PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

EVALUATION OF SYSTEM-LEVEL IMPACTS OF A PERSISTENT MAIN MEMORY ARCHITECTURE

TACIANO PEREZ

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: CÉSAR A. F. DE ROSE

Porto Alegre
2012

P438e Perez, Taciano

Evaluation of system-level impacts of a persistent main memory architecture / Taciano Perez. – Porto Alegre, 2012.

105 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS.

Orientador: Prof. Dr. César A. F. De Rose.

1. Informática. 2. Arquitetura de Computador. 3. Memória de Computador. I. De Rose, César A. F. II. Título.

CDD 004.22

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada ***"Evaluation of System-Level Impacts of a Persistent Main Memory Architecture"***, apresentada por Taciano Dreckmann Perez como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído, aprovada em 15/03/2012 pela Comissão Examinadora:

[illegible]

Prof. Dr. César Augusto Fonticielha De Rose - Orientador — PPGCC/PUCRS

[illegible]

Prof. Dr. Ney Laert Vilar Calazans - — PPGCC/PUCRS

[illegible]

Prof. Dr. Rodolfo Jardim de Azevedo - — UNICAMP

Homologada em 22 / 05 / 2012, conforme Ata No. 011 pela Comissão Coordenadora.

[illegible]

Prof. Dr. Paulo Henrique Lemelle Fernandes
Coordenador.

"Disks are a hack, not a design feature.

The time and complexity penalty for using disks is so severe that nothing short of enormous cost-differential could compel us to rely on them. Disks do not make computers better, more powerful, faster, or easier to use. Instead, they make computers weaker, slower, and more complex. They are a compromise, a dilution of the solid-state architecture of digital computers. (...)

Wherever disk technology has left its mark on the design of our software, it has done so for implementation purposes only, and not in the service of users or any goal-directed design rationale."

Alan Cooper, About Face 2.0 [18]

ACKNOWLEDGEMENTS

This work was crafted along 4 years across two organizations: Hewlett-Packard, where I've been working during the last decade, and PUCRS, where this research took place. I have received the gifts of a creative environment, financial support, and computing infrastructure from both these organizations, and without any of them, this work would have never been possible.

The idea for this research was inspired by a presentation on Memristor made by Greg Astfalk in 2008, who has since then been very supportive of this work, discussing ideas, reviewing reports and articles, and helping me connect to people who contributed to the final result. I cannot find words to thank him enough for his support.

César A. F. De Rose, my advisor, encouraged me to come up with an original research theme, and led me wisely and patiently through the challenge that is conducting such an endeavour. He taught me how to successfully navigate in an academic environment, and whenever I faced an obstacle, he was there to help me remove it.

Ney Laert Vilar Calazans acted as a co-advisor, and enlightened me on several discussions regarding hardware modeling and simulation, particularly concerning the energy model employed in this experiment. He also contributed with thorough and detailed reviews of early texts.

Several people on Hewlett-Packard supported this work in different forms. Eduardo Arioli, my manager, always made efforts to provide me basic means to craft this research. Marcelo Thielo believed in this work before I did, and encouraged me to move forward towards this particular theme. Katia Saikoski, Karin Becker and Luis Saraiva provided assistance for me to make the first moves towards an academic research.

The team of the PUCRS high-performance computing lab (LAD - Laboratório de Alto Desempenho) provided critical help for the experiment, particularly Raquel Dias who set up the whole environment. The staff of PPGCC office, especially Thiago Lingener and Régis Escobal da Silva, were of great assistance during the 4 years that this work was developed.

There are several important people in my life that contributed directly or indirectly to this work: friends, family, colleagues. I won't cite any names, for they know who they are. I thank you very much for being there.

This work is dedicated, in loving memory, to my grandmother Iris Vezzani Perez.

EVALUATION OF SYSTEM-LEVEL IMPACTS OF A PERSISTENT MAIN MEMORY ARCHITECTURE

ABSTRACT

For almost 30 years, computer memory systems have been essentially the same: volatile, high speed memory technologies like SRAM and DRAM used for cache and main memory; magnetic disks for high-end data storage; and persistent, low speed flash memory for storage with low capacity/low energy consumption requirements such as embedded/mobile devices. Today we watch the emergence of new non-volatile memory (NVM) technologies that promise to radically change the landscape of memory systems. In this work we assess system-level latency and energy impacts of a computer with persistent main memory using PCRAM and Memristor. The experimental results support the feasibility of employing emerging non-volatile memory technologies as persistent main memory, indicating that energy improvements over DRAM should be significant. This study has also compared the development and execution of applications using both a traditional filesystem design and a framework specific of in-memory persistence (Mnemosyne). It concludes that in order to reap the major rewards potentially offered by persistent main memory, it is necessary to take new programming approaches that do not separate volatile memory from persistent secondary storage.

Keywords: Non-Volatile Memory; Persistent Main Memory; PCRAM; RRAM; Memristor.

AVALIAÇÃO DE IMPACTOS EM NIVEL DE SISTEMA DE UMA ARQUITETURA DE MEMÓRIA PRINCIPAL PERSISTENTE

RESUMO

Por cerca de 30 anos, os sistemas de memória computacional têm sido essencialmente os mesmos: tecnologias de memória volátil de alta velocidade como SRAM e DRAM utilizadas para caches e memória principal; discos magnéticos para armazenamento persistente; e memória flash, persistente e de baixa velocidade, para armazenamento com características de baixa capacidade e baixo consumo de energia, tais como dispositivos móveis e embarcados. Hoje estão emergindo novas tecnologias de memória não-volátil, que prometem mudar radicalmente o cenário de sistemas de memória. Neste trabalho são avaliados impactos (em nível de sistema) de latência e energia supondo um computador com memória principal persistente usando PCRAM e Memristor. Os resultados experimentais suportam a viabilidade de se empregar tecnologias emergentes de memória não-volátil como memória principal persistente, indicando que as vantagens de consumo de energia com relação a DRAM devem ser significativas. Esse estudo também compara o desenvolvimento de aplicações usando tanto uma abordagem tradicional usando sistema de arquivos quanto utilizando um framework específico para persistência em memória. Conclui-se que, para colher os principais benefícios potencialmente oferecidos por memória principal persistente, é necessário utilizar novas abordagens de programação que não estabelecem uma separação entre memória volátil e armazenamento secundário.
Palavras-chave: Memória não-volátil; Memória principal persistente; PCRAM; RRAM; Memristor.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

API	*Application Programming Interface*
ATA	*Advanced Technology Attachment*
BSBI	*Blocked Sort-Based Index*
CAS	*Column Address Strobe*
CB	*Copy Buffer*
CPU	*Central Processing Unit*
DIMM	*Dual In-line Memory Module*
DRAM	*Dynamic Random-Access Memory*
ETTNW	*Estimated Time To Next Write*
FC	*Fibre Channel*
HDD	*Hard Disk Drive*
JEDEC	*Joint Electron Devices Engineering Council*
LOC	*Lines Of Code*
MRAM	*Magnetoresistive Random-Access Memory*
NVM	*Non-Volatile Memory*
OS	*Operating System*
PCRAM	*Phase-Change Random-Access Memory*
PCM	*Phase-Change Memory*
PRAM	*Phase-change Random-Access Memory*
RAM	*Random-Access Memory*
RAS	*Row Address Strobe*
RBT	*Red-Black Tree*
RRAM	*Resistive Random-Access Memory*
SAS	*Serial Attached SCSI*
SATA	*Serial Advanced Technology Attachment*
SCSI	*Small Computer Systems Interface*
SRAM	*Static Random-Access Memory*
STL	*Standard Template Library*
STT-RAM	*Spin-Torque Transfer Random-Access Memory*
TML	*Tunelling Magneto-Resistance*
UML	*Unified Modeling Language*

TABLE OF CONTENTS

1. INTRODUCTION

For almost 30 years the memory hierarchy used in computer design has been essentially the same: volatile, high speed memory technologies like SRAM and DRAM used for cache and main memory; magnetic disks for high-end data storage; and persistent, low speed flash memory for storage with low capacity/low energy consumption requirements, such as embedded/mobile devices [31].

Today we watch the emergence of new memory technologies under research that promise to change significantly the landscape of memory systems. Non-Volatile Memory (NVM) technologies such as Phase-Change RAM (PCRAM), Magnetic RAM (MRAM) and Resistive RAM (RRAM) will possibly enable memory chips that are non-volatile, require low-energy and have density and latency closer to current DRAM chips [13].

Even though most of these technologies are still in early prototyping stages, their expected properties in terms of area, latency, energy and endurance are already estimated. Based on these estimations, several papers have been published proposing practical applications for these new NVM technologies. These papers provide initial insight on what the impacts on memory systems should be if such technologies evolve to be competitive in the market. A survey of these studies was presented in a previous work [52].

The memory system is one of the most critical components of modern computers. It has attained a high level of complexity due to the many layers involved in memory hierarchy: application software, operating system, cache, main memory and disk. Any relevant change in one level of the memory hierarchy (such as the introduction of the aforementioned NVM technologies) can potentially impact several other layers and overall system behavior. For this reason, it is essential to consider the different possible scenarios for using these new memory technologies from a broad perspective.

The are several research projects in motion to create "Universal Memory" [52], a single memory technology that could replace all different memory technologies of today (SRAM, DRAM, disk and flash). It is not yet clear if the Universal Memory vision can be realized by NVM technologies under research, but the creation of a byte-addressable, non-volatile solid state memory could make a significant amount of persistent main memory available to computer systems, allowing to consolidate these two different levels of the storage hierarchy - main memory and persistent storage - into a single level, something that has never been possible before (at least not in mass scale). The advent of main memory as the primary persistent storage can affect deeply the complete computing stack, including application software, operating system, busses, memory system and its interaction with other devices, such as processors and I/O adapters [58, 68]. In order to fully assess system-wide impacts in latency, energy, heat, space and cost, it is required to take into account all these different layers when modeling or simulating a hypotethical computer system with persistent main memory.

In this work we evaluate the impacts on latency and energy of a computer system with persistent main memory, through the use of simulation. We simulate scenarios of systems using different technologies for main memory (DRAM, PCRAM and Memristor) using a full-system architectural simulator (Virtutech Simics [42]), and compare the execution of worloads using both a typical scenario using the filesystem as storage backend and an API specific for in-memory persistence (Mnemosyne [69]).

Preliminary results were published in [51].

This work is organized in the following way:

- Chapter 2 presents the **current technologies** used in memory and storage hierarchy.
- Chapter 3 discusses the **current organization** of computer memory.
- Chapter 4 presents some of the **emerging non-volatile memory technologies**.
- Chapter 5 discusses the impacts of these emerging memory technologies concerning how they can contribute to create computers with **persistent main memory**.
- Chapter 6 surveys **tools for modeling/simulating** systems with hypothetical new architectures, especially where memory subsystem is concerned.
- Chapter 7 describes the **evaluation methodology**, **results** and **analysis** concerning the execution of workloads in a simulated persistent main memory computer.
- Chapter 8 discusses potential **perspectives for future research** towards computation with persistent main memory.
- Chapter 9 **concludes** this study summarizing the main findings of this research.

2. CURRENT MEMORY TECHNOLOGIES

In order to discuss the emergent NVM technologies, it is necessary to start by summarizing the current state of memory systems. This is a well-known subject, and we have extracted most of the information present in this section from the comprehensive textbook *Memory Systems: Cache, DRAM, Disk* by Bruce Jacob, Spencer W. Ng and David T. Wang [31] and Ulrich Drepper's online paper *What Every Programmer Should Know About Memory* [25].

2.1 Memory Hierarchy

An ideal memory system would be fast, cheap, persistent and big (highly dense). Until now, all known memory/storage technologies address only some of these characteristics. Static RAM (SRAM) is very fast, but it's expensive, has low density and is not persistent. Dynamic RAM (DRAM) has better densities and is cheaper (but still expensive) at the cost of being a little slower, and it's not persistent as well. Disks are cheap, highly dense and persistent, but very slow. Flash memory is between DRAM and disk; it is a persistent solid-state memory with higher densities than DRAM, but its write latency is much higher than the latter.

Fortunately, it is possible to design a memory system that incorporates all these different technologies in a single **memory hierarchy**. Such hierarchy allows the creation of a system that approaches the performance of the fastest component, the cost of the cheapest component and energy consumption of the most power-efficient component. This is possible due to a phenomenon known as **locality of reference**, so named because memory references tend to be localized in time and space. This can be so summarized:

- If you use something once, you are likely to use it again (temporal locality).
- If you use something once, you are likely to use its neighbor (spatial locality) [31].

This phenomenon can be exploited by creating different memory levels; the first levels are smaller and more expensive, but have fastest access, and the other layers are progressively bigger and cheaper, but have larger access times. Appropriate heuristics are then applied to decide which data will be accessed more often and place them at the first levels, and move the data down on the hierarchy as it ceases to be used frequently. Figure 2.1 depicts the concept of memory hierarchy.

For at least the last 30 years, we have been building memory hierarchies that rely on SRAM for caches, DRAM for main memory and disk or flash as persistent storage. In the next sections we will describe these technologies in more detail.

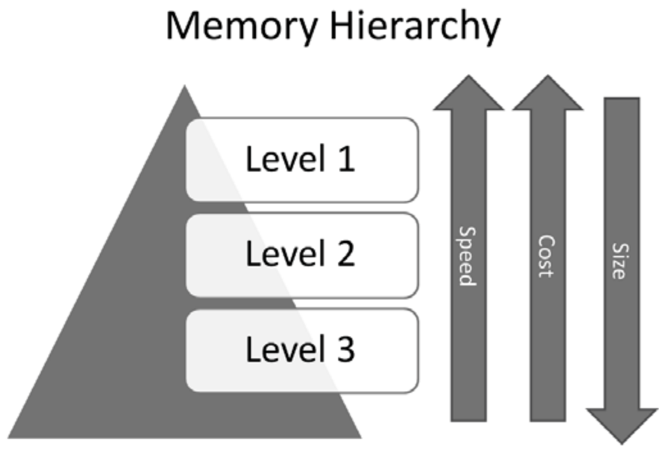


Figure 2.1: Diagram illustrating the concept of memory hierarchy.

2.2 Processor Cache: SRAM

Caches are used to store frequently used information close to who needs it. They are used at several layers of computer architecture, but probably the most important are processor caches. In order to explore locality of reference, we need to place memory with very fast access close to the processor. The result is that processor caches reside between the processor and main memory, as depicted in Figure 2.2a.

Processor caches usually are **transparent caches**, meaning that the same addressing space is shared between the cache and the backing storage. When the processor asks for the contents of a specific memory address, these contents are first searched in the cache. If they are found (a **cache hit**), they are immediately returned. If they are not found (a **cache miss**), the memory system goes on to the next hierarchy level (e.g., main memory) and fetches the result. If there is a high probability that the returned result will be used again in the future, it is stored in the cache as well. Diverse heuristics can be used to manage cache content [31].

The speed of a cache is typically in inverse proportion to its size. This happens because the more addresses a cache can contain, more search needs to be done in order to determine whether there is a cache hit or miss. For this reason, modern computers have more than one cache level, frequently up to 3 levels. It is also common to have a separate Level 1 cache for instructions and another for data. Figure 2.2b shows a processor with such an organization. Since the same data can reside in different cache levels (as well as in the backing storage), transparent caches use techniques to ensure consistency between copies in these different levels.

Finally, it is important to mention that modern computers frequently have multi-processor, multi-core and multi-thread architectures. Usually each core has its own Level 1 instruction and data cache, and all cores in the same processor share Level 2 and 3 caches. Such organization is depicted in Figure 2.2c. Please notice that this is not a rule: several architectures can explore different organizations. This example is provided just as an illustration of an architecture commonly found today [25].

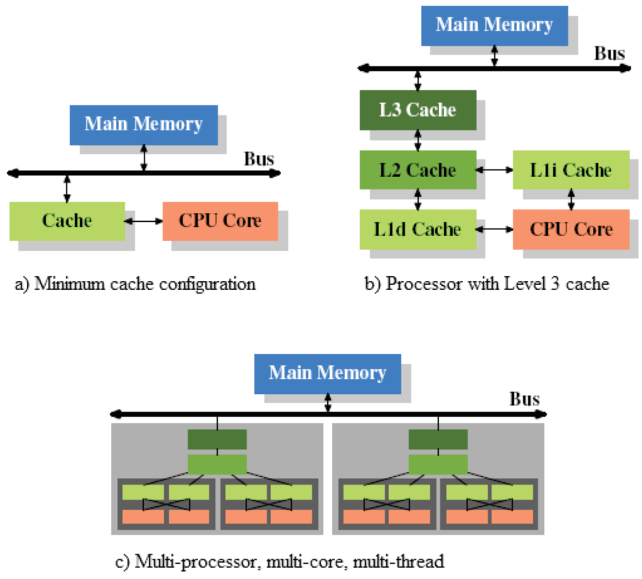

Main Memory
Bus
Cache
CPU Core
a) Minimum cache configuration
Main Memory
Bus
L3 Cache
L2 Cache
L1i Cache
L1d Cache
CPU Core
b) Processor with Level 3 cache
Main Memory
Bus
c) Multi-processor, multi-core, multi-thread

Figure 2.2: Processor caches: logical organization. Extracted from Drepper, 2007 [25].

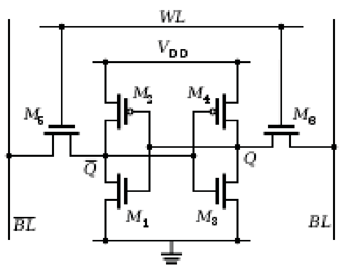


Figure 2.3: 6-T SRAM cell. Extracted from Drepper 2007 [25].

As we can see, processor caches have some distinctive requirements:

1. They need to be very fast.
2. They can be small (in fact, to a certain extent, they need to be small to save search time).
3. Since they are small, it is acceptable if their cost/bit is expensive when compared to other components of memory hierarchy.

Today, the technology that best matches this set of requirements is **Static Random Access Memory (SRAM)**.

The most common SRAM cell design uses 6 transistors as shown in Figure 2.3. The cell core is comprised of the four transistors M1 to M4 which form two cross-coupled inverters. They have two stable states, representing 0 and 1. The state is stable as long as power on Vdd is available. If access to the state of the cell is needed the word access line WL is raised. This makes the state of the cell immediately available for reading on BL and $\bar{BL}$. If the cell state must be overwritten the BL and $\bar{BL}$ lines are first set to the desired values and then WL is raised [25].

SRAM has some properties that are important for our discussion:

- The cell state is available for reading almost immediately once the word access line WL is raised, making its access very fast.
- The SRAM cell is complex, requiring six transistors in the most common design. This takes up space, and for this reason SRAM has lower densities than other technologies (such as DRAM).
- SRAM is more expensive in a cost/bit comparison.
- Maintaining the state of the cell requires constant power.

Essentially, SRAM is currently preferred for caches because it is very fast; the fact that it is less dense and more expensive is not as relevant for processor caches as access speed.

2.3 Main Memory: DRAM

A computer program is comprised of a set of instructions, and it manipulates data while it is being executed. The place where both instructions and data are loaded during program execution is **main memory**. We have already seen that some of the most frequently used data are kept in cache to lower access time, but most of the memory addressing space used by a program usually is stored in main memory.

Almost every computer uses **virtual memory**, a technique that allows for an addressing space greater than the actual main memory capacity. When there is a request for memory allocation beyond main memory's boundaries, **page swapping** is used to temporarily send to the backing storage (usually disk or flash) a set of memory pages that is not immediately needed and reclaim their space. When those pages are needed again, they are fetched from the backing storage in exchange of another set of pages, and so on. Since this is an expensive process, page swapping is avoided whenever possible.

Based on this understanding, we can devise some requirements for main memories:

1. They need to be big enough to hold most of the data needed by a program to execute (to avoid paying too often the performance price of page swapping).
2. They can be slower than caches, since the most frequently used data are already being cached.
3. Their cost/bit needs to be lower than caches, since they need more capacity.

The technology that best fits into this description today is **Dynamic Random Access Memory (DRAM)**.

A typical DRAM cell has one transistor and one capacitor, as shown is Figure 2.4a. It keeps its state in capacitor C. The transistor M is used to provide access to the state. To read the state of the cell the access line AL is raised; this either causes a current to flow on the data line DL or not, depending on the charge in the capacitor. To write to the cell the data line DL is appropriately set and then AL is raised for a time long enough to charge or drain the capacitor. A DRAM cell

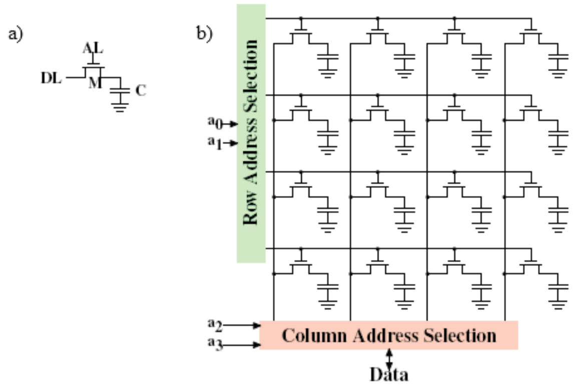


Figure 2.4: a) DRAM cell and b) DRAM array organization. Extracted from Drepper 2007 [25].

