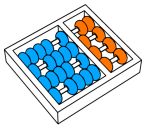
Universidade Estadual de Campinas
Instituto de Computação



Maurício Gagliardi Palma

Simulation-driven Exploration of NVM as a Substitute for DRAM in the Main Memory

Explorando a Substituição de DRAM por NVM na Memória Principal através de Simulação

CAMPINAS
2017

Maurício Gagliardi Palma

Simulation-driven Exploration of NVM as a Substitute for DRAM in the Main Memory

Explorando a Substituição de DRAM por NVM na Memória Principal através de Simulação

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Rodolfo Jardim de Azevedo
Co-supervisor/Coorientador: Dr. Emílio de Camargo Francesquini

Este exemplar corresponde à versão final da Dissertação defendida por Maurício Gagliardi Palma e orientada pelo Prof. Dr. Rodolfo Jardim de Azevedo.

CAMPINAS
2017

Agência(s) de fomento e nº(s) de processo(s): CAPES, 1564396

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

P18s

Palma, Maurício Gagliardi, 1991-
Simulation-driven exploration of NVM as a substitute for DRAM in the main memory / Maurício Gagliardi Palma. – Campinas, SP : [s.n.], 2017.

Orientador: Rodolfo Jardim de Azevedo.
Coorientador: Emílio de Camargo Francesquini.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Sistemas de memória de computadores. 2. Memória de acesso aleatório não-volátil. 3. Arquitetura de computador. I. Azevedo, Rodolfo Jardim de, 1974-. II. Francesquini, Emílio de Camargo. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Explorando a substituição de DRAM por NVM na memória principal através de simulação
Palavras-chave em inglês:
Computer memory systems
Nonvolatile random access memory
Computer architecture
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Rodolfo Jardim de Azevedo [Orientador]
Alexandro José Baldassin
Luiz Eduardo Buzato
Data de defesa: 18-09-2017
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Maurício Gagliardi Palma

Simulation-driven Exploration of NVM as a Substitute for DRAM in the Main Memory

Explorando a Substituição de DRAM por NVM na Memória Principal através de Simulação

Banca Examinadora:

- Prof. Dr. Rodolfo Jardim de Azevedo
  IC/UNICAMP
- Prof. Dr. Alexandro José Baldassin
  IGCE/UNESP
- Prof. Dr. Luiz Eduardo Buzato
  IC/UNICAMP

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 18 de setembro de 2017

Resumo

O sistema de memória dos computadores tem se baseado fortemente no uso de memórias voláteis para prover um bom desempenho. A tecnologia SRAM é utilizada como um intermediário que acelera o acesso à memória principal, comumente composta pela tecnologia DRAM. Memórias não-voláteis são colocadas como memórias secundárias. Pelo fato dos dados persistentes estarem armazenados no nível de memória mais distante do processador, eles normalmente são manipulados de maneira indireta através de cópias transientes. Tais cópias transientes, além de possívelmente estarem presentes em mais de um nível de memória volátil, podem não ter a mesma forma de suas formas persistentes, o que leva à necessidade de uma tradução entre essas formas. Tecnologias emergentes de memórias não-voláteis (NVMs) prometem possibilitar a existência de dados persistentes na memória principal, permitindo que os mesmos sejam manipulados diretamente, e potencialmente reduzindo a quantidade de cópias transientes. Infelizmente, NVMs ainda não estão amplamente disponíveis no mercado, e pesquisas em seu uso são normalmente feitas através de simulação. Neste documento é apresentado um simulador que tem como fim explorar o uso de NVMs na memória principal. Por enquanto, a tecnologia DRAM provê um tempo de acesso inferior ao das NVMs, restringindo o uso de NVMs na memória principal em questão de desempenho. São mostrados aqui dois cenários para o uso do simulador. No primeiro caso, há a utilização de uma memória principal composta apenas de NVM. Como NVM é mais lenta, são observados certos *slowdowns* de até 5,3, mas em alguns programas o desempenho é marginalmente afetado. Em um segundo caso, há a exploração da memória híbrida, onde DRAM e NVM coexistem na memória principal. Uma API, chamada NVMalloc, é fornecida para permitir que programas consigam utilizar a não volatilidade presente na memória principal. É mostrado que há casos onde a manipulação direta dos dados persistentes é vantajosa, mas existem outros em que ainda é preferível trabalhar com cópias transientes na DRAM. É esperado que esse simulador seja utilizado como um ponto de partida para futuras pesquisas sobre o uso de NVMs.

Abstract

Computer memory systems have relied on volatile memories to enhance their performance for quite a time by now. SRAM technology is used at the closest layer to the CPU to accelerate the access time to the main memory, which is traditionally composed by DRAM technology. Non-volatile memories are left as secondary memories, serving as an extension of the main memory and allowing data to be persisted. Persistent data, for residing in the farthest memory layer from the CPU, are commonly not manipulated directly. They are indirectly manipulated with their transient copies that may differ, in form, from their persistent form. These transient copies will also be scattered throughout the several volatile memories in the memory hierarchy, incurring in data replication. This scenario may change with the adoption of emerging non-volatile memories (NVMs), like phase change memory for example, that may allow persistent data to exist in the main memory. This might allow a direct manipulation of persistent data, accelerating their access time and probably reducing the usage of replications. Unfortunately, NVMs are still not broadly available on the market, and research on their usage is still mostly done through simulation. We present a simulator to explore the usage of NVMs in the main memory. We demonstrate the usage of the simulator in two scenarios, the first where DRAM is completely replaced for NVMs, and the second in which a hybrid architecture employing DRAM and NVM is explored. For now, DRAM provides faster access times when compared with NVMs. We show that the use of a main memory composed exclusively of NVMs may incur in slowdowns as high as 5.3, but may be negligible in some cases. In the hybrid main memory scenario, we showed that, although persistent data can be manipulated directly, there are cases in which is still better to work with transient copies, depending on the frequency of usage of the persistent data. To allow programs to make use of the non-volatility presented in main memory, we provide an API, called NVMalloc, that is able to allocate persistent memory in the main memory. We expect the simulator to be a starting point for future researches regarding the usage of NVMs.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

The way data is manipulated by the software is highly influenced by how the hardware is built and organized. Transient data, that do not need to outlive a program's execution, can be stored in volatile memories, while persistent data cannot. Current memory systems are composed by several memory technologies, each serving for a purpose. These memories are in a hierarchical organization, letting smaller but faster memories closer to the processor and slower but denser memories farther [43]. These smaller and faster memories are volatile, usually built using the Static Random Access Memory (SRAM) technology for caching and Dynamic Random Access Memory (DRAM) technology as main memory, and so responsible for dealing with transient data. The slower and denser technologies are non-volatile memories, usually with Hard Disk Drives (HDDs) and Solid State Drives (SSDs), responsible for providing a high storage capacity and a place to store persistent data. The volatile memories role in this context is usually known as working memory, and the non-volatile memories role is known as storage memory.

The urge for better qualities in entertainment media [24], the Internet of things (IoT) [22] and the growth in big data usage [97] may indicate that the amount of data will grow in the coming years, to an amount that current memory system may not be suitable to handle [11,32,88,93]. The current memory system, organized as it is, relies too much on volatile memory to lower its latency, that consumes energy just to hold the data. As there are several levels of volatile memory, such as main memory and cache, there is the replication of data in these, resulting in more than one memory holding the same data. Besides that, persistent data must be manipulated using transient copies of itself. The data replication through the hierarchy might result in an excess of energy consumption and a suboptimal memory capacity utilization, and the incapacity of manipulating persistent data directly may lead to performance loss.

