 2010.

[71] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramirez, and M. Valero. On the simulation of large-scale architectures using multiple application abstraction levels. *ACM Trans. Archit. Code Optim.*, 8(4):36:1–36:20, Jan. 2012.

[72] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, P. Dubey, S. Junkins, A. Lake, R. Cavin, R. Espasa, E. Grochowski, T. Juan, M. Abrash, J. Sugerman, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *IEEE Micro*, 29(1):10–21, 2009.

[73] A. D. Simpson, M. Bull, and J. Hill. *Identification and Categorisation of Applications and Initial Benchmarks Suite*, 2008. www.prace-project.eu.

[74] J. Spong, Speriosu, R. E. Fontana, M. M. Dovek, and T. Hylton. Giant Magnetoresistive Spin Valve Bridge Sensor. *IEEE Transactions on Magnetics*, 1996.

[75] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453:80–83, May 2008.

[76] D. B. Strukov and R. S. Williams. Four-dimensional address topology for circuits with stacked multilayer crossbar arrays. *Proceedings of The National Academy of Sciences*, 106:20155–20158, 2009.

[77] S. Tyson, G. Wicker, T. Lowrey, S. Hudgens, and K. Hunt. Nonvolatile, high density, high performance phase-change memory. In *Aerospace Conference Proceedings, 2000 IEEE*, volume 5, pages 385–390 vol.5, 2000.

[78] A. M. Vasilevskiĭ, V. A. Volpyas, A. B. Kozyrev, and G. A. Konoplev. Effect of UV radiation on the relaxation characteristics of ferroelectric thin-film capacitors. *Technical Physics Letters*, 34:561–564, July 2008.

[79] R. Venkatesan, S. Herr, and E. Rotenberg. Retention-aware placement in dram (rapid): software methods for quasi-non-volatile dram. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 155–165, Feb 2006.

[80] J. Vetter and C. Chambreau. mpiP: Lightweight, Scalable MPI Profiling, 2005.

[81] D. Vicente and J. Bartolome. BSC-CNS Research and Supercomputing Resources. In *High Performance Computing on Vector Systems 2009*, pages 23–30. Springer, 2010.

[82] C. Villavieja, Y. Etsion, A. Ramirez, and N. Navarro. FELI: HW/SW support for on-chip distributed shared memory in multicores. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 282–294, Berlin, Heidelberg, 2011. Springer-Verlag.

[83] R. Waser, R. Dittmann, G. Staikov, and K. Szot. Redox-Based Resistive Switching Memories–Nanoionic Mechanisms, Prospects, and Challenges. *Advanced Materials*, 21(25-26):2632–2663, 2009.

[84] R. Waser, R. Dittmann, G. Staikov, and K. Szot. Redox-based resistive switching memories âĂŞ nanoionic mechanisms, prospects, and challenges. *Advanced Materials*, 21(25-26):2632–2663, 2009.

[85] D. H. Woo, N. H. Seong, D. L. Lewis, and H.-H. S. Lee. An optimized 3D-Stacked memory architecture by exploiting excessive, high-density tsv bandwidth. In *Symp. on High-Performance Computer Architecture*, pages 1–12, 2010.

[86] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Intl. Symp. on Computer Architecture*, pages 24–36, 1995.

[87] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, Mar. 1995.

[88] Y. Xie. Modeling, Architecture, and Applications for Emerging Memory Technologies. *Design Test of Computers, IEEE*, 28(1):44–51, 2011.

[89] J. J. Yang, M. D. Pickett, X. Li, D. A. A. Ohlberg, D. R. Stewart, and R. S. Williams. Memristive switching mechanism for metal/oxide/metal nanodevices. *NATURE NANOTECHNOLOGY*, 3(7):429–433, July 2008.

[90] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. Energy reduction for stt-ram using early write termination. In *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, pages 264–268, Nov 2009.