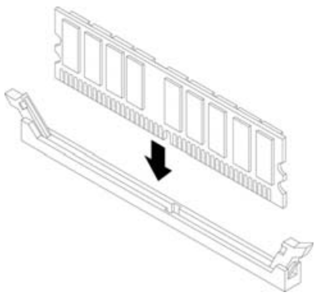


Figure 2.5: A dual in-line memory module (DIMM) with two ranks of nine banks each.

is much simpler than a SRAM cell, allowing for greater memory density (being a smaller cell) and lower manufacturing cost [25].

Despite these advantages, DRAM has a number of drawbacks when compared to SRAM. In order to allow huge numbers of cells, the capacity of the capacitor must be low. It only takes a short time for the capacity to dissipate, which is known as "charge leakage". In order to address this problem, DRAM cells must be refreshed frequently (every 64ms in most current DRAM devices). This tiny charge also creates another issue: the information read from the cell is not immediately usable, since the data line must be connected to a sense amplifier which can distinguish between a stored 0 or 1. Finally, the capacitor is discharged by read operations, so every read operation must be followed by an operation to recharge the capacitor. This requires time and energy [25].

DRAM cells are grouped together in **arrays**, as shown in Figure 2.4b. Each cell is at a cross-point of a row and a column. Both a row and a column need to be selected in order to read the content of that particular cell. Several arrays are read at the same time to provide the contents of a memory word. This group of arrays ganged together are called **banks**. Banks are aggregated in hardware modules called **Dual In-line Memory Modules (DIMMs)**. Each DIMM has one or two **ranks** of banks. Figure 2.5 shows a DIMM with two ranks of nine banks each [31].

The essential characteristics of DRAM can be thus summarized:

- DRAM cells are simple, allowing high densities at a low cost/bit (when compared to SRAM).

- DRAM is slower than SRAM, yet still fast.
- Constant refreshes due to current leakage make DRAM power-consuming.

These properties make DRAM the best match for the main memory technology requirements previously listed.

2.4 Persistent Storage: Disk, Flash

Beyond using data during computations, there is need to store data even when computers are turned off. The requirements for persistent storage can be so described:

1. It must be permanent, retaining data for several years without energy supply.
2. It must be very dense, allowing the storage of huge amounts of data in a tiny space.
3. Its cost/bit must be very low.
4. It is acceptable for persistent storage to be slower than main memory (given that it is the last memory/storage hierarchy level).

It is important to notice that, in an ideal world, there would be no distinction between main memory and persistent storage. If there was a single memory technology that could satisfy both the requirements for main memory and for persistent storage (requirements 1-3 above), there wouldn't be a separation between these two layers. Since there is still no such technology, we consider acceptable the 4th requirement mentioned above.

There are two main technologies which match these characteristics today: **disk** and **flash**.

2.4.1 Disk

The most common storage technology today is the **Hard Disk Drive (HDD, or simply "disk")**. Magnetic disks have been around since the 50s, and they have been for a long time the preponderant storage media. Most personal and enterprise computers employ disks as their persistent storage.

HDDs exploit the properties of ferromagnetic materials to retain magnetic fields. A typical HDD (depicted in Figure 2.6) has one or more **disk platters** with a magnetic coat. Data is read/written by a **head** device, that can induce or sense magnetism in the disk. A central axis, called spindle, rotates the disk. Together with the **actuator**, which moves the head between the center and the border of the disk, all disk surface can be reached by the head. HDDs also have circuitry to convert disk data to electronic form and communicate with the computer's I/O subsystem.

Disks have great areal density (bits/square inch) and a very low cost/bit, and for this reason have been the most used persistent storage for a long time. Their drawbacks are related to the fact that disks are mechanical devices with moving parts. Every time a block is accessed, the HDD needs to rotate the spindle and move the actuator in order to place the head at the appropriate place. This

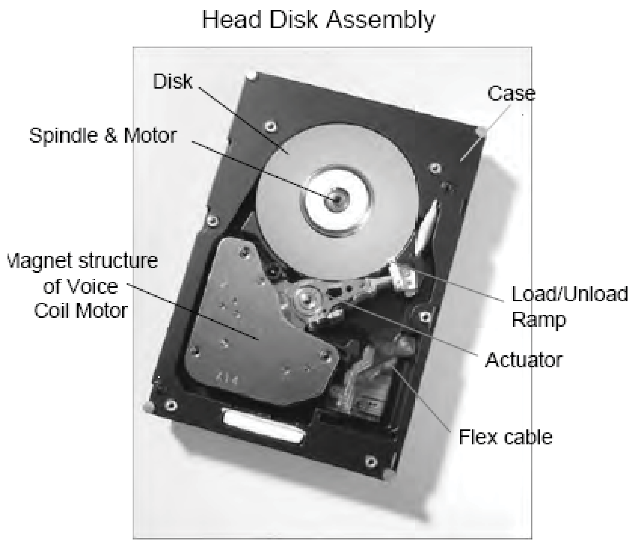


Figure 2.6: Hard disk internals overview. Reproduced from [31].

takes significantly more time than for doing the same operation in a solid-state memory technology such as SRAM or DRAM. In order to keep response times high, disks are kept continually spinning, which consumes additional energy.

These are the main steps that contribute to the time needed to complete a random disk I/O:

1. **Command overhead** - the time needed by the disk drive's microprocessor and electronics to process and handle an I/O request. It can be sped up by disk caches.

2. **Seek time** - the time to move the read/write head from its current cylinder to the target cylinder of the next command. Because it is mechanical time, it is one of the largest contributors to I/O time.

3. **Rotational latency** - the time it takes for the disk rotation to bring the start of the target sector to the head. It is also mechanical time, being another major contributor for overall I/O time.

4. **Data transfer time** - the time it takes to transfer data to/from the disk drive. It depends on data rate and transfer size. There are two distinct relevant data rates: *media data rate*, associated with how fast data can be transfered to and from the magnetic media, and *interface data rate*, associated with how fast can data be transferred between the disk drive and the host over the interface.

2.4.2 Flash

Flash memories, a type of non-volatile solid-state memory, have been built since the beginning of the 80s and dominate some niche markets today. Flash is more limited than disk regarding capacity

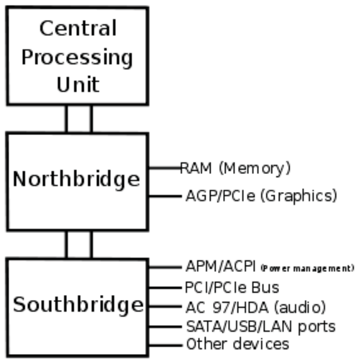


Figure 2.7: A typical north/southbridge layout. Extracted from http://en.wikipedia.org/wiki/Southbridge_(computing).

and endurance, but has much lower power consumption and thus are very suitable for mobile and embedded devices.

Flash cells are floating-gate devices that can be erased electrically. There are two major kinds of flash memories: NOR and NAND. In NOR memory, each cell in a memory array is directly connected to the wordlines and bitlines of the memory array. NAND memory devices are arranged in series within small blocks. Thus, while NAND flash can inherently be packed more densely than NOR flash, NOR flash offers significantly faster random access [13]. Due to cell density, NAND flash is the preferred type for persistent storage.

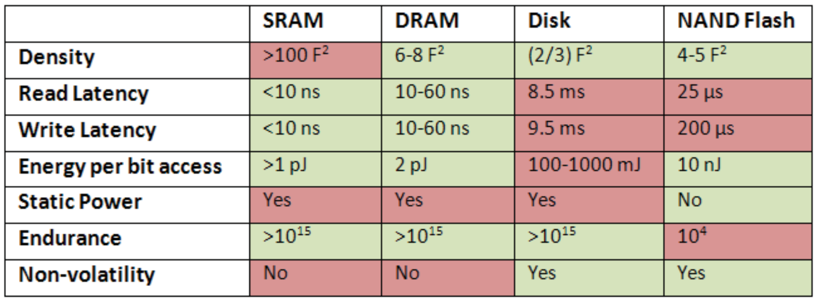
Flash is faster than disk and consumes less energy. However, it has some drawbacks that prevent it from replacing disks except for some niche applications: less density, higher cost/bit and lower endurance [13].

Both disks and NAND flash are **block devices**. Typically they are not directly addressed by the processor, like main memory, but their contents must be moved from persistent storage to memory so the processor is able to manipulate them, and must be moved from memory to persistent storage at the end of processing. This process is not transparent to the processor, and is usually managed at Operating System (OS) level. Traffic between the processor and main memory (including off-chip caches) is made through a high-speed memory controller bus, also known as **northbridge**. Access to block devices is made through the slower I/O controller hub known as **southbridge**. Figure 2.7 shows the typical relationship between the processor, the northbridge and the southbridge.

2.5 Limitations of Current Memory Technologies

The memory subsystem has become one of the most important topics in computer design. As Bruce Jacob [31] put it,

> "The increasing gap between processor and memory speeds (...) is so severe we are now one of those down-cycles where the processor is so good at number-crunching it has completely sidelined itself; it is too fast for its own good, in a sense. (...) memory



	SRAM	DRAM	Disk	NAND Flash
Density	>100 F^2	6-8 F^2	(2/3) F^2	4-5 F^2
Read Latency	<10 ns	10-60 ns	8.5 ms	25 μs
Write Latency	<10 ns	10-60 ns	9.5 ms	200 μs
Energy per bit access	>1 pJ	2 pJ	100-1000 mJ	10 nJ
Static Power	Yes	Yes	Yes	No
Endurance	$>10^{15}$	$>10^{15}$	$>10^{15}$	10^4
Non-volatility	No	No	Yes	Yes

Table 2.1: Comparison of current memory/storage technologies: SRAM, DRAM, disk and NAND flash. Cell size uses standard feature size (F^2)

> subsystems design is now and has been for several years the *only* important design issue for microprocessors and systems. Memory-hierarchy parameters affect system performance *significantly* more than processor parameters (e.g., they are responsible for 2-10x changes in execution time, as opposed to 2-10%)"

The high level of sophistication attained by modern memory systems is largely derived from the process predicted by Gordon Moore in 1965 [47], known as Moore's Law. It states that the number of devices that can be integrated on a chip of fixed area would double every 12 months (later amended to doubling every 18–24 months). Moore's law was the driving force behind dramatic reductions in unit cost for memory, enabling products of higher density and putting huge amounts of memory in the hands of the consumer at much reduced cost. This behavior has made the prediction of near-future product developments extremely reliable because the underlying device physics, materials, and fabrication processes have all been scalable, at least until now [13]. Figure 2.1 summarizes the main characteristics of current memory/storage technologies.

An important issue for the near future concerns DRAM approaching physical limits that might limit its growth in the next decade, creating a "power wall". DRAM must not only place charge in a storage capacitor but must also mitigate sub-threshold charge leakage through the access device. Capacitors must be suficiently large to store charge for reliable sensing and transistors must be suficiently large to exert effective control over the channel. Given these challenges, manufacturable solutions for scaling DRAM far beyond current limits is a challenge [36]. This fuels a concern with the growing power demands of DRAM:

> "For several decades, DRAM has been the building block of the main memories of computer systems. However, with the increasing size of the memory system, a significant portion of the total system power and the total system cost is spent in the memory system. For example, (...) as much as 40% of the total system energy is consumed by the main memory subsystem in a mid-range IBM eServer machine." [53]

Disk density (and consequently capacity) is unlikely to face a similar scaling problem in the near future, but the latency gap between DRAM and disk is an issue that would be worth to solve. As Burr et al. put it,

> "The best-case access time of a magnetic HDD has remained fairly constant over the past decade at approximately 3-5 milliseconds (...). A 3-GHz microprocessor will execute nine million instructions while waiting for this data. In fact, an enormous part of a modern computing system is designed expressly to hide or finesse this unpleasant discrepancy in performance." [13]

Flash also has its limitations as persistent storage:

> "(...) the cost per gigabyte (GB) for flash memories is nearly 10x that of magnetic storage. Moreover, flash memories face significant scaling challenges due to their dependence upon reductions in lithographic resolution as well as fundamental physical limitations beyond the 22 nm process node, such as severe floating gate interference, lower coupling ratio, short channel effects, and low electron charge in the floating gate. Thus, to replace HDDs, alternative NVM technologies that can overcome the shortcomings of NAND flash memories and compete on a cost per TB basis with HDDs must be found." [34]

...and as main memory:

> "NAND flash has very limited number of write/erase cycles: 10^5 rewrites as opposed to 10^{16} for DRAM. NAND flash also requires a block to be erased before writing into that block, which introduces considerably extra delay and energy. Moreover, NAND flash is not byte-addressable." [79]

In a 2007 article, Barroso and Hölzle [6] proposed the notion of **energy-proportional** computing and maintained that it should be a primary design goal for computer design. They argue that datacenters are designed to keep server workload between 10-50% of their maximum utilization levels. This is done to ensure that throughput and latency service-level agreements will be met and that there will be room for handling component failures and planned maintenance. The problem is that the region under 50% utilization is the lowest energy-efficient region of servers, due to static power consumption (depicted in Figure 2.8a). As seen, memory/storage components consume both dynamic power (used to change memory contents) and static power (used for data retention and component availability). Static power used by SRAM and DRAM is used mostly for refreshes due to power leakage; in disks, most static power is used to keep the disk spinning and ready to answer requests with low latencies. As Barroso and Hölzle put it, "essentially, even an energy-efficient server still consumes about half of its full power when doing virtually no work" [6]. This problem is aggravated by the fact that the costs are not reflected only in the server electricity bill, but also in energy for cooling the datacenter and infrastructure provisioning costs, that increase proportionally to the server energy budget.

The authors of the article also mention that CPUs are the components that exhibit most energy-proportional behavior today, and that the components that most contribute to less server energy efficiency are, in this order, DRAM (50%), disk drives (25%) and networking switches (15%). A

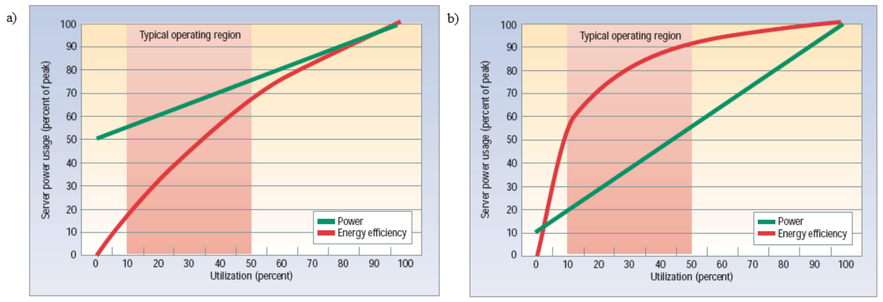


Figure 2.8: Relationship between power and energy-efficiency for a) current server behavior; b) theoretical energy-proportional server behavior. Reproduced from Barroso and Hölzle 2007 [6].

machine with a dynamic power range of 90% could cut by one-half the energy used in datacenter operations, due to its better energy efficiency in low-utilization regions, shown in figure 2.8b [6].

Due to all these limitations, there is intense research to create new alternatives of memory technology that can address these problems and prevent a "power wall" from being reached. In chapter 4 some of the most promising emerging technologies are presented. The next chapter describes how current technologies are applied in today's memory system organization.

3. MEMORY SYSTEM ORGANIZATION

The previous chapter presented the main technologies used today for memory and storage. This chapter describes how these technologies are employed in today's memory system organization. This is an important topic to understand, since the emerging technologies presented in chapter 4 will also potentially impact memory system design and organization.

3.1 Overview

A typical PC organization is displayed in Figure 3.1, illustrating a two-way multiprocessor. This system has the most common characteristics usually found in modern computers, and will be used as a reference during the rest of this section.

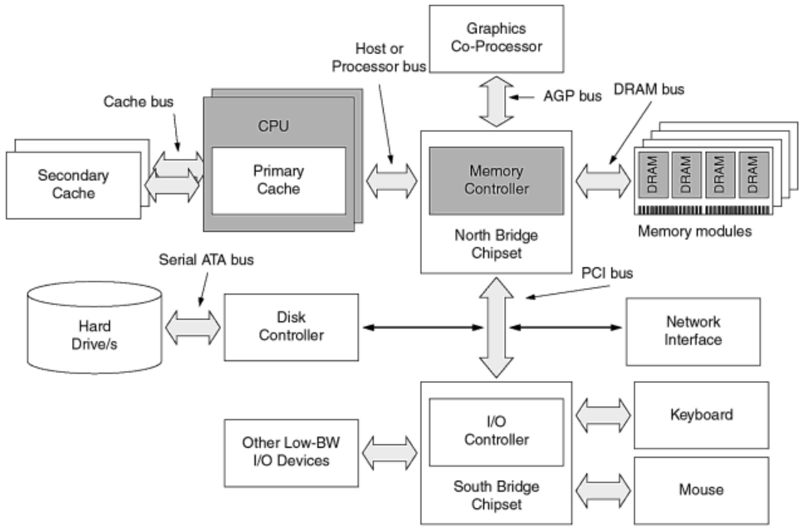


Figure 3.1: A typical PC organization, showing a two-way multiprocessor, with each processor having its own secondary cache. Extracted from [31].

Each of the two **CPUs** (Central Processing Units) has its own on-chip primary cache, also known as **L1 cache**. In addition, each CPU also has access to an off-chip secondary cache (**L2 cache**) using a dedicated cache bus. Access to **main memory** is made through the **memory controller** residing inside the **north-bridge** chipset. The north-bridge mediates the communication between the processors and the DRAM **memory modules** through a DRAM bus. It also mediates communication with the **graphics co-processor** through an AGP bus and I/O devices (such as the **disk controller** and **network interfaces**) through a PCI bus. The north-bridge also is connected to the **south-bridge** chipset, which has an I/O controller responsible for communicating with **low bandwidth I/O devices** such as keyboard, mouse, etc. The disk controller communicates with **disks** through a Serial ATA (SATA) bus.

In the next section we will examine in more depth the main components of the memory system: **caches**, **main memory** and **disks**.

3.2 Cache

Processor **caches** enhance overall system performance providing very fast access to small subsets of memory data, expected to be accessed much more frequently than others. This is effective due to a phenomenon known as **locality of reference**, so named because memory references tend to be localized in time and space. This can be so summarized:

- If you use something once, you are likely to use it again (temporal locality);
- If you use something once, you are likely to use its neighbor (spatial locality).

Usually a processor cache is divided in three distinct areas: a **data** cache, an **instruction** cache and a **Translation Lookaside Buffer (TLB)**, used to speed up virtual-to-physical address translation for both executable instructions and data. They usually are **transparent caches**, meaning that the same addressing space is shared between the cache and the backing storage.

The data cache stores chunks of data that come from main memory, called **cache blocks** or **cache lines**. Each cache line contains:

- a **cache tag** indicating the index of the cached data in main memory
- the **status** of the cache line, indicating if its data are **clean** (consistent with the value stored in main memory) or **dirty** (recently written in cache but not in main memory)
- the cache data.

When the processor asks for the contents of a specific memory address, these contents are first searched in the cache. If they are found (a **cache hit**), they are immediately returned. If they are not found (a **cache miss**), the memory system goes on to the next hierarchy level (main memory) and fetches the result. The proportion of accesses that result in a cache hit is known as the **hit rate**. The time needed to discover whether a datum is cached or not is called the **hit time**.

Cache blocks can be grouped in **sets** of memory locations, and the number of blocks per set is called **cache associativity**. Associativity is a trade-off. If there are x places to which the replacement policy could have mapped a memory location, then to check if that location is in the cache, x entries must be searched. Checking more places takes more power, chip area, and potentially time. On the other hand, caches with more associativity suffer fewer misses, so that the CPU wastes less time reading from the slow main memory. The following associativity schemes exist:

- **Direct-mapped** - each set contains only one cache block. It has the best hit times but the worst hit rates.

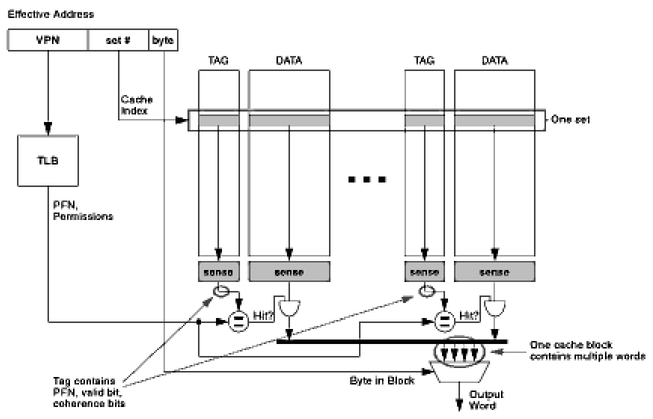


Figure 3.2: Organization of a Static RAM (SRAM) processor cache using set-associative mapping. Extracted from [31].

- **Fully-associative mapping** - there is a single set for all lines, i.e., any line can store the contents of any memory location. This option has the best hit rates but the worst hit times.

- **Set-associative mapping** - this is a compromise between the direct-mapped and fully-associative approaches. Each set contains n check blocks, usually 2, 4 or 8.

Figure 3.2 shows the organization of a Static RAM (SRAM) processor cache using set-associative mapping.

In order to be efficient, caches must store memory locations that are frequently used. The responsibility to decide what will be stored in the cache and what will not is called **cache content-management**. Several distinct content-management heuristics exist; one of the most known is Least Recent Usage (LRU), that keeps in the cache data most frequently used in the past.