A solution to adapt the memory system to the future amount of data is to rely less on volatile memories, trying to get persistent data closer to the processor and eliminate data replication through the memory hierarchy. Emerging non-volatile memory technologies, being collectively known as Non-Volatile Memories (NVMs) [23, 38, 54, 100], may allow persistent data to exist closer to the processor. They are byte-addressable, have access timings similar to DRAM and SRAM, and might be as dense as current storage memories. Actually, these new memory technologies are good candidates to fulfill the role of both storage and working memory [26, 67, 89, 94]. This opens the possibility to bring both

working and storage memory to a same level in the memory hierarchy. Where NVMs will be used in the memory hierarchy and how software will take advantage of them are questions that arise when one considers this solution to adapt the memory system.

Unfortunately, NVMs are still not widely accessible, making it a hard task to explore their usage. For this reason, researchers usually rely in the use of simulators. Simulation of a machine through software is something that has been done for at least 60 years [36] and it is a well-known technique to optimize the exploration of computer's architecture [96]. In order to facilitate the exploration of NVMs usage in the memory system, we propose a new customized simulator. The simulator is able to simulate a computer with a main memory composed partially or totally of NVM. In Chapter 2 we present a deeper discussion about the reasons that made the current memory systems to take their current form, bringing to the context where this work was made. In Chapter 3, we describe the simulator, how it was made and its characteristics. We also describe an Application Programming Interface (API) to be used together with the simulator, called NVMalloc, to explore the possibility of usage of a hybrid main memory by the software.

In Chapter 4, we demonstrate the usage of the simulator for both a partial and a total NVM adoption as main memory. Despite of the promises from the NVMs, they are still slower than the DRAM technology. The results show that utilizing a main memory based only on NVMs may result in slowdowns as high as 5.3, but it is also, in some cases, negligible. Considering a hybrid approach, where the main memory is made by both DRAM and NVM, we show how advantageous it can be to persist data in the main memory, and discuss when still is worth to transfer this data to DRAM.

Finally, in Chapter 5, we conclude and discuss possible enhancements to the proposed simulator. Results obtained during the development of the simulator were already published in two occasions. The first as a short paper [70] and the second as a full paper [69].

Chapter 2

Basic concepts and related work

It is desirable that a computer computes as fast as possible. Considering that computers execute instructions, the time to do computation is the sum of the time to obtain the instruction's input and the time from the actual execution of this instruction. In modern computers, the part responsible to provide the data as input to instructions is called memory system, and the part responsible to actually execute instructions is called central processing unit (CPU). An important detail here is that, besides providing the input to instructions, the memory system also provides the instruction itself. So, before the CPU requires an instruction's input, it must require the instruction. This concludes that instructions are also data, and that without the provisioning of the memory system, a CPU is useless. Hence, the performance of execution from the CPU and the data provisioning from the memory system define the overall performance from a computer system. Therefore, in seek of a better performance, the relationship between these two parts has driven the way computers work and how they are designed.

Since the memory system is the responsible to provide data to the CPU, it is also responsible to obtain these data from wherever they are. It ends up meaning that the memory system is also responsible to obtain data from outside the computer, making an interface that allows data to be inserted in the computer system. It is also desirable to have a way to access the data that is inside the computer from outside of it, making necessary an interface to output the data. These interfaces of input and output are usually mentioned as a separated part from the memory system, being composed by input/output (I/O) devices and not memories, and being refered just as I/O. It makes sense considering that it is not always that the memory system will rely on input interfaces to obtain data (data can be generated from executing instructions), but since they may need to access them to obtain data to provide to the CPU, they turn to be a part of the memory system.

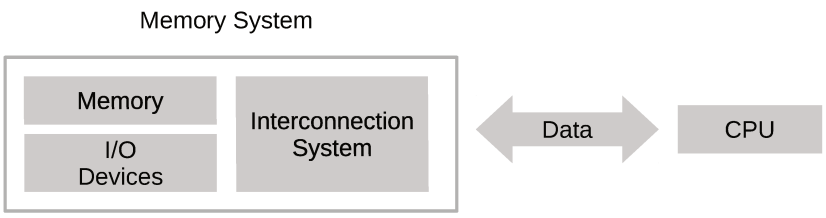


Figure 2.1: Computer System Overview.

Figure 2.1 ilustrates an overview of the computer system and its two sides, CPU and memory system, both connected by an interconnection system, with data flowing in both ways. The amount of data required by the CPU will be dictated by the software that it is running. The same is valid to the amount of data that will go back to the memory system. Taking this into account, the enchancement of this relationship is possible by both software, adapting it to take the best performance, and hardware, designing the memory system according to the software.

Despite having passed more than 60 years, concepts that appeared in the draft of the Electronic Discrete Variable Computer (EDVAC) [66] are still present in the current computer architecture. Hence, we will use this computer, that is simple for current standards, as a starting point to explore the concepts that surrounds the computer architecture and the memory hierarchy.

2.1 Memory system from a hardware perspective

One of the first modern computers was the Electronic Discrete Variable Computer (EDVAC) [37]. It was built for military purposes and to be one of the most performant computers of its time. Its memory was a mercury delay line memory [30]. This kind of memory utilized mechanical waves propagated in a material, mercury in this particular case, to store the data. Since a mechanical wave tends to lose energy as it travels through a material, it needs to be reestimulated from time to time in order to keep the wave, and in this way, to keep the data. If the reestimulation did not happen, the wave would be lost, and so would the data. The medium that data are kept in a memory was named memory cell, and it usually defines a memory technology. In the example of the delay line memory used in EDVAC, the mercury was the memory cell.

The characteristic that memory cells need to be reestimulated in order to keep the data leads to an inevitable characteristic for these kind of memories, that they are volatile. A volatile memory is a memory that can not hold data without a constant supply of energy. A volatile memory may not be the most reliable of the memories, since a power loss would mean that all data in it will be lost. This is the reason why EDVAC also had a magnetic drum memory [29, 50] as a non-volatile memory. The drum memory was used to store important data that needed to be kept even after the computer was turned off. It is said that, when data need to be stored in a non-volatile memory, they need to be persisted, and these data are labeled as persistent data. In contrast to persistent data, those that are not needed to be persisted are labeled as transient data.

One may ask why the EDVAC did not have only the drum memory, since it is non-volatile and, in principle, could store data just like the mercury delay line memory. Performance was the answer. The mercury delay line memory had an average of 1000 times lower access time than the magnetic drum memory, being faster to provide data. The downsides of the mercury delay line memory were that it was volatile and had a little storage capacity when compared to the magnetic drum memory. In addition to the drum memory, EDVAC had also punched and magnetic tapes [50, 82]. All of these additional storage memories were non-volatile, however they also had a higher access time.