Caches also need **consistency-management**, to ensure that the overall memory space, as viewed by different processors and levels of the hierarchy, is consistent. This comprises three main charges:

1. **Keep the cache consistent with itself** - ensure that a specific memory address has the same value in the cache. This can be done by keeping just one copy of a given address or use some coherency mechanism to update all copies of the same address whenever a change happens.

2. **Keep the cache consistent with the backing store** - make sure that all values changed in the cache get persisted in the backing store. Two policies can be employed: **write-through**, where all changes in the cache are immediately applied to the backing store, and **write-back**, where changes are applied initially only to the cache, and are written to the backing store at some future point in time, usually when there is a block replacement.

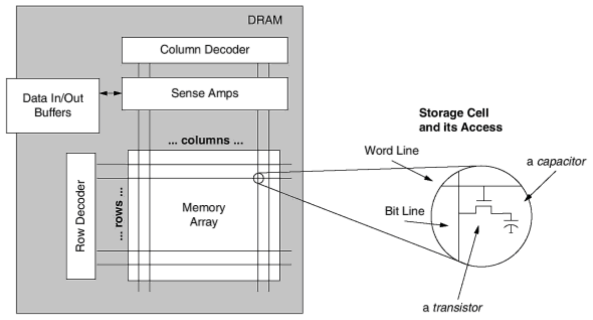


Figure 3.3: Basic organization of DRAM internals. Extracted from [31].

3. **Keep the cache consistent with other caches** - this is similar to the first charge, but applies to different caches and cache levels. It is also handled in a similar way.

3.3 Main Memory

The description below, except when indicated, was obtained from [31].

A **memory module** is typically connected indirectly to a CPU through a **memory controller**. In PC systems, the memory controller is part of the **north-bridge** chipset.

Memory cells are disposed on a grid called a **memory array**. Each cell stores a bit in the intersection of a row and a column. Figure 3.3 shows a DRAM array. A set of memory arrays that operate together are grouped in memory **banks**. The number of arrays in a bank determines the **data width** of the memory device (e.g., 4 arrays indicate a *x4* data width). Data widths range from x2, x4 and x8 up to wider outputs of x16 and x32.

A **memory chip**, as represented in Figure 3.4, may contain several banks. Each memory chip contains a **row decoder** and a **column decoder** which are used to select a particular bit of each memory array. For each array there is a differential **sense amplifier** that senses the electrical properties of the cell (e.g., voltage, charge) and converts it to a digital value (0 or 1). The output of the sense amplifier is stored in a data buffer before it can be sent over the bus to the memory controller.

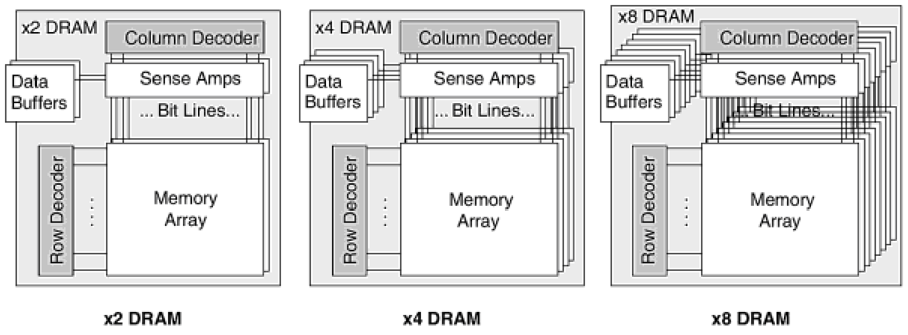


Figure 3.4: Logical organization of wide-out DRAMs. Extracted from [31].

The most common form factor for memory modules is called **DIMM** (dual-inline memory module), where multiple memory chips are placed in a single PCB (printed circuit board) with one or two **ranks** of memory chips, each one in one side of the PCB. At the bottom of the DIMM there are sets of **pins** used to communicate with the **memory bus**. A system can usually support several DIMMs, and all of them are connected to the memory controller by a common memory bus. Figure 3.5 displays the relationship between memory arrays, banks, ranks, DIMMs and the memory controller in a DRAM-based system.

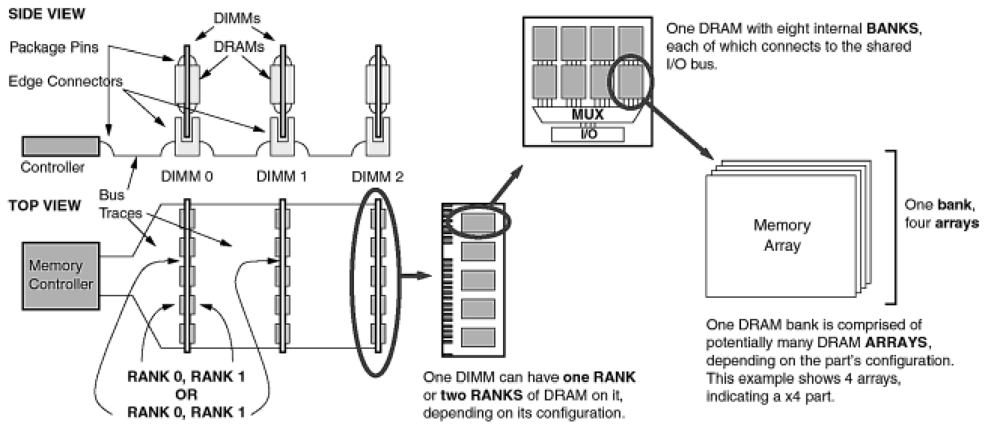

SIDE VIEW
Package Pins
Edge Connectors
DIMMs
DRAMs
Controller
DIMM 0
DIMM 1
DIMM 2
TOP VIEW
Bus Traces
Memory Controller
RANK 0, RANK 1 OR RANK 0, RANK 1
One DIMM can have one RANK or two RANKS of DRAM on it, depending on its configuration.
MUX
I/O
One DRAM with eight internal BANKS, each of which connects to the shared I/O bus.
Memory Array
One bank, four arrays
One DRAM bank is comprised of potentially many DRAM ARRAYS, depending on the part's configuration. This example shows 4 arrays, indicating a x4 part.

Figure 3.5: DIMMs, ranks, banks and arrays in a DRAM system. Extracted from [31].

DRAM memory modules today are a commodity; any DRAM DIMM should be equivalent to any other with similar specifications (width, capacity, speed, interface, etc.). The organization that defines standards for DRAM memory modules is the **JEDEC** (Joint Electron Device Engineering Council). The most common memory modules today follow the JEDEC standard for **DDR** (Double Data Rate) **SDRAM** (Synchronous DRAM). This standard mandates the memory bus to be split in 4 separate busses, each one with their own traces:

- **Data** - moves data **words** between the memory modules and the memory controller;
- **Address** - specifies the **address** to be read/written from/to the memory module;
- **Control** - sends **commands** from the memory controller to the memory module;
- **Chip-Select** - selects which memory module a command is being directed to.

Figure 3.6 shows a JEDEC-style memory bus organization. Each set of traces connecting a memory controller to memory modules is called a **channel**. A memory controller can connect to modules through one or more channels. Organizations with more channels provide more parallelism at the expense of higher cost and complexity. Organizations with less channels are simpler and less expensive, but can be a bottleneck for communication between the memory controller and the memory modules. Synchronous DRAM (SDRAM) is so called because it uses the system clock to control timing in the memory busses, exchanging data between a memory controller and a memory module in every system clock **tick**. Double Data Rate (DDR) SDRAMs are so named use a dual-edged clock, transmitting data in both the rising and the falling edge of a clock tick. So for every clock tick there is a memory bus **beat**.

The memory controller is responsible for receiving memory read/write requests from CPUs. These memory transactions are persisted in one or more **queues** inside the memory controller before being sent to the memory modules, and can be reordered by the memory controller to increase overall efficiency. One common reordering strategy is **write caching**: read requests are typically critical in terms of latency, but write requests are not; for this reason, read requests are usually prioritized over write requests, as long as the functional correctness of programs is not violated.

Another main responsibility of the memory controller is translating the addressing scheme used by the CPU (**physical address**) to the actual row/column address in a DRAM bank (**DRAM address**). There are usually three distinct levels of memory addressing in a typical PC:

- **Virtual Addresses** - used by software applications and managed by the operating system, and potentially larger than the physical memory available;

- **Physical Addresses** - a continuous representation of the physical memory available - this is how the CPU addresses memory. The mapping from virtual to physical addresses is made by the Operating System with help from the Translation Lookaside Buffer (TLB) contained in the processor L1 cache;

- **DRAM Addresses** - the actual location of memory data in the memory modules, including channel, rank, bank, row and column. The expansible nature of JEDEC-style memory makes it possible for memory modules to be added/subtracted from the system. Since the memory controller manages the current physical organization of memory modules, it is responsible for the translation between physical addresses and DRAM addresses.

The memory controller is also responsible for **refresh management** of DRAM memories. DRAM cells must be frequently refreshed due to capacitor power leakage. Memory controllers usually send periodical refresh commands to the memory modules to avoid losing memory contents.

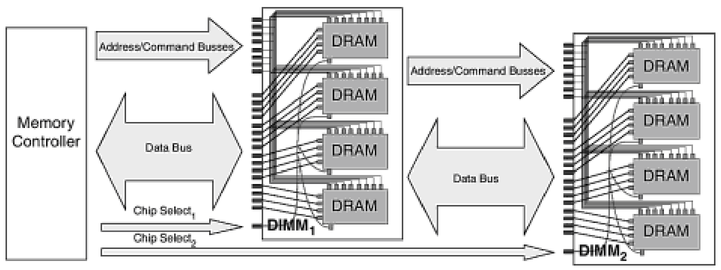


Figure 3.6: JEDEC-style memory bus organization, consisting of a memory controller and two memory modules with a 16-bit data bus and an 8-bit address and command bus. Extracted from [31].

In order to do a memory read transaction, the following steps are typically followed:

1. A CPU requests a memory read transaction to a memory controller.

2. The memory controller converts the transaction in to a sequence of memory commands (translating the physical address to a DRAM address) and queues them.

3. When the command is unqueued, the memory controller sends to the appropriate memory module a **Precharge** (**PRE**) command. This sets the bitlines to a voltage level halfway between 0 and 1, which is needed for the sense amplifier to detect the value in each capacitor of a row.

4. The memory controller sends to the memory module a **Row Address Strobe** (**RAS**) command, which causes the contents of an entire row to be activated and placed at the sense amplifier.

5. The memory controller sends to the memory module a **Column Address Strobe** (**CAS**) command, which specifies which specific column from the row currently at the sense amplifier should be moved to the data bus where it can be transferred to the memory controller.

6. Most of the current DRAM implementations don't transfer just a single column, but a predefined number of sequential columns. It saves bandwidth avoiding the need for the memory controller to send several of CAS commands to fetch the value of a series of contiguous columns. This is called a **burst,** and is a form of prefetching based on locality principles. Memory modules can have a **Programmable Burst Length** set by the memory controller.

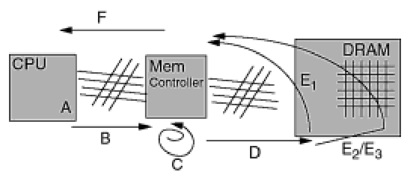
7. The memory controller sends back to the CPU the memory contents of the requested physical address.

It is important to notice that some memory controllers reset the contents of the sense amplifier after a CAS command, making it faster to read contents from a different row. This is called an **open-page row-buffer-management policy**, and it is used more frequently in systems with more processors and less temporal and spatial locality. Other memory controllers keep the sense amplifier open after a CAS command, saving time and bandwidth if a new CAS command is issued against the same row. This is called a **closed-page row-buffer-management policy**, and it is used in systems with less processors and more temporal and spatial locality.

In open-page memory systems, when reading new column in a row already open, steps 3 and 4 (described above) are not needed. In both open-page and closed-page systems step 3 is not needed if the memory bank is already precharged. A summary of these steps can be seen at Figure 3.7.

3.4 I/O

The I/O subsystem is responsible for communicating with persistent storage devices, including disk drives and/or flash memory. As previously mentioned in section 3.1, this interaction occurs via



A: Transaction request may be delayed in Queue
B: Transaction request sent to Memory Controller
C: Transaction converted to Command Sequences (may be queued)
D: Command/s Sent to DRAM
E_1: Requires only a **CAS** or
E_2: Requires **RAS + CAS** or
E_3: Requires **PRE + RAS + CAS**
F: Transaction sent back to CPU
DRAM Latency = A + B + C + D + E + F

Figure 3.7: Steps of a DRAM read. Extracted from [31].

the south-bridge chipset. Most of the communication essentially involves reading from or writing to these devices through **I/O commands**. The most typical I/O commands have this basic structure:

1. Whether the command is a read or write operation;

2. The starting address for the command;

3. the number of sectors of data to be read or written.

The communication channel over which I/O requests are from the processor to the storage devices and through which data transfers for reading/writing happen is called the **storage interface**. A basic model for attaching an storage device to a computer system (also called "host") is depicted in Figure 3.8. The **I/O bus** has a higher latency compared to the memory bus. The **host side controller** may have different names, depending on the system and interface (Host Bus Adapter, Storage Controller, Disk Drive Adapter, etc.), but is always responsible for interfacing the host communication with the storage devices. Each storage device (drive) has its own **drive controller**, that directly controls the storage media (either disk or flash).

The communication between the host side controller and the drive controller is made through a specific storage interface. Such interfaces usually are comprised of two layers: a **lower layer**, that defines cables, connectors, electrical or optical signals and other transport procedures; and a **higher layer**, that defines the protocol and set of commands that can be exchanged between the host and the drive. This separation of layers allows different interfaces to share the same higher or lower level layer. For instance, the ATA and SATA interfaces have the same higher layer, but the lower layer of the first is parallel (in the sense that multiple signals are sent in parallel over multiple wires), and the second is serial. The most usual interfaces in use today are the following:

- **ATA** - one of the first standardized interfaces, is still very common for being a low-cost option. It is a parallel interface that allows up to 4 drives in a host controller.

- **SATA** - as the name implies, Serial ATA is the serial version of the ATA interface. It has better performance than its parallel version.

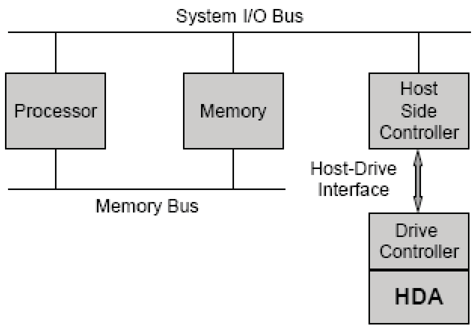


Figure 3.8: Model of a storage device (disk drive) attached to a host system via an interface. Extracted from [31].

- **SCSI** - the Small Computer Systems Interface is a more advanced interface with functionalities and features not available in ATA, and consequently higher cost. It is a parallel interface.
- **SAS** - stands for Serial Attached SCSI, and essentially is a serial interface based on the SCSI protocol, for enhanced performance.
- **FC** - Fibre Channel is a high-end, feature-rich, serial interface. It is the most complex and expensive of all, but presents far more performance and flexibility.

Caches play an important role in increasing storage device performance. Typically both the host side controller and the drive controller have built-in caches. Main memory is also used to buffer data read from storage devices, and can sometimes act as a cache, as the data being read is already available in DRAM. Since storage devices are more efficient manipulating sequential data, the drive cache is very important to amortize access time through prefetching mechanisms.

The next chapter will present the emerging non-volatile memory technologies being researched, and chapter 5 discusses the impacts of these emerging memory technologies concerning how they can contribute to create computers with **persistent main memory**.

4. EMERGING MEMORY TECHNOLOGIES

Chapter 2 presented the current technologies used in memory hierarchy, discussed their limitations and the arguments for the research of more efficient memory technologies. This chapter presents the emerging memory technologies that can be an answer to these issues.

There are several new Non-Volatile Memory (NVM) technologies under research. One study [34] lists 13 of such technologies: FRAM, MRAM, STTRAM, PCRAM, NRAM, RRAM, CBRAM, SEM, Polymer, Molecular, Racetrack, Holographic and Probe. Most these technologies are in different stages of maturity. Some of them are still in early research stages, others have working prototypes, and some of them are already entering into commercial manufacturing.

In the present work, we will limit our study to three of these technologies: Phase-Change RAM, Resistive RAM (including Memristors) and Magnetoresistive RAM (including Spin-Torque Transfer RAM). All these fall into the category of the most actively researched today, are backed by solid companies of the technology industry, and considered most promising of being commercially feasible.

For those who are interested in a more broad analysis of all different kinds of NVM under research, the works of Burr et al. [13], and Kryder and Kim [34] can be consulted.

4.1 Phase-Change RAM (PCRAM)

Phase-Change Random Access Memory (also called PCRAM, PRAM or PCM) is currently the most mature of the new memory technologies under research. It relies on some materials, called **phase-change materials**, that exist in two different phases with distinct properties:

1. An **amorphous phase**, characterized by high electrical resistivity;

2. A **crystalline phase**, characterized by low electrical resistivity [56].

These two phases can be repeatedly and rapidly cycled by applying heat to the material. To make it amorphous, it is melt-quenched using a high-power electrical pulse that is abruptly cut off. To crystallize the material, it is heated above its crystallization temperature using a moderate power pulse with a longer duration. Since the duration of this pulse varies according to the crystallization speed of the material being used, this operation tends to dictate the writing speed of PCRAM. Reflectivity can vary up to 30%, but resistivity changes can be as large as five orders of magnitude. [13, 56]

The principle of phase-change memory is known since the 1960s, but only recent discoveries of phase-change materials with faster crystallization speeds led to the possibility of commercially feasible memory technology. The most important materials are chalcogenides such as $Ge_2Sb_2Te_5$ (GST), that can crystallize in less than 100 ns [56].

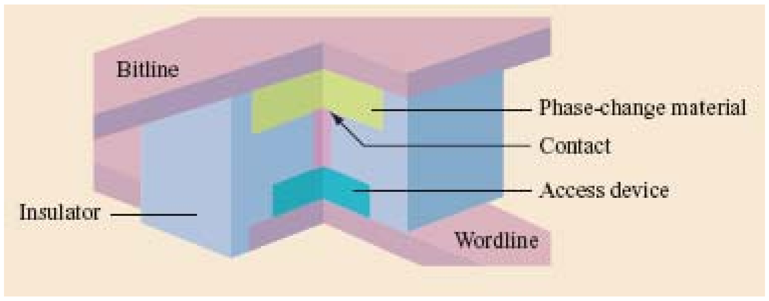


Figure 4.1: Example of a phase-change memory cell. The current is forced to pass through the phase-change material. Image obtained from [56].

Figure 4.1 shows a memory cell based on phase-change principles. The SET operation is achieved by crystallizing the material and RESET by making it amorphous. Unlike Flash memory, PCRAM can be switched from 0 to 1 and vice-versa without an ERASE operation.

Given the great difference in resistance, phase-change materials can be easily used to store binary states per cell (single-level cell) and even more states (multi-level cell) [13, 36].

PCRAM is argued to be a scalable technology. As the feature density increases, phase-change material layers become thinner and need less current for the programming operations. It has been demonstrated to work in 20nm device prototype and is projected to scale down to 9nm. DRAM presents challenges to scale down beyond 40nm [36, 56].

As previously mentioned, the SET latency is the longest and determines the write performance. Latencies of 150 ns for SET and 40 ns for RESET operations have been demonstrated. Write energy is determined by the RESET operation, which dissipates 480 μW, while SET dissipates 90 μW. The read latency is 48 ns and dissipates 40 μW. Both read and write latencies are several times slower than DRAM, although only by tens of nanoseconds [36].

Endurance is bound to the number of writes. This happens because when current is injected into a phase-change material, thermal expansion and contraction degrade the contacts, so that currents are no more reliably injected into the cell. The current write endurance varies between 10^4 and 10^9 writes, but we can conservatively assume 10^8 as a reasonable reference value [36, 79].

Today PCRAM is positioned as a Flash replacement. It offers great advantages over Flash, but given the current limitations of access latency, energy consumption and endurance, further development is required in order to employ it as a replacement for DRAM [36, 37].

4.2 Resistive RAM (RRAM)

Despite the fact that PCRAM also uses resistance variances to store bit values, the term **Resistive RAM** (RRAM) has been applied to a distinct set of technologies that explore the same phenomenon. Essentially these technologies fall into one of two categories [13]:

1. Insulator resistive memories: based on bipolar resistance switching properties of some metal oxides. The most important example is the memristor memory device, which will be further

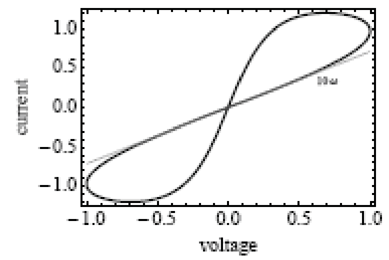


Figure 4.2: Relationship between current and voltage in a memristor, known as a “hysteresis loop”. Extracted from Wang, 2008 [71].

described in more detail.

2. Solid-electrolyte memories: based on solid-electrolyte containing mobile metal ions sandwiched between a cathode and an anode. Also known as Programmable Metallization Cell (PMC) or Conductive Bridge RAM (CBRAM).

There is a long list of RRAM technologies [11, 13, 34, 77]. In this paper we will concentrate our attention on the **memristor**, which is currently the most promising RRAM technology under research [77].

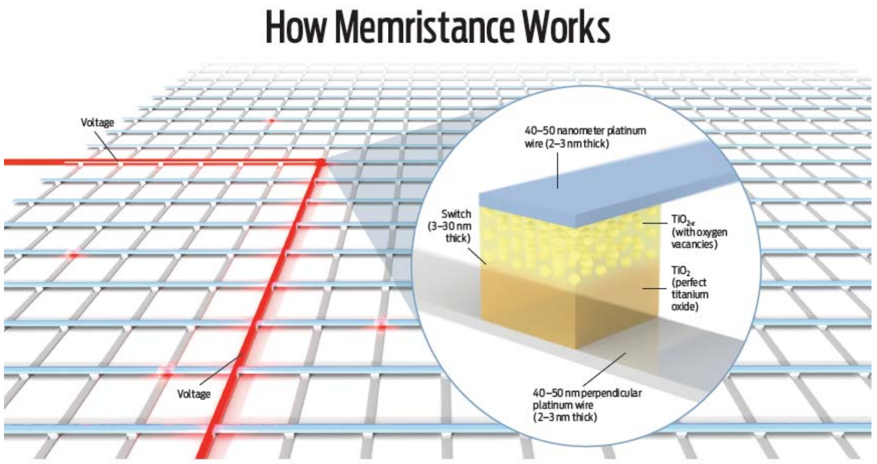
4.2.1 Memristor

Since the 19th century, we have known three fundamental passive circuit elements: the resistor, the inductor and the capacitor. In 1971, Leon Chua theorized the existence of a fourth passive circuit element, which he called the **memristor** [15], but no actual physical device with memristive properties could be constructed. In 2008, a group of scientists reported the invention of a device that behaved as predicted for a memristor [62]. Later the same year, an article detailed how that device could be used to create nonvolatile memories [77].

The property of memristors particularly relevant to memory devices is the nonlinear relationship between current (I) and voltage (V), depicted on Figure 4.2. In the words of R. Stanley Williams, the leader of the research team that invented the first memristor device:

> “Memristor is a contraction of ’memory resistor’, because that is exactly its function: to remember its history. A memristor is a two-terminal device whose resistance depends on the magnitude and polarity of the voltage applied to it and the length of time that voltage has been applied. When you turn off the voltage, the memristor remembers its most recent resistance until the next time you turn it on, whether that happens a day later or a year later. (...) The ability to indefinitely store resistance values means that a memristor can be used as a nonvolatile memory.” [72]

This memristor device consisted of a crossbar of platinum wires with titanium dioxide (TiO_2) switches, as shown in Figure 4.3. Each switch consists of a lower layer of perfect titanium dioxide


How Memristance Works
Voltage
Voltage
40–50 nanometer platinum wire (2–3 nm thick)
Switch (3–30 nm thick)
TiO_{2-x} (with oxygen vacancies)
TiO_2 (perfect titanium oxide)
40–50 nm perpendicular platinum wire (2–3 nm thick)

THE CROSSBAR ARCHITECTURE: The crossbar architecture is a fully connected mesh of perpendicular wires. Any two crossing wires are connected by a switch. To close the switch, a positive voltage is applied across the two wires to be connected. To open the switch, the voltage is reversed.

THE SWITCH: A switch is a 40-nanometer cube of titanium dioxide (TiO_2) in two layers: The lower TiO_2 layer has a perfect 2:1 oxygen-to-titanium ratio, making it an insulator. By contrast, the upper TiO_2 layer is missing 0.5 percent of its oxygen (TiO_{2-x}), so *x* is about 0.05. The vacancies make the TiO_{2-x} material metallic and conductive.

Figure 4.3: How memristance works [72].

(TiO_2), which is electrically insulating, and an upper layer of oxygen-deficient titanium dioxide (TiO_{2-x}), which is conductive. The size of each layer can be changed by applying voltage to the top electrode. If a positive voltage is applied, the TiO_{2-x}layer thickness increases and the switch becomes conductive (ON state). A negative voltage has the opposite effect (OFF state). This behavior matches the hysteresis loop previously shown in Figure 4.2. [62, 72, 77].

Titanium dioxide was used in the mentioned prototype, but several other oxides are known to present similar bipolar resistive switching, and there are multiple research projects in motion to explore these other materials for similar memory device implementations [11, 13].

Memristive memory technology is less mature than PCRAM, and DIMM prototypes aren't available yet, but there are already a few papers that predict some fundamental characteristics of such memories [29, 38].

Scalability is one aspect where memristor is most efficient. A cell density of 10 nm has been achieved and a density between 4-5 nm is predicted for the next few years [38, 72]. Beyond that, memristor memory devices can benefit from multi-level cell (MLC) storage and 3D die stacking, which can greatly improve the overall density [11, 29, 38, 72].

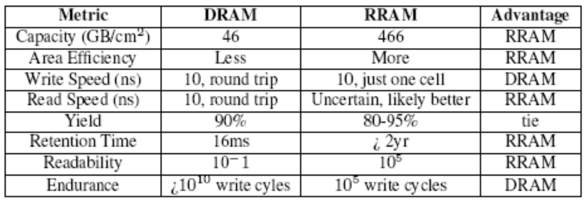
We haven't found any report of actual read speeds, but they are expected to be faster than DRAM speeds, which are around 10 ns. The write speed is 10 ns for a single cell; not as fast as DRAM, which can do round-trip writes (including bus latency, addressing, transaction and data return) in the same time [38].

The switching power consumption of memristor can be 20 times smaller than flash [29].

Another property of memristors is that they are made from common semiconductors that are CMOS-compatible, and can easily be produced in existing semiconductor factories [13, 72].

The Achilles heel of memristor is endurance. The existing memristor prototypes have shown so far an endurance of 10^5write cycles, and main memory devices need an endurance in the order of

Table 4.1: Comparison between DRAM and RRAM (Memristor) metrics. Extracted from Lewis and Lee [38].



Metric	DRAM	RRAM	Advantage
Capacity (GB/cm^2)	46	466	RRAM
Area Efficiency	Less	More	RRAM
Write Speed (ns)	10, round trip	10, just one cell	DRAM
Read Speed (ns)	10, round trip	Uncertain, likely better	RRAM
Yield	90%	80-95%	tie
Retention Time	16ms	¿ 2yr	RRAM
Readability	10^-1	10^5	RRAM
Endurance	¿10^{10} write cyles	10^5 write cycles	DRAM

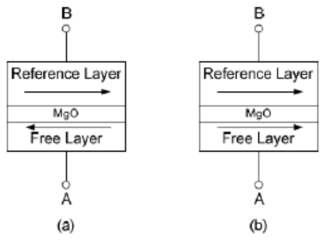


Figure 4.4: A conceptual view of MTJ structure. (a) Anti-parallel (high resistance), which indicates "1" state; (b) Parallel (low resistance), which indicates "0" state. Reproduced from Dong, 2008 [24].

10^{17}. R. Stanley Williams believes this can be improved in the next years, but memristor devices, as they are today, cannot be directly used for main memory. It could be used as a flash replacement, for write-seldom, read-often devices such as FPGAs, and as part of hybrid main memory devices using a combination of RRAM persistent storage and DRAM buffers [38].

Lewis and Lee published an interesting table comparing RRAM memristor against DRAM metrics, that we show in Figure 4.1.

4.3 Magnetoresistive RAM (MRAM)

Magnetoresistive RAM (MRAM), sometimes called Magnetic RAM, is a memory technology that explores a component called **Magnetic Tunnel Junction** (MTJ). An MTJ, depicted on Figure 4.4, consists of two ferromagnetic layers separated by an oxide tunnel barrier layer (e.g.,: M_gO). One of the ferromagnetic layers, called the **reference** layer, keeps its magnetic direction fixed, while the other, called the **free** layer, can have its direction changed by means of either a magnetic field or a polarized current. When both the reference layer and the free layer have the same direction, the resistance of the MTJ is low. If they have different directions, the resistance is high. This phenomenon is known as **Tunneling Magneto-Resistance** (TMR) [13, 24, 34].

MRAM is a relatively mature technology, but current implementations suffer from density and energy constraints that seriously affect its capacity to compete with existing memory technologies such as DRAM or Flash. An specific type of MRAM under development called **STT-RAM** has a good chance to overcome these problems and position MRAM as a commercially feasible NVM alternative [34]. In the next section we'll explore STT-RAM in more detail.

4.3.1 STT-RAM

Conventional MRAM (also called "toggle-mode" MRAM) uses a current induced magnetic field to switch the MTJ magnetization. The amplitude of the magnetic field must increase as the size of MTJ scales, which compromises MRAM scalability. Spin-Torque Transfer MRAM (STT-RAM) technology tries to achieve better scalability by employing a different write mechanism based on spin polarization [39].

Read/write latencies for STT-RAM are very fast. One study estimates a read latency of 5.5 ns, faster than DRAM. The same study predicts slower write speeds, around 12.5 ns, but they are still very fast [24].

Endurance is excellent, reaching figures above 10^{15}cycles [39].

Regarding energy efficiency, STT-RAM cells consume around 0.026 nJ for reading and 2.833 nJ for writing. This is a relatively high consumption, but it's very favorable when compared to SRAM and DRAM, since MRAM, as all other non-volatile memories, has very low energy leakage [24].

The less favorable aspect of STT-RAM is density. Its cells have less density than currently achieved by DRAM. A comparison shows that while DRAM scale down to 6-8F^2, STT-RAM currently doesn't scale below $37F^2$, thus being 1.7 times less dense than DRAM. STT-RAM supports MLC and 3D integration [24].

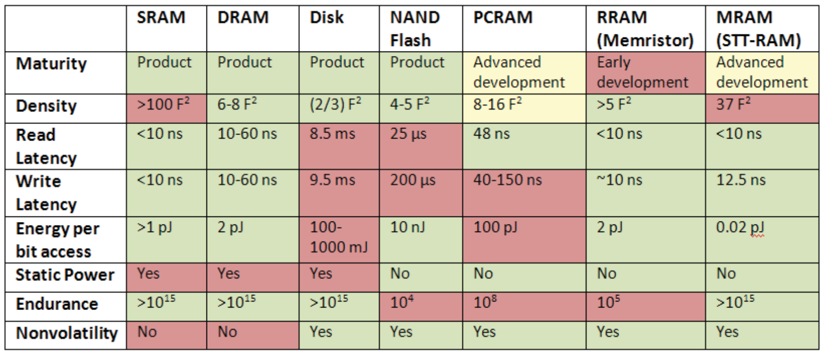
In 2005, Johan Åkerman published an article [54] proposing that MRAM could be the "universal memory" technology that would render obsolete all other components of memory hierarchy (SRAM, DRAM, disks, flash). Since then, papers on MRAM (including STT-RAM) often mention its possible future as "universal memory". In practice, due to MRAM current density limitations, it is not seen as a real replacement for DRAM as main memory (not to mention disks). Current ideas focus on using STT-RAM as an SRAM replacement for building cache memories, due to its excellent read/write speeds, endurance, and much superior energy efficiency [24, 39].

4.4 Comparison Between Memory Technologies

So far have seen the most promising NVM technologies. We will now draw a comparison between each of one of them as well as against the main current memory hierarchy components: SRAM, DRAM, disks and flash. In order to do this comparison, we will employ the following set of criteria:

1. **Maturity**: whether the technology is currently used in the market or it's in early or later research stages before being commercially mature.
2. **Density**: the number of memory cells per space unit, using standard feature size (F^2).
3. **Read Latency**: the speed of reading values from a memory cell.
4. **Write Latency**: the speed of writing values to a memory cell.
5. **Endurance**: the number of write cycles that a memory cell endures before eventually wearing out.

Table 4.2: Comparison of memory technologies. The colors indicate how well each technology scores against the specified criteria. Green means a good score, red means a bad score, yellow is for scores in between. Data gathered from [13, 24, 34, 38, 46, 53].



	SRAM	DRAM	Disk	NAND Flash	PCRAM	RRAM (Memristor)	MRAM (STT-RAM)
Maturity	Product	Product	Product	Product	Advanced development	Early development	Advanced development
Density	>100 F^2	6-8 F^2	(2/3) F^2	4-5 F^2	8-16 F^2	>5 F^2	37 F^2
Read Latency	<10 ns	10-60 ns	8.5 ms	25 µs	48 ns	<10 ns	<10 ns
Write Latency	<10 ns	10-60 ns	9.5 ms	200 µs	40-150 ns	~10 ns	12.5 ns
Energy per bit access	>1 pJ	2 pJ	100-1000 mJ	10 nJ	100 pJ	2 pJ	0.02 pJ
Static Power	Yes	Yes	Yes	No	No	No	No
Endurance	$>10^{15}$	$>10^{15}$	$>10^{15}$	10^4	10^8	10^5	$>10^{15}$
Nonvolatility	No	No	Yes	Yes	Yes	Yes	Yes

6. **Energy**: energy spent per bit access. Related with dynamic power.

7. **Static Power**: whether power needs to be spent while not accessing the memory device. This includes refreshing solid memory contents due to energy leakage or keeping disks spinning to achieve lower access latencies.

8. **Non-volatility**: whether the memory technology is volatile or not.

Based on the data presented in the last sections of this work, table 4.2 compares the discussed memory technologies against the defined set of criteria.

This concludes our survey of the new NVM technologies. The next chapter explores the potential impacts of such technologies on persistent main memory systems.

5. PERSISTENT MAIN MEMORY

The previous chapter presented the emerging NVM technologies under research. This chapter explores potential applications of these technologies to computers with persistent main memory, as well as implications in current memory system organization as described in chapter 3.

5.1 Persistent Main Memory Architectures

Main memory today is implemented almost exclusively using DRAM. Most of current non-volatile main memory ideas involve the usage of PCRAM, alone or combined with other memory technologies. Memristor is also considered a suitable replacement for DRAM, but since it is a less mature technology, there are currently no published studies exploring this scenario. In this section, the existing literature on persistent main memory architectures will be reviewed.

5.1.1 Pioneer Work: Flash-based Main Memory

In 1994, when flash memory was the only commercial NVM technology available, Wu and Zwanenepoel [74] proposed a non-volatile main memory system using a hybrid implementation of flash memory and SRAM. Their system was named "eNVy". Despite the fact that eNVy didn't use the NVM technologies we are considering in the present study, their pioneer work was precursor of several current proposed non-volatile main memory architecture, and thus worth mentioning.

They state that solid-state memories provide a factor of 100,000 improvement in access times (100ns versus 10ms) over disk and had become cheap enough to allow for arrays of a few gigabytes, although of one order of magnitude or more greater than the disk. They argue that a non-volatile main memory system could greatly improve the performance of disk-bound applications with storage needs limited to a few gigabytes, such as medium to small high performance databases. Such device would bring several additional benefits:

> "We argue that access to this permanent storage system should be provided by means of word-sized reads and writes, just as with conventional memory. This interface simplifies data access routines because there is no need to be concerned with disk block boundaries, optimizations for sequential access, or specialized disk "save" formats. Substantial reductions in code size and in instruction pathlengths can result. For backwards compatibility, a simple RAM disk program can make a memory array usable by a standard file system." [74]

Flash memory has some drawbacks when compared to DRAM: memory cannot be updated in-place (a whole block must be erased before it can be reprogrammed), memory writes are much slower and endurance is more limited. To overcome these limitations, eNVy uses copy-on-write and memory remapping to provide normal in-place update semantics and a battery-backed SRAM buffer

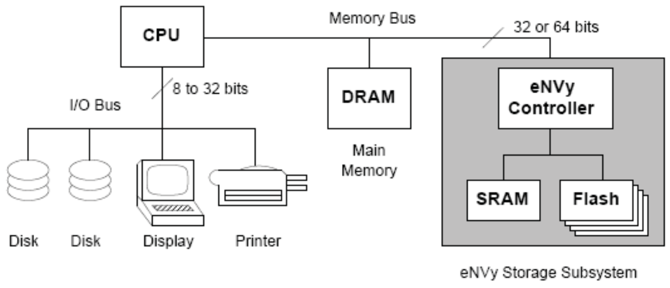


Figure 5.1: Diagram of the high-level eNVy architecture. Extracted from Wu and Zwaenepoel [74].

to hide the latency of flash write access. They also propose a cleaning algorithm to reclaim space in the flash array invalidated by copy-on-write operations. Figure 5.1 illustrates the high-level eNVy architecture [74].

5.1.2 PCRAM As Main Memory

In the 36th International Symposium on Computer Architecture (ISCA) that took place at Austin in 2009, two different papers proposed independent main memory systems based on PCRAM.

Improving latency, energy and endurance through buffer organization and partial writes

Lee et al. [36] point out that DRAM has inherent scalability limits that will be reached soon, and propose a specific PCRAM array architecture to bring this technology within competitive range against DRAM as main memory. They argue that the array design must address PCRAM's weak spots, such as write latency, high programming energy cost and finite endurance. In order to achieve this goal, they propose two strategies, which they have evaluated through simulation:

1. **Buffer organization**: reorganizing a single, wide buffer into multiple, narrow buffers reduces both energy costs and delay. Their proposed design involves four 512B-wide buffers instead of a single 2048B-wide buffer. PCRAM delay penalties are reduced from 1.60x to 1.16x when compared to DRAM.

2. **Partial Writes**: data modifications are tracked, propagating this information from the L1 cache down to the buffers at the memory banks. When a buffered row's content is flushed to the PCRAM array, only modified data is written. As the number of buffers increase, this solution becomes more efficient. Using the four buffers described above, endurance improved up to 5.6 years in the best case [36].

Improving endurance and energy consumption through 3D stacking

The work of Zhou et al. [79] tackles the same problems with a different circuit design proposal. Their paper point out the energy issues that plague DRAM today and how NVM technologies such as PCRAM can help overcoming them. They propose a 3D die stacked chip multiprocessor that puts together several processors and main memory on the same chip. This is difficult to achieve with DRAM today because such design requires tight power and thermal constraints that aren't fit for memories with high power leakage. They evaluate that the energy efficiency of using PCRAM in this design can be 65% better than DRAM.

Although PCRAM has very low static energy consumption, they acknowledge, on the other hand, that PCRAM has endurance and dynamic power problems. In order to address these problems, they propose the following actions:

1. **Removing redundant bit-writes**: in a typical DRAM implementation, a write updates the contents of an entire row (also called a page), even when a single bit value changes. Changing this behavior to write only the changed bits requires an additional read operation before writing. This wouldn't be an advantage for MRAM since both read/write operations have similar latencies and energy consumption, but PCRAM reads are faster and more efficient than writes, thus justifying this strategy. Simulations showed that endurance could increase between 3x-4.5x using this technique.

2. **Wear leveling through row shifting**: memory updates typically have high locality, with some specific sets of bits changing much more often. This creates "hot spots" that fail much sooner than the rest. This can be addressed by periodically shifting the cells in a row to even out the wear. Simulations showed that endurance can be improved around 3x with this change.

3. **Wear leveling through segment swapping**: row shifting brings some improvements to PCRAM, but not enough to achieve endurance to compete with DRAM. The problem is that some rows are written more often than others, thus failing sooner. In addition, they propose an additional wear leveling technique called segment swapping. It consists in periodically swapping memory segments of high and low write accesses. Simulations indicate that endurance can improve up to 10x by means of this change.

Their final conclusion is that using all these three techniques it is possible to increase the lifetime of PCRAM main memory from 171 days to 13~22 years, while achieving energy savings of 65% when compared to DRAM [79].

Combining both approaches

In 2010, the authors of both papers [36, 79] published an article together proposing the combination of both into a single approach [37].

5.1.3 Hybrid PCRAM-DRAM Main Memory

Qureshi et al. hybrid approach

Another paper, by Qureshi et al. [53] was presented in the ISCA '09, independently proposing a main memory system using a combination of PCRAM and DRAM. The authors stress the importance of overcoming DRAM's charge issues that prevent the technology from scaling in future years. They point out that PCRAM has the drawbacks of low endurance and slow access when compared to DRAM. To address both issues, they propose a hybrid architecture where a DRAM buffer is placed in the front of the main PCRAM storage, as depicted in Figure 5.2. They evaluate that a buffer of 3% of the size of the main storage can bridge the latency gap between DRAM and PCRAM.

In order to exploit this architecture, the following mechanisms are proposed:

1. **Lazy-write organization**: when a page fault is serviced, it is copied from disk to the DRAM buffer only, without incurring the write penalty of PCRAM. When the page is evicted from the DRAM buffer, if the page is dirty it is written to PCRAM storage. This helps decreasing latency and improve endurance.

2. **Line-level writes**: to avoid writing a complete memory page (as DRAM does), only dirty lines would be written in the PCRAM storage (this is similar to the "removing redundant bit-writes" technique mentioned before). This improves endurance.

3. **Fine-grained wear-leveling**: since memories tend to have high locality, some pages are written more often, and can wear out sooner than the rest. They propose a rotating mechanism between pages in order to distribute wear more evenly (this is similar to "wear leveling through segment swapping" mentioned earlier). This helps to increase endurance.

4. **Page Level Bypass for write filtering**: some applications not only do not benefit from the proposed architecture, but can even unnecessarily wear out the PCRAM storage faster. A typical example are streaming applications, which access a large amount of data with little reuse. The authors propose a Page Level Bypass (PLB) bit that can be enable by the operating system to indicate that the page should not be written to the PCM storage after eviction from the DRAM buffer, but merely be discarded.

The authors point out that the combination of these techniques can improve memory lifetime of a DRAM-PCRAM hybrid system from 3 years to 9.7 years. Simulations considering a 16-core system with 8GB of main memory showed that the proposed architecture can reduce page faults by 5X and provide a speedup of 3X with an area overhead of just 13% [53].