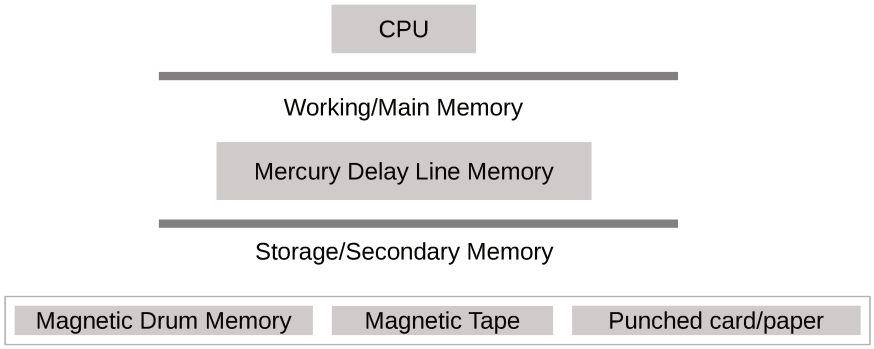


Figure 2.2: EDVAC - Memory Hierarchy.

The EDVAC had several kinds of memories. One for being fast (mercury delay line memory with low access time), and several others that were non-volatile with a high storage capacity but slow (magnetic drum memory, punched card/papers, magnetic tapes). Thus, each memory had its purpose. The mercury delay line memory was to treat transient data, providing a faster memory to the CPU. A memory that fulfills this purpose is called a working memory. The purpose for the magnetic drum memory and the other non-volatile memories presented in EDVAC was to treat persistent data and to provide a high storage capacity. A memory that fullfills this purpose is called a storage memory[1]. Working memory and Storage memory are usually also called, respectivelly, primary or main memory and secondary memory.

Figure 2.2 ilustrates the memories from the EDVAC. Since the CPU shall process the data that are in the working memory, and the working memory might need to fetch the data from the secondary memories, a hierarchy is formed. The hierarchical relation between the memories is known as memory hierarchy [72]. The purpose of the memory hierarchy is to give ideal characteristics to the memory system, that is, having the access times of the working memory and the storage capacity of the storage memory.

A sequence of instructions, or a program, should be written to computers like EDVAC in order to guide them to do worthy computation, that would solve a problem. Programs defines the way that CPU will interact with the memory system. Two characteristics have been detected in several kinds of programs about how they require data: if a datum in position $\mathbf{x}$ were required, it is most likely that the next datum to be requested will be a datum in a position close to $\mathbf{x}$. Also, datum in position $\mathbf{x}$ tends to be required in a near future. These characteristics are respectivally known as spatial locality and temporal locality [72].

The spatial and temporal locality led computer designers to evaluate another characteristic from memory: the correlation from capacity in its access time. They found out that less capacity would mean less access time. This fact culminated in the adoption of memories with little storage capacity between the working memory and the CPU, creating a new level to the memory hierarchy. Memories in this level, initially known as slave memories [87], came to be known as cache memories [43]. Cache memories are memories that are made with the same technology as main memory or with a one that has lower

[1]Note that a volatile memory can not be storage memory, but a non-volatile memory can be working memory.

access time, and smaller storage capacity. Cache memories purpose is to hold recent data that was requested by the CPU in chunks known as cache lines, and keep them as much as the CPU needs them, reducing the number of accesses to the working memory. Since it is faster to access cache than the main memory, the more a memory system can rely on cache, the faster it will be. Caches work best when the program displays good spatial and/or temporal locality behaviors.

2.2 Memory system from a software perspective

Cache memories are invisible to programs. That means that programmers do not need to worry about managing them[2]. On the other hand, the working memory and storage memory are handled by programs. As mentioned, working memory is for temporal data, and storage memory is for persistent data. A program must know if data is persistent in order to require its storage to the storage memory. An issue that a programer may encounter is when the amount of transient data from a program surpasses the main memory storage capacity. When it happens, the storage memory can be used as an extension from the working memory to store transient data, making the working memory to behave like a cache memory to the storage memory. Initially, in computers like EDVAC, this extension of working memory to the storage memory were left as a responsibility of the programs. Later, a concept called virtual memory [25, 33, 51, 83] was implemented in computer systems that allows this management to be made in an automatic way, decoupling this responsability from the program.

Indeed, management of computer resources is far from trivial. Operating Systems (OS) [83] are employed in computational systems to do it. An OS provides an abstraction of the hardware to programs, so programs do not need to worry about every aspect of the hardware. An example is virtual memory. Paging [25, 83] is a method of implementing it. In Paging, the memory is divided in chunks called pages. If a memory access does not have its page loaded in main memory, a page fault will occur, and the page will be allocated in main memory. When the main memory is full of pages and a new one is needed, a page from the main memory will be transfered to storage memory, and a new page will be allocated in its place. This page that was moved to storage memory may be accessed again, and if so, it will be moved back to the main memory in the place of another page that will be moved to storage memory. This operation of swapping pages between main memory and storage memory is called swap. This mechanism is completely handled by the OS, and oftentimes programs do not need to be aware of it.

Besides providing an abstraction of the hardware to programs, the OS also comes to enhance the usage of computational resources. Since the memory system might not always give the fastest access times, the CPU might become idle when waiting for data. Trying to maximize the CPU usage, OSs implement a concept known as multiprograming [83], that allows the scheduling, at the same time, of two or more programs to be executed. The programs are organized in a queue, usually known as execution queue, and if a program

[2]Even if programmers do not need to manage the cache memory, they should be aware of it, since keeping spatial locality and temporal locality in their programs should improve their performance.

issue an instruction that may result in a long access time to the memory system, it stops and goes to the end of the execution queue, waiting for its data to arrive, yielding the CPU to another program. An access to the storage memory usually is considered a long access time, resulting in a program to go to the end of the execution queue.

The storage memory, as already stated, can store both transient and persistent data. Besides being responsible for automatically manage the exchange of transient data between working and storage memory, the OS also provides the access to persistent data in storage memory. This access occurs through system calls (syscalls) to read and write data in storage memory. The persistent data is stored in an abstract structure called file, and the whole storage memory, excluding the swap partition, is organized by a file system [83]. Every file, when accessed, needs to be loaded to the main memory. In this way, it reinforces the terminology of storage memory as secondary memory, being treated as an I/O device.

Usually, transient data, when need to be persisted, can not be persisted in the same format that they are in the transient form. For example, the several nodes from a linked list might be sparse, and if so, to persist them, they will need to be arranged together in memory in order to be inserted into a file. If this file is ever accessed to recover the linked list, the program might need to translate it back to its transient form. This need of translation between transient and persistent form is known to be a burden to the programmers. It is so that several programming languages have proposed special interfaces to aid the programmers in dealing with persistent data [4, 5, 79]. PS-algol [4] for example extends the S-algol language to use instructions in a key-value manner that automatically translates transients to persistent data and vice-versa. There is also the possibility to map an entire file to the program's address space, acessing it directly like transient data, something known as memory mapped file [84], removing several syscalls that would be needed to modify them. Actually, memory mapped files are implemented using syscalls, so the OS is the one that makes it possible and is the one responsible for managing it. If a file is memory mapped, it will be divided in pages and loaded to working memory as soon as the pages are being accessed.

2.3 Current memory technologies and future memory systems

Besides all the enhancements that were made in the memory system, the division of its memories has not changed that much since the EDVAC [64]. Figure 2.3 shows the current state of the memory system [43]. Dynamic Random Access Memory (DRAM) as the main memory, backed by Hard Disk Drives (HDDs) and Solid State Drives (SSDs)[3], and being accelerated by several levels of cache made with Static Random Access Memory (SRAM).

Several levels of caches were implemented to exploit as much as possible the characteristic of being smaller means being faster. The closest cache level to the CPU, called

[3]Although the drum memories and punched cards/tapes are now history, magnetic tapes are still employed as storage memories [20], but they are not in the majority of the computer systems and are only used in specific cases.