FLAM hybrid approach

In 2008 Mogul et al. [46] proposed a hybrid design for main memory combining NVM with DRAM, with two variants: flash+DRAM and PCRAM+DRAM. Their main motivation is a main

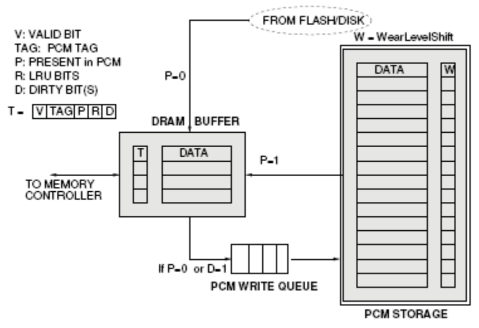


Figure 5.2: Hybrid DRAM-PCRAM main memory system. Reproduced from Qureshi et al. [53]

memory that has less cost per bit, lower power consumption and higher density. Non-volatility would be an additional benefit. They call their proposal FLAM (flash+DRAM, although they consider a scenario with PCRAM as well).

FLAM address space would be divided into different regions:

1. **NVM**: directly mapped for reading, but the CPU cannot write directly in this region.

2. **DRAM copy buffer (CB)**: can be both read and written by the CPU.

3. **Control registers**: accessed via the standard System Management Bus.

Since the CPU cannot write directly into the NVM area, there is a migration mechanism that allows for data written in the DRAM buffer region (CB) to be migrated to the NVM region. When the CPU requests a CB page P_d to be copied to an NVM page P_n, the copy can be made in background without stalling the CPU. When the copy is done, the memory controller signals the OS that can then remap the corresponding virtual page from P_d to P_n. Their solution assumes that the choice of which pages will be stored in CB or NVM regions is made by the Operating System (OS), which has better information to make this choice.

FLAM addressable space is divided in a DRAM region, that can be both directly read/written by the CPU, and a NVM region, that the CPU can read but cannot write directly - instead, it needs to issue a request to the memory controller for copying a page from the DRAM region to the NVM region. The criteria for deciding where to place a page is based on its *estimated-time-to-next-write* (ETTNW): pages with a short ETTNW should be placed at the DRAM region, and pages with a long ETTNW should be placed at the NVM region. The authors of the proposal argue that the OS should be the responsible for deciding which pages will be placed in each region, since it has some privileged information to support this decision. These are the different pieces of information that can help the OS to establish the ETTNW of a page:

- **Page type**: some page types, like stack pages and non-file pages shared between two processes, are bad candidates for NVM region. Others, such as code pages, are better candidates.

- **File type**: files such as executables, libraries, network shares are mostly read-only and can be good candidates for NVM region.

- **File reference modes**: some OSs, like Microsoft Windows, have a "temporary file" flag that can be applied to files. Flags like this are a good indicator of files that should not be placed at the NVM region.
- **Application-supplied page attributes**: certain large, memory-hungry applications (e.g.,: databases) could be adapted to provide coarse ETTNW values for the OS.
- **File usage history**: the OS can track the actual *time-to-next-write* (TTNW) of all files and use these historical data to estimate the ETTNW for each file.
- **Page usage history**: in theory the OS could do the same tracking at page level. In practice, it would probably consume a great portion of DRAM to achieve this, which would be against the point of using FLAM (to save DRAM).

Preliminary experiments were conducted using a micro-architectural simulator to verify if a significant part of main memory could be migrated to NVM without hurting overall system performance. In other words, they tried to establish if a significant portion of main memory is used during long times just for reading and not for writing, using real workloads. The results varied depending on the workload, but they found that at least half of the main memory contents stay long periods without being written (above 15 secs), which points out that the proposed solution is worth further exploration [46].

5.2 Hardware Impacts

Independently of the specific architecture that is chosen, a system with persistent storage collapsed into main memory would have impacts in several aspects, from a hardware perspective:

- **Timing**: the latency for reading/writing NVM cells tends to be higher than DRAM. On the other hand, without the need to access high-latency block devices, the overall performance should be dramatically increased for most applications.
- **Energy**: DRAM today accounts for about 30% of the energy consumption of a server, and disk is responsible for another 10% [6]. These components consume both dynamic power (used to change data) and static power (used for data retention and component availability). Static power used by DRAM is consumed mostly for memory refreshes, and can account for more than half of the total power consumption [79]; in disks, most static power is used to keep the disk spinning and ready to answer requests with low latencies. Static power consumption has also the disadvantages of increasing with the memory size and not being *energy-proportional*, i.e., it is constant over time instead of proportional to the workload [6]. Typically, NVM technologies have higher dynamic power consumption than DRAM, but negligible static power consumption. Some studies project that NVM can increase overall memory energy efficiency up to 65% [52]. The precise order of magnitude of the energy savings obtained through the

removal of disks and replacement of DRAM by NVM is not yet known, but based on these facts, it is expected to be considerable.

- **Heat**: heat and energy are two intertwined subjects. Heat emission tends to be proportional to energy dissipation. Today, air conditioning is the dominant solution to cool datacenter servers. Air conditioning, in turn, is another major power consumer. Its components, including CRACs (Computer Room Air Conditioning), chillers and humidifiers, account for about 45% of the **total** datacenter power consumption [6]. Roughly speaking, in a datacenter today for each dollar spent to power a server another dollar is spent to cool it. Since energy and heat are so interconnected, thermal efficiency should also be very positively affected by NVM adoption.

- **Space**: space is one of the less studied aspects related with the adoption of NVM. The volume occupied by the internal components of servers is likely to be significantly impacted. Magnetic disks have higher areal density (bits/square inch) than solid state memory, but require several components such as spindle, head, actuator and motor that make disk drives bulky units. The volume taken by storage media is relevant both for mobile devices and datacenters with high server density. The authors are not aware of any existing study estimating the impact of NVM on space.

- **Cost**: cost of NVM-based systems is another aspect not studied so far. Specifically for datacenters, Barroso and Holzle [6] proposed a cost model that considers the Total Cost of Ownership (TCO) of the datacenter as being composed of datacenter depreciation, datacenter operational expenses, server depreciation and server operational expenses. Their study demostrates that the electricity bill is one of the major costs of a datacenter operation, and the superior energy efficiency of NVM should have a positive effect on it. On the other hand, the cost/bit of solid-state memory today is much higher than disks: the cost of 1 MB of DRAM is about \$0.05, while 1 MB of disk costs about \$0.0003 [26]. We still don't have market prices for NVM, but it is expected that their cost/bit will also be higher than disk, possibly by a significative factor. This is another area pending study.

Looking at all discussed aspects, it becomes clear that the impact of NVM-based persistent memory cannot be adequately evaluated at component or subsystem level, but requires a holistic assessment.

Different architectural approaches can be employed. The first proposals for the application of NVM technologies evaluated the individual replacement of existing memory hierarchy levels (such as processor cache, main memory and persistent storage) by NVM counterparts, with gains of performance and efficiency at subsystem level [52]. Most of these proposals do not imply a radical redesign of computing systems as a whole, but localized changes to specific subsystems. The advantage of this approach is avoiding disruption on existing standards, with the limitation of not exploiting the full potential of these technologies.

More recently, proposals for more radical system redesigns started being published. A good example is the architecture of Nanostores [59], that proposes parallel systems with a massive number of low-cost processors co-located with non-volatile data stores. This system is targeted for data-centric workloads, such as search, sort and video transcoding. Systems such as these have the potential for more sophisticated uses of NVM technology, but depart more radically from existing standards, which probably will result in later adoption.

5.3 Software Impacts

The effects of persistent main memory systems are not limited to hardware, but potentially affect software significantly as well. In early 1980s, a research group led by Malcolm Atkinson [4] observed that most traditional programming languages provide separate abstractions for handling short-term and long-term data: short-term data is typically stored in memory, and manipulated using concepts such as arrays, records, sets, monitors and abstract data types (a.k.a objects), while long-term data is typically stored in file systems or database management systems (DBMS), and manipulated through file abstractions or DBMS models, such as the relational, hierarchical, network and functional models. The existence of two different view of data presents two problems, according to Atkinson et al. [4]:

1. In a typical program, about 30% of the code is concerned with transferring data to and from files or a DBMS. Is it necessary to spend effort in writing, maintaining and executing this code, and the quality of the application programs may be impaired by the mapping between the program's form of data and the form used for the long storage medium.

2. Data type protection offered by programming language is often lost across this mapping. The structure that might have been exploited in a program to aid comprehension is neither apparent nor protected and is soon lost.

In order to solve these problems, they propose that from a programmer's standpoint there should be no differences in the way that short-term and long-term data are manipulated, using single-level storage abstractions. Towards this goal, they proposed the concept of *orthogonal persistence* that was extensively explored in several research initiatives during the decades of 1980 and 1990, including both programming languages and operating systems. The works of Alan Dearle et al. [19, 21] provide a good overview of this topic. Despite its pioneer vision, this body of research didn't reach mainstream adoption in the computing industry due to the complexity and performance penalties of automatically mapping a single-level storage abstraction into a multi-level storage hierarchy (memory and disk/flash).

The advent of non-volatile memory persistence is expected to allow single-level storage systems with persistent memory, thus restoring interest in orthogonal persistence and other single-level storage abstractions. In a machine with persistent memory, it is even expected that such abstractions will reduce complexity and enable substantial performance improvements [16,69]. We have knowledge of three persistent memory APIs being researched: Mnemosyne [69], NV-heaps [16], and CDDS [66].

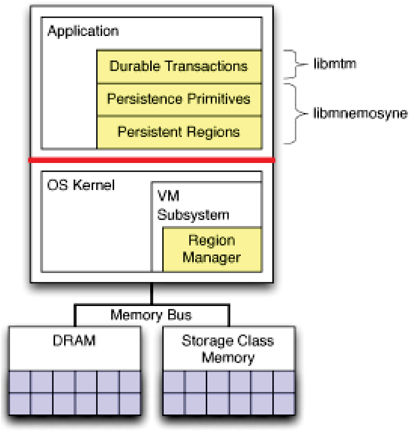


Figure 5.3: Mnemosyne architecture. Extracted from [69].

5.3.1 Mnemosyne

Mnemosyne [69] proposes an abstraction of *persistent memory* to be made available by operating systems to applications. This abstraction enables programmers to make in-memory data structures persistent without converting it to serialized formats. Direct access also reduces latency because it bypasses many software layers including system calls, file systems and device drivers. It uses transactional memory constructs in order to ensure consistency. Mnemosyne's goals are providing a simple interface for the programmer, ensure consistency across modifications and being compatible with existing commodity processors.

Mnemosyne provides a low-level programming interface, similar to C, for accessing persistent memory. It supports three main abstractions:

1. *Persistent memory regions* — segments of virtual memory mapped to non-volatile memory. Regions can be created automatically to hold variables labeled with the keyword `pstatic` or allocated dynamically. Persistent regions are virtualized by swapping non-volatile memory pages into a backing file.

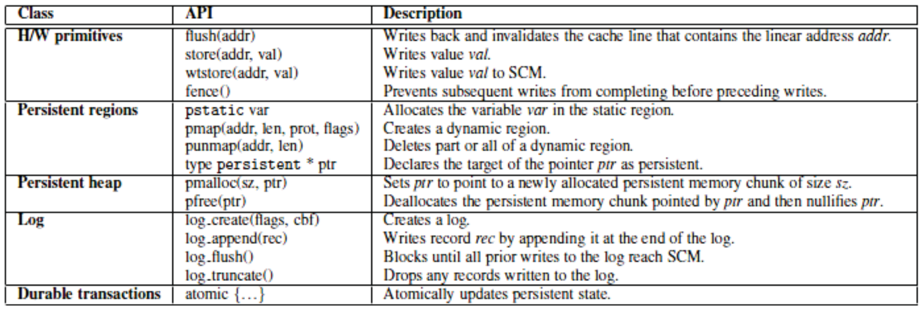
2. *Persistence primitives* — low-level operations that support consistently updating data.

3. *Durable memory transactions* — mechanism that enables consistent in-place updates of arbitrary data structures.

Mnemosyne architecture can be seen at Figure 5.3.

The Linux virtual memory system is extended by a *region manager*, responsible for memory management in the non-volatile space address. A list of the virtual pages allocated in persistent memory is maintained in a *persistent mapping table*. The region manager reconstructs persistent regions when the OS boots. Persistent pages are swapped to backing files, mapped using a variation of the `mmap` system call.

Mnemosyne provides *persistence primitives* of distinct abstraction levels, as depicted in Table 5.1. At the lowest level, there are hardware primitives to allow storing and flushing data into persistent



Class	API	Description
H/W primitives	flush(addr)	Writes back and invalidates the cache line that contains the linear address *addr*.
	store(addr, val)	Writes value *val*.
	wtstore(addr, val)	Writes value *val* to SCM.
	fence()	Prevents subsequent writes from completing before preceding writes.
Persistent regions	`pstatic` var	Allocates the variable *var* in the static region.
	pmap(addr, len, prot, flags)	Creates a dynamic region.
	punmap(addr, len)	Deletes part or all of a dynamic region.
	type `persistent` * ptr	Declares the target of the pointer *ptr* as persistent.
Persistent heap	pmalloc(sz, ptr)	Sets *ptr* to point to a newly allocated persistent memory chunk of size *sz*.
	pfree(ptr)	Deallocates the persistent memory chunk pointed by *ptr* and then nullifies *ptr*.
Log	log_create(flags, cbf)	Creates a log.
	log_append(rec)	Writes record *rec* by appending it at the end of the log.
	log_flush()	Blocks until all prior writes to the log reach SCM.
	log_truncate()	Drops any records written to the log.
Durable transactions	atomic {...}	Atomically updates persistent state.

Table 5.1: Summary of Mnemosyne's programming interface. Extracted from [69].

memory. At a higher level, variables can be automatically assigned to be persisted, both statically (through the `pstatic` declaration) and dynamically (being allocated/freed with `pmalloc`/`pfree`). A the highest level, all code inside an `atomic` block has transactional properties. Mnemosyne uses write-ahead redo logging to ensure the atomicity of transactions, and exposes its log API so application code can also make use of it for their own purposes.

The code below is an example of using Mnemosyne with static variables. Every time the program executes, the value of the integer variable `flag` will switch between 0 and 1.

```
#include <stdio.h>
#include <mnemosyne.h>
#include <mtm.h>
MNEMOSYNE_PERSISTENT int flag = 0;
int main (int argc, char const *argv[]) {
  printf("flag: %d", flag);
  MNEMOSYNE_ATOMIC {
    if (flag == 0) { flag = 1; }
    else { flag = 0; }
  }
  printf(" --> %d\n", flag); return 0;
}
```

This second example shows a simple hash table being manipulated with dynamic variables.

```
#include <mnemosyne.h>
#include <mtm.h>
MNEMOSYNE_PERSISTENT hash = NULL;
main() {
  if (!hash) {
    pmalloc(N*sizeof(*bucket), &hash);
  }
}
```

```
update_hash(key, value) {
  lock(mutex);
  MNEMOSYNE_ATOMIC {
    pmalloc(sizeof(*bucket), &bucket);
    bucket->key = key;
    bucket->value = value;
    insert(hash, bucket);
  }
  lock(mutex);
}
```

The performance of applications such as OpenLDAP and Tokyo Cabinet (a key-value store) using Mnemosyne's persistent memory increased by 35–117 percent when compared to Berkeley DB or flushing the data to disk [69].

The implementation of Mnemosyne consists of a pair of libraries and a small set of modifications to the Linux kernel for allocating and virtualizing non-volatile memory pages. It runs on conventional processors.

The purpose of Mnemosyne is to reduce the cost of making data persistent, since current programs devote large bodies of code to handling file system and/or database access. On the other hand, Mnemosyne is proposed not as a replacement for files, but as a fast mechanism to store moderate amounts of data; thus, applications should still use files for interchanging data and for compatibility between program versions when internal data-structure layouts change.

5.3.2 NV-heaps

Non-volatile Memory Heaps (NV-heaps) [16] is a persistent object system implemented specifically for non-volatile memory technologies such as PCM and MRAM. It was designed concurrently with Mnemosyne. NV-heaps was designed to present the following properties:

1. **Pointer safety** — it prevents programmers, to the extent possible, from inadvertently corrupting their data structures by pointer misuse or memory allocation errors.

2. **Flexible ACID transactions** — NV-heaps supports multiple threads accessing common data, and provides consistency across system failures.

3. **Familiar interface** — the programming interface is similar to the familiar interface for programming volatile data.

4. **High performance** — it was designed to explore as much as possible the speed of the underlying persistent memory hardware.

5. **Scalability** — NV-heaps scale to datastructures up to many terabytes.

NV-heaps proposes the following abstractions:

- **NV-heaps** — non-volatile heaps that are managed as files by a memory-based ordinary file system (which provides naming, storage management and access control), and that can be mapped by the framework directly into the program space address.
- **Persistent objects** — objects that inherit from a superclass provided by the framework (`NVObject`), and can be persisted to NV-heaps.
- **Root pointer** — every NV-heap has a root pointer from which all objects reachable are persisted.
- **Smart pointers** — smart pointers are needed to reference persistent objects. NV-heaps uses these pointers to do reference counting and manage object locks. Two types of pointers are supported: V-to-NV (volatile references to non-volatile data) and NV-to-NV (non-volatile references to non-volatile data).
- **Transactions** — programmers explicitly define the boundaries of transactions with ACID properties.

The example below exemplifies the removal of a value `k` from a persistent linked list.

```
class NVList : public NVObject {
  DECLARE_POINTER_TYPES(NVList);
public:
  DECLARE_MEMBER(int, value);
  DECLARE_PTR_MEMBER(NVList::NVPtr, next);
};
void remove(int k) {
  NVHeap * nv = NVHOpen("foo.nvheap");
  NVList::VPtr a = nv->GetRoot<NVList::NVPtr>();
  AtomicBegin {
    while(a->get_next() != NULL) {
      if (a->get_next()->get_value() == k) {
        a->set_next(a->get_next()->get_next());
      }
      a = a->get_next();
    }
  } AtomicEnd;
}
```

The `NVList` class extends `NVObject`, thus being persistent. The `DECLARE_POINTER_TYPES`, `DECLARE_MEMBER` and `DECLARE_PTR_MEMBER` macros declare the smart pointer types for NV-to-NV (`NVList::NV_Ptr`) and V-to-NV (`NVList::V_Ptr`) and declare two fields. The declarations generate private fields in the class and public accessor functions (e.g., `get_next()` and `set_next()` that provide access to data and perform logging and locking).

The program invokes `NVHOpen` to open an NV-heap and retrieves the root pointer. The block delimited by `AtomicBegin`/`AtomicEnd` atomically iterates through the nodes of the linked list and removes the one equals to `k`.

NV-heaps is implemented as a C++ library under Linux. NV-heaps are created as regular files, and mapped to to the application's virtual address using `mmap()`.

5.3.3 CDDS

The Consistent and Durable Data Structures (CDDS) [66] concept is proposed to allow atomic memory updates using commodity processors. It does not involve new abstractions for programming, but is a technique that can be used to implement efficient atomic memory updates without requiring logging or hardware changes.

A CDDS is built by maintaining a limited number of versions of the data structure with the constraint that an update should not weaken the structural integrity of an older version and that updates are atomic.

The biggest challenge behind keeping consistency while modifying memory contents is that the memory management unit (MMU) of modern processors queues memory operations (reads and writes) and can reorder them in order to access physical memory more efficiently. This is not a problem for volatile memory because a power loss would reset all memory contents, but can leave non-volatile memory in an inconsistent state. There are no processor primitives that a program can issue in order to specify the boundaries of atomic transactions.

The Intel x86 processor architecture specifies an `mfence` instruction that ensures every load and store instruction preceding it in program is globally visible before any further load or store executes. However, it only guarantees visibility, and not that the operations are actually committed to physical memory, and in the correct order. The `clflush` instruction invalidates a cache line, ensuring that it is written to physical memory. In order to emulate the semantics of an inexistent `flush` operation that would commit a set of operations to physical memory before starting another transaction, CDDS executes `mfence` and then a `clflush` in all cache lines of all modified memory regions. This ensures that it is possible to atomically write 8 byte sequences that can coupled with versioning of data structures to create CDDS.

Internally, a CDDS maintains the following properties:

- There exists a version number for the most recent consistent version. This is used by any thread which wishes to read from the data structure.
- Every update to the data structure results in the creation of a new version.
- During the update operation, modifications ensure that existing data representing older versions are never overwritten. Such modifications are performed by either using atomic opera-

tions or copy-on-write style changes.

- After all the modifications for an update have been made persistent, the most recent consistent version number is updated atomically.

Garbage collection must take place eventually to reclaim space used by obsolete versions of the CDDS. In the event of a failure in the middle of a CDDS update, during the recovery process a "forward garbage collection" takes place where all update operations executed after the most recent consistent version are discarded.

The proponents of the CDDS concept implemented a B-tree and modified the Redis in-memory distributed key-value store (KV store) to use it as backing storage. The resulting KV store (called *Tembo*) was demonstrated to increase throughput by 194%–256% when compared to a two-level KV store (Apache Cassandra).

6. MEMORY SYSTEM MODELING

The previous chapters presented the current state of memory systems, the emerging NVM technologies under research and how they can be applied to persistent main memory computers. In order to evaluate the potential impacts of such computers, it is useful to use system models. The purpose of this chapter is to describe the main techniques employed to model memory systems, including analytical models and various forms of simulation. It also includes a survey of techniques used so far for modeling non-volatile memory systems.

6.1 Overview

In their book *Simulation Modeling and Analysis* [35], Law & Kelton list the different ways that can be used to study a system (Figure 6.1):

- *Experiment with the actual system vs. experiment with a model of the system* - experimenting with the **actual system** yields the most accurate results, but often cannot be done because a) it does not exist yet; b) it is too expensive or disruptive on the existing system. For this reason, it's far more usual to work with a **model** of the system.
- *Physical model vs. mathematical model* - both **physical** and **mathematical** models are useful to obtain a better understanding of the reality being modeled, but due to cost and feasibility constraints, mathematical models are much more usual than physical models.
- *Analytical solution vs. simulation* - the most accurate type of mathematical model is the **analytical solution**, where the problem at hand is captured as a set of closed-form mathematical equations (e.g., in the $d = rt$ example, if we know both the distance and velocity, we can obtain the time using $t = d/r$). Unfortunately, some problems are too complex to be realistically modeled as an analytical solution. In these cases, a **simulation** is used, where the

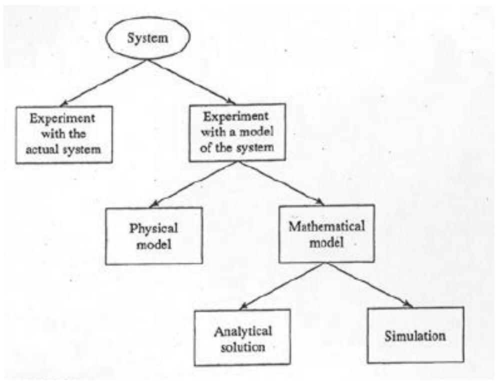

System
Experiment with the actual system
Experiment with a model of the system
Physical model
Mathematical model
Analytical solution
Simulation

Figure 6.1: Taxonomy of the ways to study a system. Extracted from [35].

model is numerically exercised for the inputs in question to check how they affect the output measures of performance [35].

Simulators can also be categorized in relation to their scope (microarchitectural or full-system) and input (trace-driven or execution-driven):

- *Microarchitectural vs. full-system simulation* - **microarchitectural** simulators can emulate the functionality and timing model of a microprocessor and its related devices, and can execute simple programs compiled for the target platform; **full-system** simulators simulate a whole computer system, and can boot a complete operating system [42].

- *Trace-driven vs. execution-driven* - in **trace-driven** simulation a sequence of instructions is captured from a real system and then used as input for the simulator; in **execution-driven** simulation, the programs are compiled for the simulator platform and executed directly over the simulator [50].

All these different means of studying a system are used to evaluate memory systems in general, and were recently used to investigate non-volatile memory systems.

In the next sections, different models and simulators used to evaluate memory systems are described in more detail.

6.2 Analytical Models

6.2.1 CACTI

CACTI is an analytical model for caches. It integrates cache and memory access time, cycle time, area, leakage, and dynamic power models. Its first version was published in 1996 [73]; since then it has been continually upgraded [49, 57, 60, 64], and is currently in version 6.5, published in 2009 [65].

CACTI accepts different inputs to characterize the system being modeled, such as cache size, line size, associativity, number of banks, and several others. Given these parameters, it applies a series of relatively simple equations that constitute the model. These equations are described in [49,57,60,64,65,73]. As output, comprehensive information on time, power and area is returned. The source code for CACTI is available at *http://www.hpl.hp.com/research/cacti/* , and is available as a web application as well in the same web site.

This model was used to assess latency, power and area of non-volatile caches in [24, 61, 63].

6.2.2 Memristor Analytical Model

In [29] there is a description of a series of closed-form equations that model a memristive memory device, at memory bank level.

6.2.3 SPICE

SPICE (Simulation Program with Integrated Circuit Emphasis) is an analog electronic circuit simulator [67]. It is maintained by the Electrical Engineering department of the University of California at Berkeley, and can be found at *http://bwrc.eecs.berkeley.edu/classes/icbook/spice/*.

It is used to model circuits containing electrical components such as resistors, capacitors, inductors, voltage and current sources, transmission lines and semiconductor devices.

SPICE was used to validate the CACTI model [73]. In [8,10,78] there is a description of equations modeling SPICE memristor devices. The modeling for SPICE MRAM cells can be found at [27, 28]. SPICE was used to model basic circuits for MRAM caches [63] and PCRAM main memory [79].

6.2.4 NVSim

NVSim is an integrated time, power and area model for PCRAM, MRAM and ReRAM-based cache and main memory systems. It also models SRAM, DRAM and eDRAM. It is available at *http://sourceforge.net/projects/nvsim/* and is an evolution of PCRAMsim [23].

NVSim usage is very similar to CACTI (described in section 6.2.1), but adds support for non-volatile memories.

6.3 Microarchitectural Simulators

As previously mentioned, microarchitectural simulators can emulate the functionality and timing model of a specific subsystem, such as a microprocessor and its related devices; they can execute simple programs compiled for the target platform or process traces captured from a real system, but they cannot boot a complete operating system [42]. In the sections below, a brief presentation of some microarchitectural simulators used to simulate memory systems is given.

6.3.1 DRAMSim

DRAMSim [70] is a cycle accurate trace-driven DRAM memory simulator. It simulates DRAM memory controller, bus and DIMM modules. It was created at the University of Maryland and is available as open-source from *http://www.ece.umd.edu/dramsim/*.

DRAMSim does not simulate processors, but can be used integrated in a complete system simulator such as GEMS (see section 6.4.2).

It is used to simulate PCRAM in [79].

6.3.2 McPAT

McPAT [40, 41] is an integrated power, area, and timing modeling framework to support comprehensive design space exploration for multicore and manycore processor configurations. At the microarchitectural level, McPAT includes models for the fundamental components of a chip multiprocessor, including in-order and out-of-order processor cores, networks-on-chip, shared caches, integrated memory controllers, and multiple-domain clocking.

At the circuit and technology levels, McPAT supports critical-path timing modeling, area modeling, and dynamic, short-circuit, and leakage power modeling for each of the device types forecast in the ITRS roadmap including bulk CMOS, SOI, and double-gate transistors. It can model components inside the memory subsystem, such as memory controllers, arrays and interconnect busses.

McPAT has a flexible XML interface to facilitate its use with performance simulators. It can be used together with M5.

6.3.3 SESC

SESC is a microarchitectural simulator that can work in both execution-driven and trace-driven modes. It was developed by the i-acoma research group at the University of Illinois at Urbana-Champaign, and is available as open-source at *http://sourceforge.net/projects/sesc/*.

SESC supports a MIPS processor. It supports different modes with distinct detail levels, so it is possible to choose between more speed or accuracy. SESC is modeled in C++ and can be easily extended.

It is one of the simulators most employed to model non-volatile memories, being used in [17, 30, 36, 61].

6.3.4 SimpleScalar

SimpleScalar is an execution-driven microarchitectural simulator. It can emulate the Alpha, PISA, ARM, and x86 instruction sets. It was released as an open-source distribution in 1995 and is available at *http://www.simplescalar.com* [5, 12].

Being a microarchitectural simulator, it enables the user to compile C or FORTRAN benchmark programs to the SimpleScalar target architecture and experiment with processor and memory hierarchy characteristics, measuring functionality and time measurements. It focus on single processor systems.

SimpleScalar was used to model MRAM memory in [24].

6.4 Full-System Simulators

A full-system simulator models a whole computer system, and can boot a complete operating system [42]. In the next sections, some of the main full-system simulators used to simulate memory systems are presented.

6.4.1 COTSon

COTSon [1] is a full-system simulator developed by HP Labs and AMD. It uses AMD's SimNow as functional simulator and adds timing simulation using statistical sampling approaches that can trade accuracy for speed. It favors speed in order to enable larger sets of data to be collected. COTSon's goal is to measure the performance of a full system composed of hundreds of multicore, multiprocessor nodes, including the full software stack and all the system devices like network cards or disks, in an affordable amount of time. COTSon is available at *http://sites.google.com/site/hplabscotson/*.

SimNow [7], the functional simulator at the core of COTSon, simulates x86 and AMD64 processors. It runs only on AMD Athlon-64 or Opteron processors.

6.4.2 GEMS

General Execution-driven Multiprocessor Simulator (GEMS) [45] is a simulation toolset built on top of Simics (see section 6.4.3) to evaluate the performance of multiprocessor hardware systems commonly used as database and web servers. It was created by the University of Winsconsin and is available at *http://www.cs.wisc.edu/gems/*. It focus on accurate performance modeling and relies on Simics for correctness details.

6.4.3 Simics

Simics [42] is an execution-driven full-system simulator. It was originally developed by the Swedish Institute of Computer Science, then moved for commercial development through a spin-off company called Virtutech, and was recently acquired by Intel's Wind River systems. It can be found at *http://www.virtutech.com/*.

Simics simulates several processors and architectures, including UltraSparc, Alpha, x86, x86-64 (Hammer), PowerPC, IPF (Itanium), MIPS, and ARM. It runs in Linux (x86, PowerPC, and Alpha), Solaris/ UltraSparc, Tru64/Alpha, and Windows 2000/x86. Simics can be used for microprocessor design, memory studies, device development and operating system development.

It was used to simulate PCRAM and MRAM in [63, 79].

6.4.4 M5

M5 [9] is an execution-driven full-system simulator. It was developed at the University of Michigan and is free software/open source and can be found at *http://www.m5sim.org*. Beyond usual system simulation, M5 was explicitly designed for accurate network simulation.

M5 can boot unmodified versions of Linux 2.4/2.6, FreeBSD, HP/Compaq Tru64 Unix, and the L4Ka::Pistachio microkernel.

It was used for modeling PCRAM in [22].

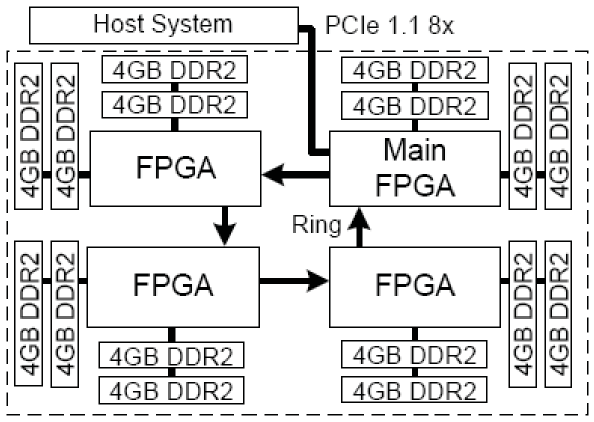


Figure 6.2: The HASTE system. A single 8x PCIe 1.1 endpoint connects for FPGAs to the host system, each controlling access to DDR2 DRAM modules. Extracted from [14].

6.5 Hardware Simulators

A simulator implemented in hardware constitutes a *physical model* of the device being evaluated. They are more expensive to build in terms of time and cost than software simulators, and are less flexible; but once they are available, experiments are much faster and accurate. As the usage of programmable logic circuits such as FPGAs (Field Programmable Gate Arrays) becomes more widespread, hardware simulation tends to be more common.

6.5.1 HASTE

HASTE (High-performance Advanced Storage Technology Emulator) [14] is a hardware simulator of non-volatile memory devices connected on the PCIe bus.

It uses DDR2 DRAM modules to store information and FPGAs to simulate the behavior of either PCRAM or STT-MRAM. The main components of HASTE can be seen in Figure 6.2. HASTE is presented, and used to evaluate the impact of NVMs in several applications, in [14].

This concludes our survey of modeling techniques and tools. In the next chapter, some of these techniques are applied in order to evaluate computers with persistent main memory.

7. EXPERIMENTAL EVALUATION OF PERSISTENT MAIN MEMORY

This chapter explains the methodology used to create an experiment to explore the system-level impacts of persistent main memory, the workloads that were utilized and presents an analysis of the final results.

7.1 Experimental Setup

In this section there is a description of the different aspects of the experimental setup used in this study, including simulation target system, tools and models.

7.1.1 Simulation Target System

In order to fully assess system-wide impacts in latency and energy, it is required to take into account all the different hardware and software layers when modeling or simulating a hypotethical computer system with persistent main memory.

For this reason, a full-system simulator was chosen for the present study, where a complete system booting a Linux Operating System was simulated using Simics [42]. The simulated computer has a standard x86 64-bit (Hammer) processor, with 4 GB of memory and 20 GB of disk space. The detailed system characteristics can be seen at Table 7.1.

Simics was chosen for being a high-quality, easy-to-use and widely adopted full-system simulator with availability of academic licenses. It can boot an operating system and allows for fine-grained instrumentation of all physical memory accesses made by the processor. This experimental setup enables us to evaluate the overall latency impact of a computer with persistent memory, both with PCRAM and Memristor technologies.

The less realistic aspect of the modeled system is the CPU clock (20 MHz); this configuration was motivated by time constraints while executing the simulated scenarios, since Simics execution time increases significantly at higher clock speeds. Nonetheless, we believe that the simulated system is adequate enough to explore the relative differences in latency and energy when comparing disk-based versus persistent memory-based workloads.

7.1.2 Memory Technology Scenarios

The memory latency and energy parameters are set to simulate three different scenarios: DRAM, PCRAM and Memristor. The latency values at device-level were derived from [36, 37, 55]. The energy parameters were obtained using CACTI [65] (for DRAM) and NVSim [23] (for PCRAM and Memristor). The technology-dependent parameters used in the experiment are displayed on Table 7.2.

Table 7.1: Experimental target configuration setup.

Processor	x86-64 (Hammer) 20 MHz (single-core)
L1 Cache	Size: 16 Kb (D-cache) + 16 Kb (I-cache) Associativity: 4-way (D-cache), 2-way (I-cache) Penalty: picosseconds Replacement policy: LRU
L2 Cache	Size: 512 Kb Associativity: 8x Penalty: 10 ns Replacement policy: LRU
Main Memory	Size: 4096 MB Penalty: technology-dependent
Disk	20 GB
OS	Fedora Core release 5 (Bordeaux) Kernel: 2.6.15-1.2054_FC5

Table 7.2: Technology parameters for DRAM, Memristor and PCRAM.

	DRAM	**Memristor**	**PCRAM**
Read Latency (ns)	50	100	50
Write Latency (ns)	48	100	150
Read Energy (nJ)	3.4539	0.5729	27.1833
Write Energy (nJ)	3.4475	1.3475	1.3501
Refresh Power (mW)	0.0867	0	0
Feature Size (nm)	45	45	45

Both PCRAM and Memristor memory technologies are fit to be used as persistent main memory, and considered in the present study. MRAM currently has limitations in density that prevent it of being a good candidate for persistent main memory (as described in chapter 4), thus it was not considered in this study.

The simulated system was configured with the specified latency and energy parameters for each scenario. No other significant changes were applied. This setup enables us to exercise variations in the memory technology parameters in a very simple commodity system. We believe that future systems with persistent main memory will have different characteristics, such as a much larger memory size (comparable to current hard disks) and different physical memory organization, since JEDEC's DDRx does not support hundreds of Gigabytes. Our purpose is not to propose a radical depart from current architectures, but to have a first-order approximation of the impact of persistent main memory in current designs. In addition, a system without radical changes will probably be a necessary step in the evolution of NVM adoption. Such a simple system allows us to understand the changes with a single element variation: the memory technology used in the main memory.

The workload scenarios were executed in a server with 32 Xeon 2.4 GHz cores, 64 GB of memory and 115 GB of local storage.

7.1.3 Energy Model

In order to estimate the overall energy consumption of each memory technology, we use an energy model that considers two separate elements:

1. **Dynamic energy** - the energy consumed in order to read/write memory addresses.
2. **Refresh energy** - the energy consumed to keep the memory contents alive. It is only relevant for DRAM, since PCRAM and Memristor don't need refresh due to their own persistent nature.

Subthreshold power leakage is a third important component of the total energy consumption. NVM should have leakage at least similar to DRAM, or probably better, since idle memory banks can be turned off without losing its contents. We will not consider leakage in this study due to the lack of published information on the leakage of memristor memory devices at this moment.

It should be noted that the energy model covers only main memory, not considering other subsystems (including disks/flash).

7.2 Experimental Workloads

The main goal concerning workload selection was to enable a comparison between a traditional implementation using files as persistent storage and a framework specific for in-memory persistence.

As framework for memory persistence, Mnemosyne [69] was chosen because it is the only existing framework available as open-source between the alternatives that match the desired characteristics for the experiment (the other alternatives were presented on Chapter 5).

7.2.1 Workload Task

As workload for comparing the performance of both a traditional implementation using files and an implementation using Mnemosyne for persistent memory storage, we have chosen the task of **indexing and searching terms in a set of text documents**. This particular workload has been chosen for a number of reasons:

1. It's a critical real-world task, performed both in personal computers (e.g. e-mail search) and huge parallel machines (e.g. web search).
2. Depends critically from persisted data, that can be easily held either in memory or in files.
3. It is a well-known task, with abundantly available litterature.

Our first choice was to adapt a popular and comprehensive open source library: CLucene (http://clucene.sourceforge.net/), a C++ version of the Apache Lucene (http://lucene.apache.org) search engine API originally written in Java. After examining CLucene's code, we concluded that the effort to adapt it would be beyond the scope of this research project, due to its complexity, richness of features and file-oriented design.

Given the unfeasibility of adapting an existing library, our choice was to create a basic search engine from scratch, using concepts presented in [43]. The features of our search engine implementation can be thus summarized:

- Generates inverted index of plain text files.
- Uses Blocked Sort-Based Indexing (BSBI) for file-based storage.
- Uses a Red-Black Tree data structure for persistent memory storage.
- Allows boolean retrieval of documents by term.

The search engine does not implement more sophisticated features such as index compression, document scoring or stemming.

7.2.2 Corpus

The corpus (set of input textual data) used in the experiment consisted in Shakespeare complete works, obtained as a set of plain text files from the University of Sydney (http://sydney.edu.au/engineering/it/~m It consists in a set of 44 documents with total size of 7.372 KB containing 24.427 distinct terms.

For each memory technology, the search engine was exercised in three distinct scenarios:

1. **Index Generation** — the task of reading the complete corpus and generating an inverted index, either in memory or disk.
2. **Search 1 (simple)** — the task of searching the index for a given term (“Brutus").
3. **Search 2 (intersection)** — the task of searching the index for two given terms (“Brutus" and “Calpurnia") and intersecting the results (logical “AND").

7.2.3 Workload Design and Implementation

The search engine is organized around three main subsystems:

1. **Index and Search Algorithms** — the common algorithms used for indexing and searching terms, independently of the persistent storage.
2. **File-based Storage** — a set of classes implementing storage in the file system using Blocked Sort-Based Indexing (BSBI).
3. **In-memory Storage** — a set of classes implementing storage in-memory using a Red-Black Tree persisted using Mnemosyne.

The following subsections describe each one of these subsystems.

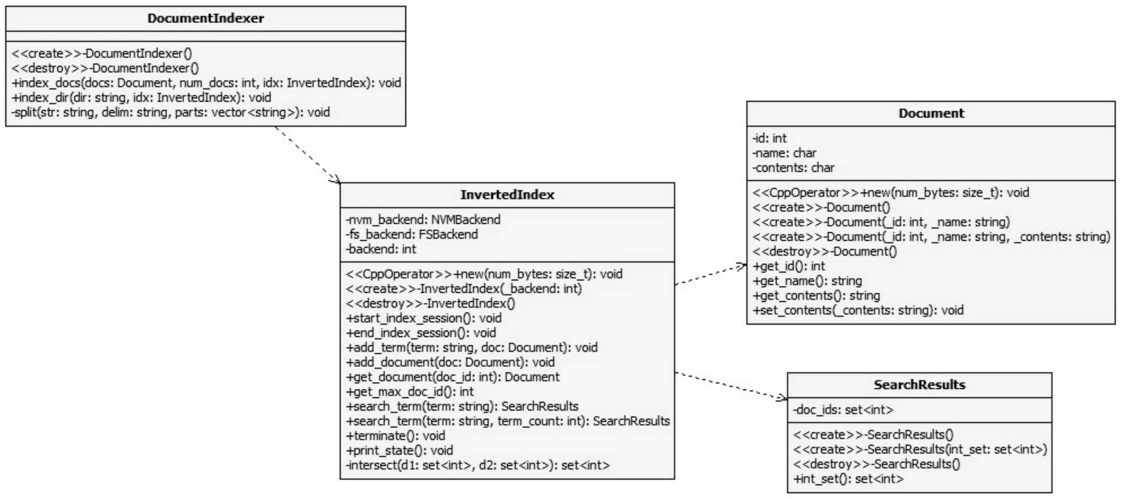

DocumentIndexer
<<create>>-DocumentIndexer()
<<destroy>>-DocumentIndexer()
+index_docs(docs: Document, num_docs: int, idx: InvertedIndex): void
+index_dir(dir: string, idx: InvertedIndex): void
-split(str: string, delim: string, parts: vector<string>): void
InvertedIndex
-nvm_backend: NVMBackend
-fs_backend: FSBackend
-backend: int
<<CppOperator>>+new(num_bytes: size_t): void
<<create>>-InvertedIndex(_backend: int)
<<destroy>>-InvertedIndex()
+start_index_session(): void
+end_index_session(): void
+add_term(term: string, doc: Document): void
+add_document(doc: Document): void
+get_document(doc_id: int): Document
+get_max_doc_id(): int
+search_term(term: string): SearchResults
+search_term(term: string, term_count: int): SearchResults
+terminate(): void
+print_state(): void
-intersect(d1: set<int>, d2: set<int>): set<int>
Document
-id: int
-name: char
-contents: char
<<CppOperator>>+new(num_bytes: size_t): void
<<create>>-Document()
<<create>>-Document(_id: int, _name: string)
<<create>>-Document(_id: int, _name: string, _contents: string)
<<destroy>>-Document()
+get_id(): int
+get_name(): string
+get_contents(): string
+set_contents(_contents: string): void
SearchResults
-doc_ids: set<int>
<<create>>-SearchResults()
<<create>>-SearchResults(int_set: set<int>)
<<destroy>>-SearchResults()
+int_set(): set<int>

Figure 7.1: UML class diagram of index and search subsystem.