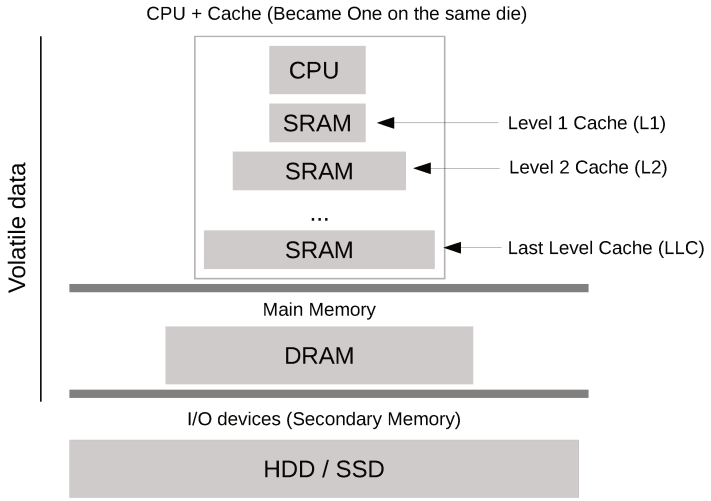


Figure 2.3: Current computer - Memory Hierarchy.

L1 cache, is usually divided in two [81], being one part solely for storing instructions, and the other one for the data. To reduce even more the caches latencies, they are built in the same die with the CPU [6]. The last level cache usually is L3, but there may be more. For example, a few CPUs from Intel's Haswell architecture have a L4 cache[4] [39].

Caches are built with SRAM technology, a volatile, expensive and energy hungry technology. For a greater storage capacity in main memory, DRAM, a cheaper and also denser technology is employed. Just like SRAM, DRAM is also volatile, and both are suitable to fit the role of working memory. For persistent data there are HDDs and SSDs, both being non-volatile and denser than SRAM and DRAM. SRAM and DRAM are capable of addressing all the bytes that they store, a characteristic known as byte-addressability. HDDs and most SSDs are unable to do it[5], dividing its addresses between chunks of data called blocks, that are greater in size than a byte, characteristic known as block-addressability. In constrast to DRAM, which is connected to the CPU through a memory bus dedicated for data exchange between memory and CPU, memories that work with blocks (known as block devices) are connected as I/O devices [43].

The memory system in its current state has some undesirable characteristics. There are data replicated throughout the hierarquical levels with the sole purpose of lowering the access time, resulting in spending more memory space and energy to store the same data. There is also the struggle about dealing with persistent data and their translations to their transient form. With such characteristics, the memory system becomes a rich field of research. As the amount of data tends to increase [42, 85, 97], so does the pressure in the memory system to able to deal with these new amounts of data. There are those who claim that the current memory hierarchy is unsuitable to deal with the future amount of data [32, 88, 93], and so a change will be needed.

There are a few emerging memory technologies that are candidates to fit any level in the memory hierarchy, that is, they may have low access time with large storare capacity, and they are non-volatile. Collectively these memory technologies are known as

[4]It is worth mentioning that DRAM is also used as cache memory. This is the case of the L4 cache from Intel's Haswell Architecture, being referred as embedded DRAM (eDRAM).

[5]Actually there are SSDs, made with NOR Flash technology, that are byte-addressable.

Non-Volatile Memory (NVM)[6] [38, 54, 100]. Magnetoresistive Random Access Memory (MRAM), like Toogle MRAM [28] and Spin-Transfer Torque RAM (STTRAM) [41], and Phase-Change Memory [90] are examples of NVMs. These non-volatile memories, as shown in Figure 2.4, have the potential to rivalize the storage capacity from HDDs and SSDs, and the access time of DRAM and SRAM, being suitable to fullfill any role in the memory hierarchy.

As storage memory, indicated in Figure 2.4b, NVMs would provide a faster storage memory than current tecnologies. NVMs, in contrast to HDDs and SSDs, are byte-addressable, being able to connected to the memory bus just like the main memory, removing the overhead of being treated as an I/O device. Several works already considered this possibility [14,19,27,92], proposing file systems that would take advantage from this. In most of them, they try to exploit the possibility of execute-in-place (XIP), that allows to access data directly from the storage memory, eliminating the need of a transient copy in the working memory[7].

As working memory, NVMs may share the role with the DRAM technology, as indicated in Figure 2.4c, allowing data to be persisted in its transient format. As NVMs still struggle in beating DRAM's access time and endurance, this seems to be the most feasible way to adopt NVMs for working memory. Just like storage memory, several works considered this possibility [13, 17, 86, 99]. They propose a new Aplication Programing Interface (API) to manage this new situation where there is non-volatile memory in the main memory. The use of NVMs as working memory tends to have a better performance than utilizing other memory technologies for this end, like in eNVy [91], that proposes the utilization of flash technology in the main memory.

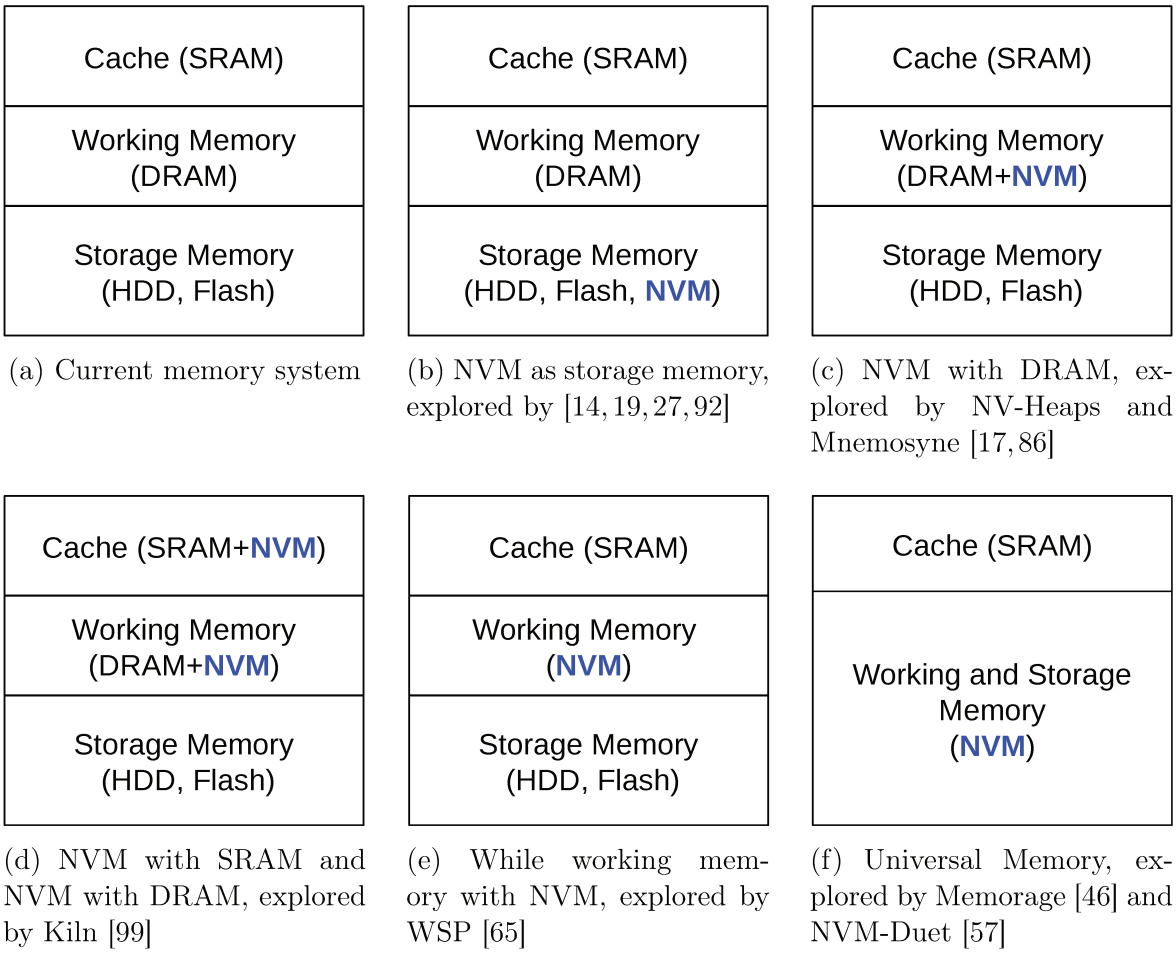
Even if NVMs still do not surpass DRAM's performance, some researchers have considered them as cache memories [95, 99], as indicated by Figure 2.4d. SRAMs are too expensive and, since they are volatile, they spend a considerable amount of energy just for holding the data. These are adversaties that do not exist in NVMs. Actually, like already said, volatile memories may not be the most reliable ones, since they lose their data when energy is removed. Whole-System Persistence [65] for example considered a whole main memory made with NVMs, as indicated in Figure 2.4e.