Index and Search Algorithms

This subsystem contains common algorithms used for indexing and searching terms, independently of the persistent storage (file system or in-memory). The main classes that comprise this subsystem are depicted on Figure 7.1, and described below:

- **DocumentIndexer** — class responsible for indexing a set of documents and returning a corresponding InvertedIndex.
- **InvertedIndex** — one of the main classes of the search engine, it represents the invertex index itself. In order to search the index, the `search_term()` methods should be called by application code. It uses internally a decorator for implementing the actual index/search routines, and this decorator implements either a file-system backend (`FSBackend`) or an in-memory persistence version (`NVMBackend`).
- **Document** — class representing a document, with its ID, name and contents.
- **SearchResults** — set of documents returned by `InvertedIndex::search_term()`.

File-based Storage (BSBI)

This subsystem implements storage in the file system using Block Sort-Based Indexing (BSBI). The concept of BSBI is described with details in [43]. For each document and each term unique numeric IDs are assigned. The index stores in a file a pair termID/docID for every document that contains a given term. A set of files containing such pairs compose the BSB Index. The contents of the files are always sorted by termID, in order to make the queries more efficient (only the files

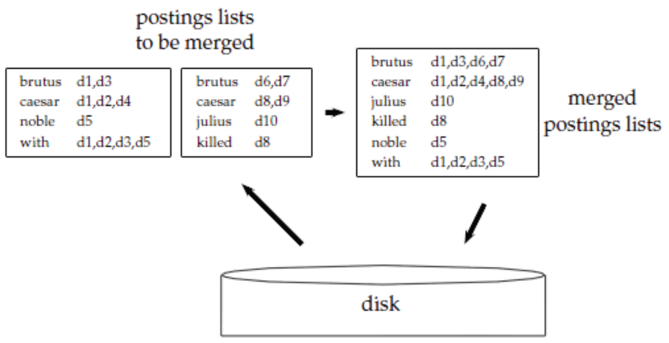


Figure 7.2: Merging in blocked sort-based indexing. Two blocks (“postings lists to be merged") are loaded from disk into memory, merged in memory (“merged postings lists") and written back to disk. We show terms instead of termIDs for better readability. Extracted from [43].

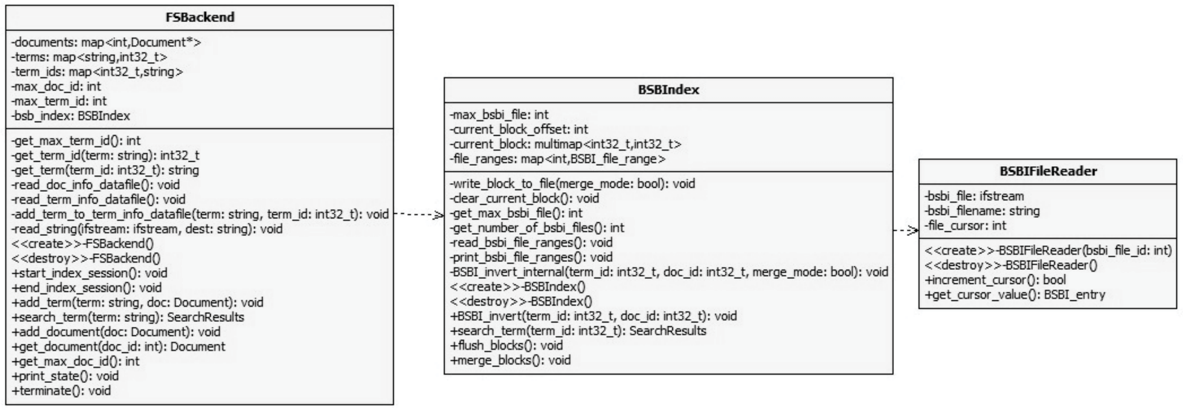

FSBackend
-documents: map<int,Document*>
-terms: map<string,int32_t>
-term_ids: map<int32_t,string>
-max_doc_id: int
-max_term_id: int
-bsb_index: BSBIndex
-get_max_term_id(): int
-get_term_id(term: string): int32_t
-get_term(term_id: int32_t): string
-read_doc_info_datafile(): void
-read_term_info_datafile(): void
-add_term_to_term_info_datafile(term: string, term_id: int32_t): void
-read_string(ifstream: ifstream, dest: string): void
<<create>>-FSBackend()
<<destroy>>-FSBackend()
+start_index_session(): void
+end_index_session(): void
+add_term(term: string, doc: Document): void
+search_term(term: string): SearchResults
+add_document(doc: Document): void
+get_document(doc_id: int): Document
+get_max_doc_id(): int
+print_state(): void
+terminate(): void
BSBIndex
-max_bsbi_file: int
-current_block_offset: int
-current_block: multimap<int32_t,int32_t>
-file_ranges: map<int,BSBI_file_range>
-write_block_to_file(merge_mode: bool): void
-clear_current_block(): void
-get_max_bsbi_file(): int
-get_number_of_bsbi_files(): int
-read_bsbi_file_ranges(): void
-print_bsbi_file_ranges(): void
-BSBI_invert_internal(term_id: int32_t, doc_id: int32_t, merge_mode: bool): void
<<create>>-BSBIndex()
<<destroy>>-BSBIndex()
+BSBI_invert(term_id: int32_t, doc_id: int32_t): void
+search_term(term_id: int32_t): SearchResults
+flush_blocks(): void
+merge_blocks(): void
BSBIFileReader
-bsbi_file: ifstream
-bsbi_filename: string
-file_cursor: int
<<create>>-BSBIFileReader(bsbi_file_id: int)
<<destroy>>-BSBIFileReader()
+increment_cursor(): bool
+get_cursor_value(): BSBI_entry

Figure 7.3: UML class diagram of file-based storage subsystem.

containing the desired terms are scanned when a query is issued). The BSBI concept is depicted in Figure 7.2.

The main classes that comprise this subsystem are depicted on Figure 7.3, and described below:

- **FSBackend** — this is the decorator class which realizes most of the methods defined by the class `InvertedIndex` using BSBI files as backend.
- **BSBIndex** — class that models a set of BSBI files internally used by `FSBackend`.
- **BSBIFileReader** — helper class to assist BSBIndex in reading BSBI files.

In-memory Storage (Mnemosyne)

This subsystem implements in-memory storage using a Red-Black Tree and the Mnemosyne framework. The main classes that comprise this subsystem are depicted on Figure 7.4, and described below:

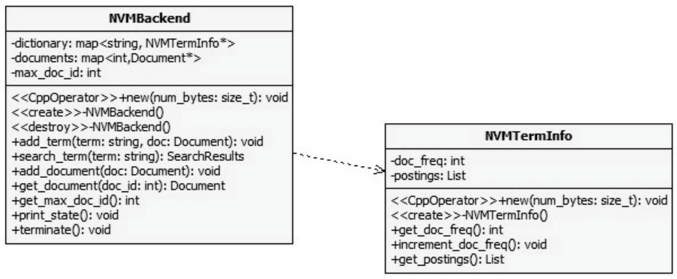

NVMBackend
-dictionary: map<string, NVMTermInfo*>
-documents: map<int,Document*>
-max_doc_id: int
<<CppOperator>>+new(num_bytes: size_t): void
<<create>>-NVMBackend()
<<destroy>>-NVMBackend()
+add_term(term: string, doc: Document): void
+search_term(term: string): SearchResults
+add_document(doc: Document): void
+get_document(doc_id: int): Document
+get_max_doc_id(): int
+print_state(): void
+terminate(): void
NVMTermInfo
-doc_freq: int
-postings: List
<<CppOperator>>+new(num_bytes: size_t): void
<<create>>-NVMTermInfo()
+get_doc_freq(): int
+increment_doc_freq(): void
+get_postings(): List

Figure 7.4: UML class diagram of persistent memory storage subsystem.

- **NVMBackend** — this is the decorator class which realizes most of the methods defined by the class `InvertedIndex` using BSBI files as backend.
- **NVMTermInfo** — class that represents a term with an associated set of documents.

The actual implementation of the Red-Black Tree that is used as persistent memory structure is not implemented in C++, but in C. It is a modified version of the open-source Red-Black Tree provided at `http://www.mit.edu/~emin/source_code/red_black_tree/index.html`.

The main changes required by Mnemosyne are essentially two:

1. Ensure that all C data structures that must be persisted are allocated with the Mnemosyne function `pmalloc()`:

```
char* key = (char*) pmalloc(sizeof(char) * len);
```

2. Ensure that all C++ objects that will be persisted use the Mnemosyne function `pmalloc()` in the `new()` operator implementation:

```
/**
 * Operator new using pmalloc for persistent heap allocation (Mnemosyne).
 */
void *operator new(size_t  num_bytes) {
    void* result;
    result = pmalloc(num_bytes);
    return result;
}
```

7.2.4 Development Metrics

The search engine was implemented using the development environment described in table 7.3.

Table 7.4 displays the main metrics related with development size, complexity and effort. Each metric is described here:

Table 7.3: Workload Development Environment

Tool	Version
Operating System	Ubuntu 9.10 (Karmic Koala)
Compiler	Intel C Compiler (ICC) Prototype Edition 4.0 20100806
Mnemosyne	Mnemosyne 0.1 usermode
IDE	Eclipse for C/C++ Indigo Release

- **LOC (Lines of Code)** — the number of lines of code, excluding comments and empty lines.
- **Comments** — the number of lines of comments.
- **# Classes** — the number of classes.
- **Dev. Hours** — the amount of development hours spent in each subsystem. Since the Red-Black Tree code was obtained from an open-source implementation, the time spent in this component is relative to integration into the search engine and adaptations necessary to make it persistent with Mnemosyne.

These metrics were chosen to allow a comparison of the distinct scenarios in terms of development efficiency. The number of lines of code and comments help understanding the software size. The number of classes when compared with the number of lines of code can be used as an indicator of complexity. The number of development hours informs the effort applied to develop each scenarios. These metrics are important for the inference of some of the analyses presented later on this chapter.

7.2.5 Development Challenges

The main challenges faced using Mnemosyne (which should be applicable to most in-memory persistence frameworks) can be so described:

1. **All objects must override the new() operator to use pmalloc()** — in order to be persisted using Mnemosyne, all C++ objects must override the `new()` operator to use `pmalloc()`. In concrete terms, it prevents the programmer from using any C++ frameworks without modifying its source code. This makes it unpractical to use essential frameworks, such as the Standard Template Library (STL). While porting the search engine, all references to STL's string type had to be converted to `char*` in order to be persisted.
2. **Data cannot be reused/shared between different programs** — this is possibly the greatest challenge for in-memory persistence frameworks: how to enable data in a given memory area to be bound to different programs, potentially using distinct programming languages. In the search engine created for this study, the data stored in the BSBI files could be reused by other programs written in diverse languages, but the data persisted in memory through Mnemosyne can only be used by the program which created it.

Table 7.4: Workload development: size and effort metrics

Scope	LOC	Comments	# Classes	Dev. Hours
Total	1783	1166	13	77 hours
Index & Search	603	348	7	30 hours
File System	499	429	3	35 hours
In-Memory	241	34	3	8 hours
Red-Black Tree	440	355	N/A	4 hours

7.3 Execution Results and Analysis

Each scenario was exercised three times and the variation between the execution results was negligible, as expected. The running time for each execution ranged between a few minutes and around 10 hours.

The execution times for each scenario are shown in Table 7.5. As expected, there is a degradation of performance in Memristor and PCRAM compared to DRAM. Most of the deviations are between 7–32% slower than DRAM. Some scenarios presented a huge difference (e.g. Search 2 Mem), but it is more likely that these are distortions caused by the experimental environment than actual differences. The fact that not only the application workload, but a complete OS with all its standard daemons was running in background contributes to some of these differences. This trade-off was consequence of our choice of evaluating a more realistic software environment.

Although Memristor is considered here 2x slower than DRAM and PCRAM write latency 3x slower than DRAM, the average overall system performance impact is less steep. This is consistent with the results published in similar studies [22, 36, 37, 53, 79]. The main factor behind this phenomenon is the high rate of L1 and L2 cache hits in a typical workload (the cache hits can be seen at Table 7.6).

A consequence of these observations is that main memory using Memristor or PCRAM cells still need processor caches in order to avoid severe performance penalties. Design proposals for persistent caches are explored in [24, 33, 75, 76].

As expected, the workloads using persistent memory are dramatically faster than their counterparts using disk for persistence. Indexing the input text files using in-memory persistence is more than 3.5x faster than using disk for the index. The difference for loading the index is remarkably 8.3×10^5x faster using memory persistence. The search time in memory is in average more than 53x faster than its disk counterpart.

The main impact of non-volatile memories is on the energy footprint of main memory, which is significantly smaller in non-volatile technologies than DRAM. The detailed energy consumption results for each workload/technology can be seen on Table 7.7. Using memristor with in-memory persistence consumes 20x less memory energy than using memristor main memory with filesystem, and 80x less memory energy than using DRAM/Disk with a filesystem approach.

When comparing Memristor and PCRAM as base technology for persistent main memory, we observe that PCRAM has a slight advantage regarding speed of execution, but Memristor is signifi-

Table 7.5: Execution times for experimental workloads, in seconds. Time deviation of Memristor and PCRAM relative to DRAM are shown as percentage (%).

	DRAM	**Memristor**	**PCRAM**
Index Mem	1913.51	2703.35 (41%)	2519.02 (32%)
Index Disk	6900.45	8535.59 (24%)	7193.05 (4%)
Index Load Mem	0.00061	0.00066 (7%)	0.00064 (3%)
Index Load Disk	384.11	538.05 (40%)	488.44 (21%)
Search 1 Mem	0.25637	0.27713 (8%)	0.13794 (-46%)
Search 1 Disk	21.98	27.64 (26%)	28.97 (32%)
Search 2 Mem	0.02339	1.03386 (4420%)	1.22842 (5252%)
Search 2 Disk	23.62	31.33 (33%)	33.69 (43%)

Table 7.6: Cache hits during experimental workloads.

		L1 Read	**L1 Write**	**L2 Read**	**L2 Write**
DRAM	**Index Mem**	94.63%	93.55%	94.73%	99.91%
	Index Disk	93.35%	78.08%	95.02%	99.87%
	Search 1 Mem	95.86%	73.29%	92.04%	99.76%
	Search 1 Disk	94.75%	82.85%	94.11%	99.83%
	Search 2 Mem	95.69%	74.93%	92.68%	99.77%
	Search 2 Disk	94.76%	83.03%	94.21%	99.83%
Memristor	**Index Mem**	94.59%	93.64%	94.81%	99.91%
	Index Disk	93.46%	80.07%	94.83%	99.87%
	Search 1 Mem	95.75%	74.98%	92.41%	99.77%
	Search 1 Disk	94.72%	85.74%	94.38%	99.85%
	Search 2 Mem	95.55%	77.00%	92.89%	99.78%
	Search 2 Disk	94.72%	85.90%	94.47%	99.85%
PCRAM	**Index Mem**	94.61%	93.63%	94.78%	99.91%
	Index Disk	93.36%	87.79%	95.13%	99.86%
	Search 1 Mem	95.71%	74.89%	92.56%	99.77%
	Search 1 Disk	94.69%	84.99%	94.30%	99.85%
	Search 2 Mem	95.53%	77.55%	92.89%	99.79%
	Search 2 Disk	94.69%	84.95%	94.33%	99.85%

cantly superior from the energy savings standpoint.

Another major benefit of using in-memory persistence is relative to application development and complexity. Table 7.4 shows that developing using in-memory persistence produces less than half lines of code and is more than 4x faster to develop than using disk.

These results support the case for not only replacing current hardware designs using NVM, but also take different approaches at software level, such as in-memory persistence. However, in order to successfully replace traditional file-system or database approaches, it is necessary to have in-memory persistence frameworks that allow sharing data seamlessly between different programs and languages, as the current approaches do. This is a still pending research problem, and the next chapter will

Table 7.7: Energy consumption of experimental workloads, in J.

		Dynamic (J)	Refresh (J)	Total (J)
DRAM	**Index Mem**	15.693	165.901	181.594
	Index Disk	29.793	598.269	628.062
	Search 1 Mem	0.152	0.022	0.174
	Search 1 Disk	3.329	34.975	38.304
	Search 2 Mem	0.174	0.002	0.176
	Search 2 Disk	3.345	35.585	38.929
Memristor	**Index Mem**	16.262	0.000	16.262
	Index Disk	8.235	0.000	8.235
	Search 1 Mem	0.035	0.000	0.035
	Search 1 Disk	0.861	0.000	0.861
	Search 2 Mem	0.041	0.000	0.041
	Search 2 Disk	0.862	0.000	0.862
PCRAM	**Index Mem**	83.912	0.000	83.912
	Index Disk	35.417	0.000	35.417
	Search 1 Mem	0.833	0.000	0.833
	Search 1 Disk	19.100	0.000	19.100
	Search 2 Mem	0.975	0.000	0.975
	Search 2 Disk	19.001	0.000	19.001

discuss the previous research concerning this topic and suggest themes for future investigation.

8. FUTURE RESEARCH PERSPECTIVES

As presented in the previous chapters, there are several advantages in using persistent main memory instead of separating the memory hierarchy in volatile main memory and persistent storage. Disks are widely employed today not for any goal-directed design rationale, but for the fact that today's technology standards make them extremely cost-effective, and it has been so since the beginning of commercial computing. As Alan Cooper [18] pointed out, "Disks do not make computers better, more powerful, faster, or easier to use. Instead, they make computers weaker, slower, and more complex. (...) Wherever disk technology has left its mark on the design of our software, it has done so for implementation purposes only, and not in the service of users or any goal-directed design rationale."

But decades of separation between volatile main memory and persistent storage have left deep marks in computer design, including hardware, basic software and application software. If the emerging memory technologies fulfill their promise of enabling persistent main memory, several layers of computer design will be affected.

Changing hardware, operating systems and programming languages internal implementation can be challenging for computer scientists and engineers, but they potentially have a mild impact on application software, when compared to conceptual changes in programming languages. Nonetheless, we believe that in order to reap the major rewards potentially offered by persistent main memory, it will be necessary to change the way application developers approach programming. Current programming interfaces have separate abstractions for handling memory (data structures) and secondary storage (files, databases). A good deal of development effort is directed towards moving data between these two layers; a seminal study estimates around 30% [4]; our own observations during this study show that developing using in-memory persistence produces less than half lines of code and is more than 4x faster to develop than using disk. In order to employ unified abstractions that model a single-level storage system, thus avoiding unnecessary effort, we need to change the programming interfaces. There are some options for the interface style, including orthogonal persistence, transactional memory, persistent stores and non-volatile heaps, and it is possible for these different styles to coexist at different levels (for instance, it is possible for a high-level object-oriented database to be implemented over persistence heaps).

Fortunately, the concept of abstracting away the difference between memory layers in programming languages is not a new idea. An extensive research has been conducted during more than 30 years to craft the concept of "orthogonal persistence" to that end. In order to make progress towards this goal, we suggest that this body of research must be revisited. In the next section, we will review the major topics regarding the evolution of this concept.

8.1 Orthogonal Persistence

In early 1980s, a research group led by Malcolm Atkinson proposed the concept of **orthogonal persistence** [4]. They observe that most programming languages provide separate abstractions for handling short-term and long-term data: short-term data is typically stored in memory, and manipulated using concepts such as arrays, records, sets, monitors and abstract data types (a.k.a objects), while long-term data is typically stored in file systems or database management systems (DBMS), and manipulated through file abstractions or DBMS models, such as the relational, hierarchical, network and functional models. The existence of two different view of data presents two problems, according to Atkinson et al.:

1. In a typical program, about 30% of the code is concerned with transferring data to and from files or a DBMS. Is it necessary to spend effort in writing, maintaining and executing this code, and the quality of the application programs may be impaired by the mapping between the program's form of data and the form used for the long storage medium.

2. Data type protection offered by programming language is often lost across this mapping. The structure that might have been exploited in a program to aid comprehension is neither apparent nor protected and is soon lost.

In order to solve these problems, they propose that from a programmer's standpoint there should be no differences in the way that short-term and long-term data are manipulated. In other words, persistence should be an *orthogonal property of data*, independent of data type and the way in which data is manipulated, thus the term *orthogonal persistence*. During the decades of 1980 and 1990, this concept was explored in several research initiatives, including both programming languages and operating systems. The works of Alan Dearle et al. [19, 21] provide a good overview of this research, and the following sections were largely based on these texts.