If NVMs may fulfill the roles of working and storage memory, they might be able to fullfill them both at the same time, becoming an Universal Memory [26,67,89,94] (that is, a memory that plays both roles). If it happens, the hierarchy levels that refers to working and storage could be collapsed in a single level, like indicated by Figure 2.4f. A few works considered this possibility [46, 47, 57, 62, 68, 73]. NVM DUET [57] for example evaluated memory bank paralelism to try to hide NVM latencies when treating both transient and persistent data in the same memory. Memorage [46] investigated the possibility of controlling, by software, portions of the same memory to fulfill both working and storage roles. Reducing the levels of the memory hierarchy can lead to less useless data replication

[6]Despite slight nuances in their meaning, NVM and all the following terms commonly found in the literature can be used interchangeably: *Storage-Class Memory* (SCM), *Byte-Addressable Persistent Random Access Memory* (BPRAM), *Nonvolatile Random Access Memory* (NVRAM), *Persistent Random Accecss Memory* (PRAM) and *Persistent Memory* (PM).

[7]It is worth to point out that XIP is already a reality with NOR Flash technology [8], so it is not an exclusivity from NVMs.

and could get all data closer to the CPU, which is something that seems desirable for the future's workloads [32].



(a) Current memory system

(b) NVM as storage memory, explored by [14, 19, 27, 92]

(c) NVM with DRAM, explored by NV-Heaps and Mnemosyne [17, 86]

(d) NVM with SRAM and NVM with DRAM, explored by Kiln [99]

(e) While working memory with NVM, explored by WSP [65]

(f) Universal Memory, explored by Memorage [46] and NVM-Duet [57]

Figure 2.4: Memory Hierarchy Design Possibilities

2.4 Computer Architecture Exploration - Simulators

A common way to optimize, or even allow, the exploration of new approaches in computer architecture or the evaluation of the current one is through simulation [96].

There are more than one way to categorize a computer architecture simulator. One of them considers if a simulator targets only a functional simulation or if it also targets some quantitative results. Functional simulation is to simulate only the behaviour of a software in a target Instruction Set Architecture (ISA). It may also simulate system calls from diferent OSs. An example of a simulator that targets only functional simulation is QEMU [7]. Simulators like QEMU are very useful to run legacy applications targeted for machines that do not exist anymore. For new architecture exploration, it is needed that the simulator outputs some quantitative information, usually some timing parameters.

For simulators that focus on quantitative results, they rely in pondering three factors: abstraction level, accuracy and flexibility. Abstraction level is how far the simulator model is from the real machine architecture. Accuracy is how realistic are the results when

compared to the execution on a real machine, and flexibility is how easy is to introduce changes in your model. Usually, to achieve higher precision, the simulator needs to lower the abstraction level, increasing the time that is need to execute the simulation.

It is also important to know if the simulator is able or not to simulate an OS. If a simulator is capable of it, it is called a Full-System Simulator. If not, it is said that the simulator only simulates the user space, and is called user space simulator or partial simulator. Examples of Full-System simulators are QEMU, MARSSx86 [71], Simics [60] and Gem5 [9]. To consider the execution of a OS in the simulation may inccur in a high overhead in simulation. For that reason, some simulators, like Simics and Gem5, also provide partial simulation as well.

The trend for manycore architectures has driven the creation of software that are multithreading, and the same occurs with simulators. MCSimA+ [2], Sniper [12] and ZSIM [78] are simulators that were built targeting the simulation of manycore architectures. They promise to have a good balance between abstraction level and precision, to have a low simulation time, being partial simulators but providing a thin layer of OS functions, that allow process schedule to processors and even time sharing. These simulators utilize dynamic binary translation, provided by the PIN [59] instrumentation tool, to input code inside a binary that is running in the simulator. This inputted code updates its timing model, and leave the functional simulation to the host machine. Delegating the functional simulation to the host machines diminishes the simulation overhead, but restrains its usage for the fact that the target simulated ISA must be the same as the host machine.

2.5 Problem statement, proposal and contribution

As indicated in Figure 2.4 there are several possibilities to introduce NVMs in the memory system. Unfortunately, despite the many prototypes that were made [15, 31, 44, 55, 58, 98], NVMs are not yet widely accessible, especially in the Dual Inline Memory Module (DIMM) format [35] that is used by the DRAM. If one wants to use NVMs as main memory, they can rely on Non-Volatile DIMM (NVDIMM) [75], that are DRAMs modules backed by Flash memory, storing data to Flash when turned off, and restoring data from Flash to DRAM on turning on. NVDIMMs uses batteries or the energy from an energized capacitor to save data in Flash memory in case of a power failure. Intel has announced a NVM technology in 2015[8], the 3D XPoint, and has recently released a product that uses it. The Intel Optane [10] is built using 3D XPoint technology, but is interfaced with the Non-Volatile Memory Express (NVMe) [16] interface, that uses the Peripheral Component Interconnect (PCI), an I/O interface, and not the memory bus, making it a secondary memory, limiting its usage.

As inaccessible NVMs may still be, researches in their utilization are already a reality. To make it possible, researchers must rely on making prototypes by themselfs, like in WSP [65], or in the use of simulators, like in Kiln [99]. Those that relies in simulators

[8]Announced at: https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/

face the fact that, usually, the simulator is not ready to work with NVMs, and may need adaptations. NVM Duet [57] for example had to use both MARSSx86 [71], a full-system simulator for the x86 architecture, with the DRAMSim2 [77], a main memory simulator. Memorage [46] modified the Linux kernel to allow to simulate NVMs using the file system. Since each research creates an environment to test their own proposals, they may not be able to be reused by other researchers and, even worse, they may not be made public. Thinking in the struggle that researching the use of NVMs might be, this document describes a simulator focusing on researches in NVM utilization, and hopes that it may contribute to future's researches in this area.

The simulator currently can already simulate the impact of NVMs being deployed as main memory, like in Figure 2.4c and Figure 2.4e, and is still in development to allow researches as secondary memory and cache. It is expected that NVMs will continue to be a hot subject in both academic side, with projects like FireBox [3], and industry side, with projects like Hewlett-Packard's The Machine [49], which just highlights the relevance of the simulator.

Chapter 3

The simulator

The simulator is built upon the ZSIM [78], that is a partial simulator of the x86 ISA and utilizes the PIN [59] binary instrumentation tool to inspect binary code of programs. By utilizing PIN, ZSIM delegates to the host machine the functional simulation, resulting in a better simulation speed but restricting its usage to x86 environments. To accelerate parallel memory access simulation, ZSIM uses an epoch scheme called Bound-Weave [78], that separates each epoch in two phases, a bound phase and a weave phase. In bound phase, every memory request is treated without memory contention, that is, it receives a constant latency and does not consider the effects of other memory requests that may be occurring. Every memory request in bound phase generates an event that is used in the weave phase. In weave phase, all the events generated in bound phase are analyzed in order to account for possible contention on memory requests.

Figure 3.1 gives an overview from the simulator. It provides models for CPU, cache and main memory. To simulate the NVMs it uses the NVMain [74] simulator, that is a main memory simulator capable of simulating several NVM technologies and also the traditional DRAM. Unfortunately, there is not a model for secondary memory yet. So, for now, the simulator does not account for requests to the secondary memory.

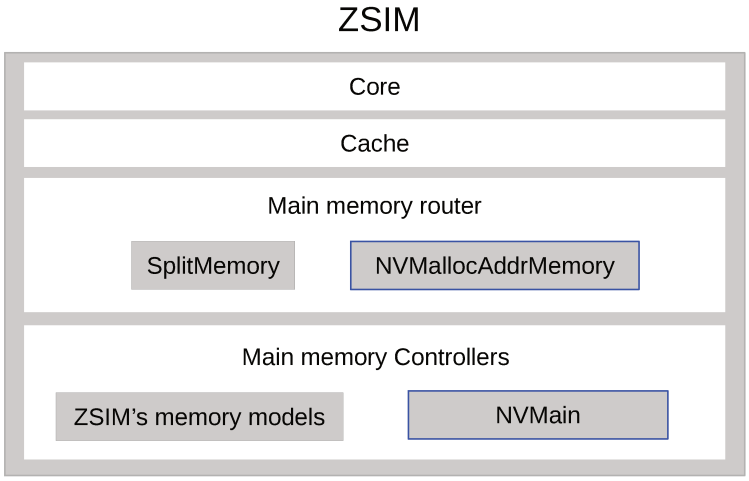


Figure 3.1: Simulator Overview.

In a hybrid main memory where it is composed by two or more technologies, a new routing scheme of memory requests might be needed to route each data to the specific

memory technology in which it must be stored. For an example, consider a main memory composed by DRAM and NVM. Memory requests for persistent data must be routed to the NVM, while transient ones to the DRAM. To simulate and make use of hybrid main memories, we created a main memory router called NVMallocAddrMemory, that works together with a proposed API called NVMalloc, that must be used in the program that will be inspected by the simulator in order to properly simulate the hybrid main memory scenario.

3.1 NVMain integration

The NVMain simulator is a trace driven main memory simulator, created with the purpose of simulating NVMs. It can work with both a precomputed trace of memory requests, or integrated with a simulator that generates memory requests, like ZSIM.

The integration between ZSIM and NVMain that exists here is the same as made by the AXLE Project [1]. It was made by modifying the most detailed DRAM's model from ZSIM, called DDRMemory, including an interface in it to the NVMain, creating a new memory model, that is called NVMain inside the ZSIM. This new memory model is capable of receiving memory requests and calculate the correct latencies for them using the NVMain. These latencies will depend on the NVMain configuration file that it was built on. As there are NVMain configuration files for NVMs, this new memory model can behave like a NVM. With this, the simulator ends up with the following main memory models:

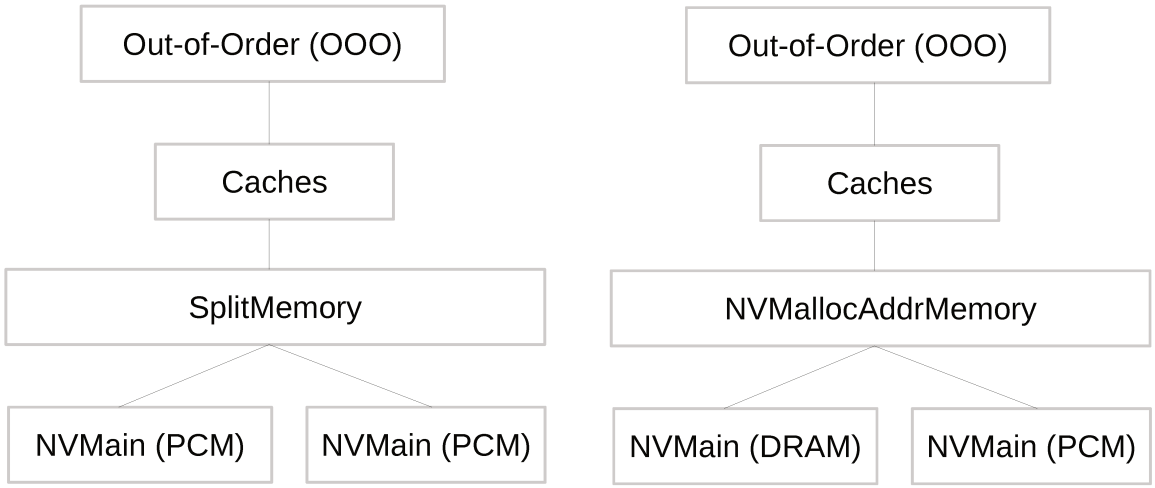
- Simple: a simple memory model that always returns the same latency in every access, and does not have support for weave phase;
- MD1: models the main memory with a M/D/1 queue, and does not have support for weave phase;
- WeaveMD1: same as MD1 but adds support to weave phase;
- WeaveSimple: same as Simple but adds support to weave phase;
- DDR: is the most detailed DRAM model from ZSIM, supports weave phase;
- DRAMSim: utilizes a memory model provided by the DRAMSim2 [77] simulator, supports weave phase;
- NVMain: utilizes a memory model provided by the NVMain simulator, supports weave phase.

To properly make use of bound weave scheme, both memory and processor must support the weave phase. ZSIM counts with the following processor models:

- Null: consider that each instruction spends only 1 cycle to execute, resulting in an instruction per cycle (IPC) equals to 1, ignoring any memory reference;

- Simple: similar to Null, but considers memory references in bound phase, resulting in every instruction spending 1 cycle, while memory references may spend more due to the latency of the memory system;
- Timing: similar to Simple, but supports weave phase, considering contention in the memory access;
- OOO: similar to Timing, but simulates a superscalar processor;

The OOO model is the most detailed processor model offered by ZSIM. It simulates a superscalar processor based on the Westmere family from Intel [52]. We propose to use the simulator using the OOO model for the processor, and the NVMain model for the main memory, as illustrated by Figure 3.2. In the case of a homogeneous main memory, where it is composed by only one technology as illustrated by Figure 3.2a, we use the SplitMemory to route the memory accesses to the different memory controllers. The SplitMemory is the standard memory router from ZSIM. It does interleaving between the memory controllers. In the case of a hybrid main memory, where it is composed by more than one technology as illustrated by Figure 3.2b, we use the NVMallocAddrMemory to route the memory accesses. In the case of the hybrid memory, the instrumented program that is running in the simulator must make use of the NVMalloc API to properly simulate this scenario (to properly make use of the NVM).