The complexity of mapping application data as represented in memory to either files or databases and vice-versa has been described as an **impedance mismatch**, and has been described as *The Vietnam of Computer Science*. Orthogonally persistent object systems propose to solve this problem by supporting a uniform treatment of objects irrespective of their types by allowing values of all types to have whatever longevity is required. The benefits of orhogonal persistence can be summarized as [21]:

- improving programming productivity from simpler semantics;
- avoiding ad hoc arrangements for data translation and long-term data storage;
- providing protection mechanisms over the whole environment;
- supporting incremental evolution; and
- automatically preserving referential integrity over the entire computational environment for the whole life-time of an application.

Atkinson and Morrison [3] identified three principles that, if applied, would yield orthogonal persistence:

1. **The Principle of Persistence Independence**: the form of a program is independent of the longevity of the data it manipulates. Programs look the same whether they manipulate short-term or long-term data.

2. **The Principle of Data Type Orthogonality**: all data objects should be allowed the full range of persistence irrespective of their type. There are no special cases where objects are not allowed to be long-lived or are not allowed to be transient.

3. **The Principle of Persistence Identification**: the choice of how to identify and provide persistent objects is orthogonal to the universe of discourse of the system. The mechanism for identifying persistent objects is not related to the type system.

These principles were applied in different degrees in several experimental programming languages and operating systems. The next sections present information on these initiatives.

8.1.1 Orthogonal Persistence and Programming Languages

PS-algol

The first language to provide orthogonal persistence was PS-algol [4,21]. It extended the existing S-algol language, providing **persistence by reachability**, where the identification of data to be persisted is made by reachability from a persistent root. Later, experience would show that this mechanism, also known as **transitive persistence**, is the only one that can be used to implement orthogonal persistence in languages that support references. It added the following abstractions to S-algol:

- **Tables** — associative stores (hash maps). A few functions are added to handle tables, such as `s.lookup`, which retrieves a value associated with a key in a table, and `s.enter`, which creates an association between a key and a value. Strictly speaking, tables are not directly related with persistence, but are useful abstractions that were not originally part of S-algol.

- **Databases** — the roots of persistence in PS-algol, containing a pointer to a table. Everything referenced directly or indirectly by this table (including other tables) is identified as persistent. Databases can be dynamically created, and are manipulated through the `open.database` and `close.database` functions.

- **Transactions boundaries** — explicitly defined by the `commit` and `abort` functions.

In order to give a flavour of the language, two examples will be provided. The first example opens a database called “addr.db” and places a person object into a table associated with the key “addr.table” found at its root. When `commit` is called, the persistence of all objects reachable from the root is ensured, including the updated table, the person object and the address object referenced by the person object.

```
structure person(string name, phone; pntr addr)
structure address(int no ; string street, town)
let db = open.database("addr.db", "write")
if db is error.record
    do { write "Can’t open database"; abort }
let table = s.lookup("addr.table", db)
let p = person("al", 3250,
                address(76, "North St", "St Andrews"))
s.enter("al", table, p)
commit
```

The second example opens the same database, retrieves the same person object and writes out their phone number:

```
structure person(string name, phone; pntr addr)
structure person (string name, phone; pntr addr)
let db = open.database("addr.db", "read")
if db is error.record
    do { write "Can’t open database"; abort }
let table = s.lookup("addr.table", db)
let p = s.lookup("al", table)
if p = nil then write "Person not known"
else write "phone number: ", p(phone)
```

A second version of PS-algol considered procedures as data objects, this allowing both code and data to be persisted. PS-algol had no support for concurrency (other than at database level), which often caused problems.

Napier88

Napier88 [21, 48] consisted not only of a language, but a complete self-contained persistent system, incorporating the whole language support within a strongly typed persistent store. It provides a pre-populated strongly typed stable store containing libraries for handling concurrency, distribution and user transactions, and a complete programming environment. Figure 8.1 displays the layered architecture of the Napier88 system.

Napier88 borrows from PS-algol the notion of persistence by reachability from a persistent root, but adds capabilities of dynamic type coercion, allowing the programmer to explicitly specify the type that a value being read from the persistent store will assume in the current context. This can be visualized in the following example, that extracts a value from the persistent store, projects it into the `person` type and changes its name and address:

```
type person is structure (name, address : string)
let ps = PS()
project ps as X onto
person:
    begin
        X(name) := "Ronald Morrison"
        X(address) := "St Andrews"
    end
default: {}
```

In the first line, the `person` type is declared. The `PS()` procedure, that returns the root of the persistent store, is then called and assigned to the variable `ps`. The `ps` variable is coerced to the type `person`. If this coercion is successful, a code block is executed to set the name and address of these types to constant values. If the coercion is not successful, the `default` option is executed (in this particular case, in case of failure the program does nothing). In order for the type coercion to be successful, the root of the persistent store must contain a pointer to data that can be cast to the `person` type.

The main additions that Napier88 introduces over PS-algol are listed below [21]:

- the infinite union type `any`, which facilitates partial and incremental scpecification of the structure of the data;

- the infinite union type `environment`, which provides dynamically extensible collections of bindings;

- parametric polymorphism, in a style similar to Java generics and C++ templates;

- existentially quantified abstract data types;

- a programming environment, with graphical windowing library, object browser, program editor and compiler, implemented as persistent objects withing the store;

- support for hyper-code, in which program source code can contain embedded direct references to extant objects;

- support for structural reflection, allowing a program to generate new program fragments and integrate those into itself.

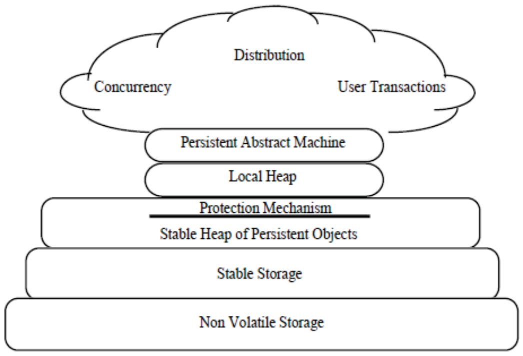


Figure 8.1: Napier88 layered architecture. Extracted from [48].

Persistent Java

During the second half of the 1990s, several orthogonally persistent versions of Java have been implemented.

Malcolm Atkinson and Mick Jordan led the creation of a prototype called PJama [2] which was the base for the definition of the Orthogonal Persistence for Java (OPJ) specification [32]. It implements persistence by reachability from the same roots used during garbage collection (the set of live threads and the static variables of the set of loaded classes) and from explicit persistent roots. The complete Java Virtual Machine (JVM) is checkpointed periodically, and its execution may be resumed in an arbitrary moment of the future from one of these checkpoints. The `Runtime` class is augmented with two methods: `checkpoint`, to explicitly generate a checkpoint and continue execution afterwards, and `suspend` to generate a checkpoint and then stop JVM execution.

Some of the most notable limitations/challenges of the PJama implementation are:

1. Handling of bulk types — There is no query language for the manipulation of large object collections.

2. Conflict with traditional Java programming constructs — Java is based in some concepts that follow a traditional programming approach and are somewhat conflicting with orthogonal persistence principles, such as the `transient` keyword definition, the extensive usage of native code (via Java Native Interface - JNI) for windowing frameworks and handling of external resources, such as files and network-related structures.

3. Program evolution — in OPJ/PJama, once one object is created, its class cannot be changed. This presents serious challenges for program evolution.

ANU-OPJ [44] implemented persistence transparently through the dynamic manipulation of byte-codes. All objects reachable from the roots using during garbage collection were considered persistent and stored in the Shore storage manager.

Persistent Java was implemented in the Grasshopper operating system [20]. Grasshopper supported persistence at operating system level, so no modifications were made to the Java language or virtual machine. All state was persisted in this implementation.

8.1.2 Limitations of Orthogonal Persistence

It is important to notice that orthogonal persistence does not imply the presence of persistent main memory. In fact, the systems that so far implemented experimentally orthogonal persistence handled internally the movement between memory and secondary storage, removing this burden from the programmer at the cost of increasing the integral complexity of these systems. The price paid in performance by doing so is one of the most significant reasons that prevented orthogonally persistent systems from becoming a mainstream technology. However, the introduction of persistent main memory solves this problem, since there are no more two different storage levels to be handled by the applications and the operating system.

The existence of different abstractions for short-term and long-term data representation and storage is not a fundamental necessity of programming languages, but an accidental result of the current technology that mandate the separation of fast, small, volatile main memory from slow, abundant, nonvolatile secondary storage. If these two different layers could be collapsed into a single fast, abundant and nonvolatile storage device, orthogonal persistence would become a very appropriate abstraction to model data handling.

It is important to notice that certain data should explicitly be volatile, including sensitive data (such as passwords and encryption keys), and persistent software systems need to be able to safely address these needs. It is also necessary to have safeguards preventing a persistent system to reach an inconsistent state from which it cannot be recovered.

9. CONCLUSION

In this work we presented a preliminary evaluation of workloads in a hypothetical computer with persistent main memory, through the use of experimental models and simulations, aiming to identify the major system-level impacts of persistent main memory in latency and energy.

It was observed that main memory using Memristor or PCRAM cells still need processor caches in order to avoid severe performance penalties. Overall performance of non-volatile technologies in main memory is similar to current DRAM results.

The main impact of non-volatile memories is on the energy footprint of main memory, which is significantly smaller in non-volatile technologies than DRAM.

The great challenge of using such technologies as main memory is their low endurance. Wear leveling techniques such as those described in [36, 37, 53, 79] are expected to contribute positively.

The experimental results support the feasibility of employing emerging non-volatile memory technologies as persistent main memory. Many architectural changes are required in order to replace a three-level storage hierarchy (caches, main memory, disk) by a two-level hierarchy (caches, main memory), including memory devices, memory subsystem organization and operating system support, but our preliminary study, considering only changes in the memory technology, indicates that performance penalties should be mild and energy improvements should be significant.

This study has also compared the development and execution of applications using both a traditional filesystem design and a framework specific of in-memory persistence (Mnemosyne). We concluded that in order to reap the major rewards potentially offered by persistent main memory, it is necessary to take new programming approaches that do not separate volatile memory from persistent secondary storage. Current programming interfaces have separate abstractions for handling memory (data structures) and secondary storage (files, databases), and a good deal of development and processing effort is directed towards moving data between these two layers. Our observations during this study show that developing using in-memory persistence produces less than half lines of code and is more than 4x faster to develop than using disk.

We suggest that future research efforts should consider revisiting the concept of orthogonal persistence in order to create programs that are better suited to a persistent main memory architecture. The major challenges currently are related with finding appropriate mechanisms for sharing data between different programs and programming languages.

Bibliography

[1] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. Cotson: infrastructure for full system simulation. *ACM SIGOPS Operating Systems Review*, 43(1):52–61, 2009.

[2] M. Atkinson and M. Jordan. A review of the rationale and architectures of pjama: a durable, flexible, evolvable and scalable orthogonally persistent programming platform. *Sun Microsystems, Inc. Mountain View, CA, USA*, 2000.

[3] M. Atkinson and R. Morrison. Orthogonally persistent object systems. *The VLDB Journal*, 4(3):319–401, 1995.

[4] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, P.W. Cockshott, and R. Morrison. An approach to persistent programming. *The computer journal*, 26(4):360, 1983.

[5] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[6] L.A. Barroso and U. Holzle. *The datacenter as a computer: An introduction to the design of warehouse-scale machines*, volume 4. Morgan & Claypool Publishers, 2009.

[7] R. Bedichek. SimNow: Fast platform simulation purely in software. In *Hot Chips*, volume 16, 2004.

[8] S. Benderli and TA Wey. On SPICE macromodelling of TiO2 memristors. *Electronics letters*, 45(7):377–379, 2009.

[9] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt. The M5 simulator: Modeling networked systems. *Micro, IEEE*, 26(4):52–60, 2006.

[10] Z. Biolek, D. Biolek, and V. Biolvoká. SPICE model of memristor with nonlinear dopant drift. *Radioengineering*, 18(2):210–214, 2009.

[11] R. Bruchhaus and R. Waser. Bipolar Resistive Switching in Oxides for Memory Applications. *Thin Film Metal-Oxides*, pages 131–167.

[12] D. Burger and T.M. Austin. The SimpleScalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

[13] G.W. Burr, B.N. Kurdi, J.C. Scott, C.H. Lam, K. Gopalakrishnan, and R.S. Shenoy. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development*, 52(4):449–464, 2008.

[14] A.M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R.K. Gupta, A. Snavely, and S. Swanson. Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[15] L. Chua. Memristor-the missing circuit element. *IEEE Transactions on Circuit Theory*, 18(5):507–519, 1971.

[16] J. Coburn, A.M. Caulfield, A. Akel, L.M. Grupp, R.K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 105–118. ACM, 2011.

[17] J. Condit, E.B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146. ACM, 2009.

[18] A. Cooper, R. Reimann, and H. Dubberly. *About face 2.0: the essentials of interaction design*. John Wiley & Sons, Inc., 2003.

[19] A. Dearle and D. Hulse. Operating system support for persistent systems: past, present and future. *Software-Practice and Experience*, 30(4):295–324, 2000.

[20] A. Dearle, D. Hulse, and A. Farkas. Operating system support for java. In *In Proceedings of the First International Workshop on Persistence and Java, Drymen, Scotland*, 1996.

[21] A. Dearle, G. Kirby, and R. Morrison. Orthogonal persistence revisited. *Object Databases*, pages 1–22, 2010.

[22] G. Dhiman, R. Ayoub, and T. Rosing. PDRAM: A hybrid PRAM and DRAM main memory system. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, pages 664–669. IEEE, 2009.

[23] X. Dong, N.P. Jouppi, and Y. Xie. PCRAMsim: system-level performance, energy, and area modeling for phase-change ram. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 269–275. ACM, 2009.

[24] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen. Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 554–559. IEEE, 2008.

[25] U. Drepper. What every programmer should know about memory. *Captured in: http://people.redhat.com/drepper/cpumemory.pdf*, December 2007.

[26] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In *Proceedings of the 3rd international workshop on Data management on new hardware*, pages 1–9. ACM, 2007.

[27] W. Guo, G. Prenat, V. Javerliac, M.E. Baraji, N. Mestier, C. Baraduc, and B. Diény. SPICE modelling of magnetic tunnel junctions written by spin-transfer torque. *Journal of Physics D: Applied Physics*, 43:215001, 2010.

[28] JD Harms, F. Ebrahimi, X. Yao, and J.P. Wang. SPICE Macromodel of Spin-Torque-Transfer-Operated Magnetic Tunnel Junctions. *Electron Devices, IEEE Transactions on*, 57(6):1425–1430, 2010.

[29] Y. Ho, G.M. Huang, and P. Li. Nonvolatile memristor memory: device characteristics and design implications. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 485–490. ACM, 2009.

[30] E. Ipek, J. Condit, E.B. Nightingale, D. Burger, and T. Moscibroda. Dynamically replicated memory: building reliable systems from nanoscale resistive memories. *ACM SIGARCH Computer Architecture News*, 38(1):3–14, 2010.

[31] B. Jacob, S.W. Ng, and D.T. Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann Pub, 2007.

[32] M. Jordan and M. Atkinson. Orthogonal persistence for the java platform — specification. 2000.

[33] C.K. Koh, W.F. Wong, Y. Chen, and H. Li. The Salvage Cache: A fault-tolerant cache architecture for next-generation memory technologies. In *Computer Design, 2009. ICCD 2009. IEEE International Conference on*, pages 268–274. IEEE, 2009.

[34] M.H. Kryder and C.S. Kim. After Hard Drives - What Comes Next? *Magnetics, IEEE Transactions on*, 45(10):3406–3413, 2009.

[35] A.M. Law, W.D. Kelton, and W.D. Kelton. *Simulation modeling and analysis*. McGraw-Hill New York, 1991.

[36] B.C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. *ACM SIGARCH Computer Architecture News*, 37(3):2–13, 2009.

[37] B.C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. Phase-Change Technology and the Future of Main Memory. *IEEE Micro*, 30(1):143, 2010.

[38] D.L. Lewis and H.H.S. Lee. Architectural evaluation of 3D stacked RRAM caches. In *IEEE International Conference on 3D System Integration, 2009. 3DIC 2009.*, pages 1–4. IEEE, 2009.

[39] H. Li and Y. Chen. An overview of non-volatile memory technology and the implication for tools and architectures. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.*, pages 731–736. IEEE, 2009.

[40] S. Li, J. Ahn, J.B. Brockman, and N.P. Jouppi. McPAT 1.0: An Integrated Power, Area, and Timing Modeling Framework for Multicore Architecture. *HP Labs*, 2009.

[41] S. Li, J.H. Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480. IEEE, 2010.

[42] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Haallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, pages 50–58, 2002.

[43] C.D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[44] A. Marquez, J.N. Zigman, and S.M. Blackburn. Fast portable orthogonally persistent java. *Software - Practice and Experience*, 30(4):449–479, 2000.

[45] M.M.K. Martin, D.J. Sorin, B.M. Beckmann, M.R. Marty, M. Xu, A.R. Alameldeen, K.E. Moore, M.D. Hill, and D.A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[46] 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*, pages 14–14. USENIX Association, 2009.

[47] G.E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[48] R. Morrison, RCH Connor, GNC Kirby, DS Munro, M.P. Atkinson, QI Cutts, AL Brown, and A. Dearle. The napier88 persistent programming language and environment. *Fully Integrated Data Environments*, pages 98–154, 1999.

[49] N. Muralimanohar, R. Balasubramonian, and N.P. Jouppi. CACTI 6.0: A Tool to Model Large Caches. *School of Computing, University of Utah, Tech. Rep*, 2007.

[50] P.M. Ortego and P. Sack. SESC: SuperESCalar simulator. In *17 th Euro micro conference on real time systems (ECRTS 05)*, pages 1–4. Citeseer, 2004.

[51] T. Perez, N.L.V. Calazans, and C.A.F. De Rose. A preliminary study on system-level impact of persistent main memory. In *Quality Electronic Design (ISQED), 2012 13rd International Symposium on*, pages 85–90. IEEE, 2012.

[52] T. Perez and C.A.F. De Rose. Non-Volatile Memory: Emerging Technologies And Their Impacts on Memory Systems (TR-060). Technical report, Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), 2010.

[53] 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*, pages 24–33. ACM, 2009.

[54] J. Åkerman. Toward a universal memory. *Science*, 308(5721):508, 2005.

[55] P. Ranganathan. From Microprocessors to Nanostores: Rethinking Data-Centric Systems. *IEEE Computer*, 44(1):39–48, 2011.

[56] S. Raoux, GW Burr, MJ Breitwisch, CT Rettner, Y.C. Chen, RM Shelby, M. Salinga, D. Krebs, S.H. Chen, H.L. Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2010.

[57] G. Reinman and N.P. Jouppi. CACTI 2.0: An integrated cache timing and power model. *COMPAQ Western Research Lab*, 1999.

[58] D. Roberts, J. Chang, P. Ranganathan, and T.N. Mudge. Is Storage Hierarchy Dead? Co-located Compute-Storage NVRAM-based Architectures for Data-Centric Workloads. Technical report, HP Labs, 2010.

[59] D.A. Roberts. *Efficient Data Center Architectures Using Non-Volatile Memory and Reliability Techniques*. PhD thesis, The University of Michigan, 2011.

[60] P. Shivakumar and N.P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical report, Citeseer, 2001.

[61] X.G.E.I.T. Soyata. Resistive Computation: Avoiding the Power Wall with Low-Leakage, STT-MRAM Based Computing. 2010.

[62] D.B. Strukov, G.S. Snider, D.R. Stewart, and R.S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, 2008.

[63] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen. A novel architecture of the 3D stacked MRAM L2 cache for CMPs. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 239–249. IEEE, 2009.

[64] D. Tarjan, S. Thoziyoor, and N.P. Jouppi. CACTI 4.0. *HP Laboratories, Technical Report*, 2006.

[65] S. Thoziyoor, N. Muralimanohar, J.H. Ahn, and N.P. Jouppi. CACTI 5.1. *HP Laboratories, April*, 2008.

[66] S. Venkataraman, N. Tolia, P. Ranganathan, and R.H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11), San Jose, CA*, pages 61–76, 2011.

[67] Andre Vladimirescu. *The Spice Book*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[68] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10), Poster, October 2010.

[69] H. Volos, A.J. Tack, and M.M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 91–104. ACM, 2011.

[70] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. DRAMsim: a memory system simulator. *ACM SIGARCH Computer Architecture News*, 33(4):100–107, 2005.

[71] F.Y. Wang. Memristor for introductory physics. *Arxiv preprint arXiv:0808.0286*, 2008.

[72] R.S. Williams. How we found the missing memristor. *IEEE spectrum*, 45(12):28–35, 2008.

[73] S.J.E. Wilton and N.P. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, 1996.

[74] M. Wu and W. Zwaenepoel. envy: a non-volatile, main memory storage system. In *ACM SigPlan Notices*, volume 29, pages 86–97. ACM, 1994.

[75] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie. Hybrid cache architecture with disparate memory technologies. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 34–45. ACM, 2009.

[76] X. Wu, J. Li, L. Zhang, E. Speight, and Y. Xie. Power and performance of read-write aware hybrid caches with non-volatile memories. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 737–742, 2009.

[77] 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, 2008.

[78] Y. Zhang, X. Zhang, and J. Yu. Approximated SPICE model for memristor. In *Communications, Circuits and Systems, 2009. ICCCAS 2009. International Conference on*, pages 928–931. IEEE, 2009.

[79] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3):14–23, 2009.