(a) Homogeneous main memory. (b) Hybrid main memory.

Figure 3.2: Proposed usage from the simulator.

Note that, in Figure 3.2, every memory controller is of the NVMain model. As already said, the timing of the NVMain model depends on the NVMain configuration file that it was built on, and besides simulating NVMs, NVMain can also simulate the DRAM. We consider fair that every memory controller is simulated using the same model instead of mixing the NVMain model with the legacy memory models from ZSIM. Since NVMain and the ZSIM are separated projects, we expect that our simulator can take advantage from advances from both of them.

3.2 Hybrid main memory and NVMalloc

To use ZSIM, one must provide a configuration file. One of the parameters from the configuration file is the main memory. Originally, ZSIM does not support different technologies in the main memory. We modified ZSIM to allow it, resulting in a change in the way that the main memory is declared in the configuration file.

In Listing 1 there are two examples of main memory declaration in the ZSIM configuration file. One, Listing 1(a), is the original way that the main memory is declared in ZSIM. The user sets the type of the memory in the parameter **type** and the number of memory controllers in the parameter **controllers**. This way of declaration did not allow the use of different memory technologies. In the new way (Listing 1(b)), a new keyword was added called **mem_controllers**, and inside this parameter, each memory controller can be declared separately. It is the same way as cache memories are declared in ZSIM, but in main memory you cannot set a hierarchical relation between them. Every memory controller in the main memory is considered to be at the same level in the memory hierarchy. Note that in Listing 1(b), we declared two memory controllers, one of the type DDR and another of the type NVMain. Note that the **techIni** parameter is specifying a NVMain configuration file that the memory model is built on. It determines what this memory will be (DRAM or NVM). Since every memory declaration is made independently from each other, it is possible to utilize any memory combination, allowing using the NVMain memory with legacy memories from ZSIM. However, we discourage this practice, because we believe, for fairness, that every memory presented in the main memory should have similar abstraction levels, what incur in using the same main memory models, just like we propose in Figure 3.2.

```
mem = {
  controllers = 2;
  type = "NVMain";
  techIni = "pcm-nvmain.config";
  envVar = "ZSIMPATH";
  outputFile = "nvmain.out";
};
```

(a)

```
mem = {
 mem_controllers = {
  DRAM = {
   type = "DDR";
  };
  NVM = {
   type = "NVMain";
   techIni = "pcm-nvmain.config";
   outputFile = "nvmain.out";
   envVar = "ZSIMPATH";
  };
 };
 splitAddrs = false;
 nvmallocIntegration = true;
};
```

(b)

Listing 1: Configuration file from ZSIM. (a) Original ZSIM main memory declaration. (b) New way to declare the main memory.

In the scenario where there are volatile and non-volatile memories in the main memory, the memory requests router should route each kind of data to the correct memory. The usual ZSIM main memory router is the SplitMemory. It uses memory address interleaving

to route the memory requests between memory controllers. It does not account for the kind of memory, making it unsuitable for a hybrid main memory scenario. To support the use of a hybrid main memory, NVMallocAddrMemory was implemented as a new main memory router. NVMallocAddrMemory works with a range of address to know if a data is persistent or not, and if it needs to be routed to the NVM or not. Listing 1(b) shows a typical NVMallocAddrMemory declaration. It must have a volatile memory and a non-volatile memory, building up a hybrid memory. NVMallocAddrMemory works together with NVMalloc, a proposed API that allows software to proper declare data in a region mapped to a NVM. A limitation of the NVMallocAddrMemory is that it only supports 2 memory controllers. The first must be the volatile memory and the second the NVM.

The file abstraction was used to persist data in the storage memory. With NVM in main memory, it is possible to persist data without the need to use a storage memory, allowing programs to decide where to allocate its data. If a datum is persistent, it must be allocated in a memory region that will be mapped to NVM. To allow programs to address their data to the correct memory region, we implemented the NVMalloc API.

Listing 2 shows the API provided by NVMalloc, that allows to work with persistent data in the heap data region of a program, just like transient data. It counts with two memory allocation functions, **pmalloc** and **pcalloc**, and one function for memory deallocation, **pfree**. They work just like the **malloc**, **calloc** and **free** provided by the C standard library, but with the guarantee that the allocated memory address belongs to a NVM.

The big difference from manipulating transient data is the concept of a root pointer. A root pointer must point to some data that is able to reach every other data that is persistent and was allocated with NVMalloc. This pointer will be used as the point of origin to recover the persistent data in future executions from the software. The function **pset_root** is used to determine the root pointer, and the function **pget_root** is used to recover the root pointer in future executions.

```
1  //Allocate size bytes at NVM
2  void *pmalloc (size_t size);
3  //Allocate size bytes at NVM and equal all of them to 0
4  void *pcalloc (size_t nmemb, size_t size);
5  //Free a persistent memory region pointed by p
6  void  pfree (void *p);
7  //Defines p as the root of the NVM data. p must point to all of the regions
8  //previously allocated at NVM with pmalloc or one of its variations
9  void  pset_root(void *p);
10 //Returns a pointer to the root of the persistent data that was defined with pset_root
11 void *pget_root();
12 //Initialize the persistent region from a dump file
13 //Initialize an empty persistent region if dump file does not exist
14 NVMALLOC_SHR_NVM_STATE *pinit (char *id);
15 //Store the persistent region to a dump file
16 void  pdump (void);
```

Listing 2: Signature of the functions provided by the NVMalloc API.

Since NVMalloc was made to still run in a computer that uses a storage memory

to persist its data, it works with a file. We call this file that holds information from persistent data allocated with NVMalloc a dump file. The function **pinit** is used to read the dump file and recover the persistent data to the programs address space. The function **pdump** is used to write the persistent data allocated with NVMalloc to the dump file. A program that uses NVMalloc should use the **pinit** in the beginning of its execution, and the **pdump** function in the end of its execution. It is important that a root pointer is determined during the program execution, using the **pset_root** function, before the use of the **pdump** function.

Listing 3 shows an example of use of the NVMalloc API. It allows a linked list to be persisted without a file abstraction. After reading a number from the standard input (line 13), it inserts or deletes an element from the list. In both cases, it uses the function `pget_root` to recover the root pointer from the persistent data, that in this case is the first element of the linked list. After the first node has changed, the root pointer is also changed with the function `pset_root`. Persistent allocations of new nodes are made with the pmalloc function (line 20) and deallocations of these nodes are made with the `pfree` function (line 18).

Note that, in this example, persistent data is manipulated in the same way as transient data. This allows that transient and persistent data to be exchanged between memories in this hybrid main memory context, without the need of translations between transient and persistent forms or making use of APIs. For example, the value of the variable `v`, that is volatile, is copied to the variable `val` from a node (line 21), that is persistent. In general, every data that is allocated with NVMalloc will be mapped to the NVM, and the rest to the DRAM.

Observe in Listing 3 that **pinit** is called in the beginning of the program (line 10), to recover the persistent data, and the pdump() is used by the end of the program (line 31). They are only needed because we still rely on files stored in the storage memory to persist the data. In a computer with NVM in the main memory, they should not be needed. To allow the program to run like if it was running in a computer with NVM in the main memory, we implemented the **pinit** and **pdump** functions in ZSIM. In this way, the program shall not need to use these functions if it is running in ZSIM. An exception here is with the function **pinit**, that is still needed to be used by the program, but will only do a part of its function. We give a further explanation about the necessity of still calling **pinit** in the program in Section 3.3.

Listing 4 shows an example of declaration of a process to be executed by ZSIM. The user declares the command to execute the process in the **command** parameter. There is the option to declare a dump file in the parameter **dumpFileName** in case the process uses the NVMalloc API. If this is the case, ZSIM will call the **pinit** function and the **pdump** for the process. Therefore, if it is executed together with ZSIM, the process does not need to use the **pdump** function and it will already have made a great part of the **pinit** function, and the recovery and persistence of the data allocated with the NVMalloc API are made almost entirely by ZSIM. This is crucial to provide the hybrid main memory scenario for the process.

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

struct ll_node {
    int val;
    struct ll_node *next;
};

int main () {
    pinit("dump.file");
    int v;
    printf ("Type a number (0 removes the 1st. element from the list):\n");
    fscanf (stdin, "%d", &v);
    struct ll_node *curr = NULL;
    if (v == 0) {//Removes the first element from the list
        struct ll_node *head = pget_root();
        if (head) curr = head->next;
        pfree (head);
    } else { //Adds an element in the beginning of the list
        curr = pmalloc (sizeof (struct ll_node));
        curr->val = v;
        curr->next = pget_root ();
    }
    pset_root (curr);
    printf("List: ");
    while (curr) {
        printf("%d ", curr->val);
        curr = curr->next;
    }
    printf("\n");
    pdump();
    return 0;
}
```

Listing 3: Example of usage of the NVMalloc API in a program to be run in ZSIM. Insertion and removal of nodes from a persistent linked list.

3.3 NVMalloc implementation

NVMalloc was built to simulate the hybrid main memory, but it still runs in the current memory system that has only DRAM in the main memory. Therefore, NVMalloc needs to make use of files stored in the secondary memory to persist the data through distinct executions. Actually, every data that was allocated with the NVMalloc API is saved into a file in the end of the execution, known as dump file, using the **pdump** function. The program may restore these persistent data in future executions by using the **pinit** function.

Figure 3.3 shows the cycle that a program that works with NVMalloc must follow to proper persist the data from the NVM. Considering the first execution of the program, several persistent data should be allocated with functions like **pmalloc** for example. These persistent data are allocated in memory regions called carriers. A carrier is the unit that NVMalloc uses to allocate memory. Every datum that is allocated through NVMalloc API is allocated inside a carrier. Carriers are allocated using the Unix system call **mmap**,

```
process0 = {
    command = "./linkedList.exe";
    dumpFileName = "dump.file";
};
```

Listing 4: Declaration, in the ZSIM configuration file, of a process that uses the NVMalloc API.

the same system call that allows files to be mapped into the programs address space [84]. To allocate the carriers, the **mmap** is used to make an anonymous map, that is, allocate a memory space not backed by a file. It works just like an ordinary **malloc**, but you can choose its virtual address. The address choice is only a hint and not guaranteed to work, although our tests showed that it usually works if called before any other memory allocation. If it is called before any memory allocation, it should not fail, since no memory allocations were made yet. Note that there are no gaps between the carriers when stored in a file, but they can be scattered in the memory when they are in the program's address space.

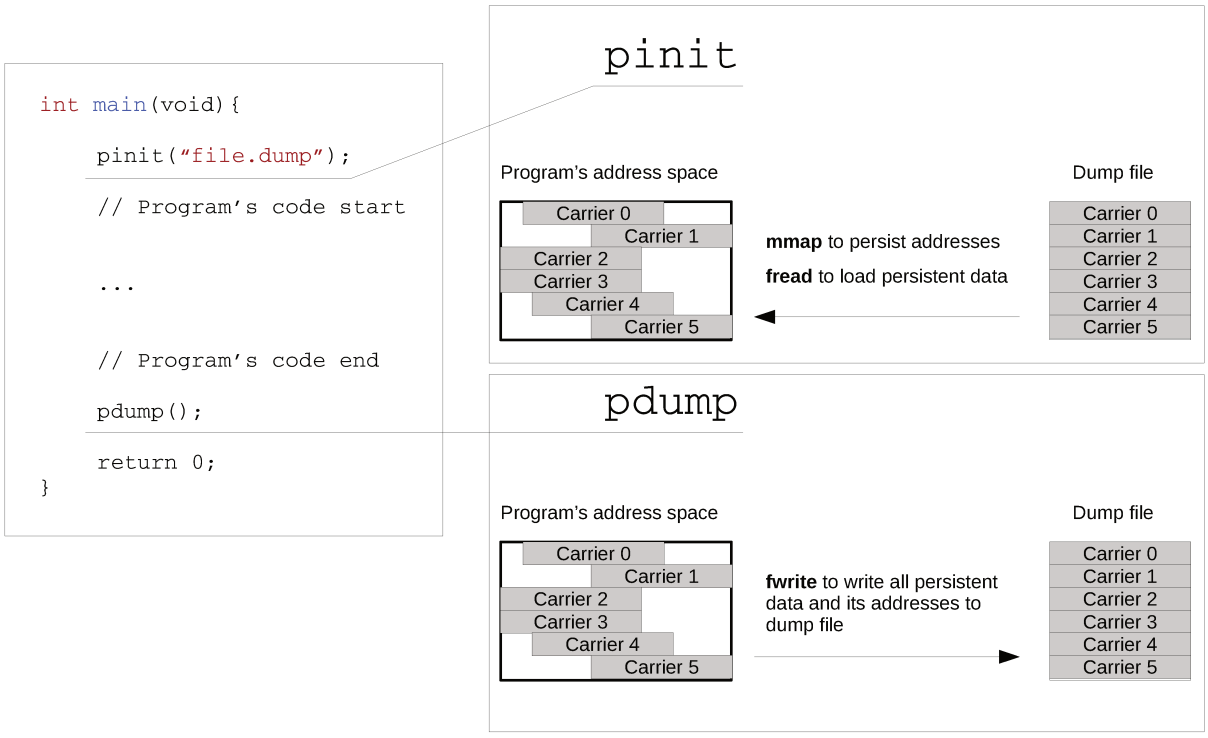


Figure 3.3: The function `pinit`, used to recover persistent data from a file and map these data to the correct addresses, and the function `pdump`, used to store the persistent data to the dump file.

Before the program's end, **pdump** should be called in order to properly persist the data. What NVMalloc does is to write all the carriers that were allocated to the dump file. It also writes the addresses from these carriers. In a next execution, the program will call **pinit**, and it will load all the carriers saved in the dump file back to the program's address space. Since this file also holds information about the carrier's addresses, it can load its data to the same memory address that it was using during the last execution,

allocating the carriers with **mmap** and loading their contents with functions like **fread**.

Observe in Figure 3.3 that the **pinit** has two functionalities: load persistent data and persist addresses. When a program that uses the NVMalloc API is executed inside ZSIM, ZSIM is able to execute **pinit** and recover the persistent data, but is unable to map this data to the process address space. Hence, the program still needs to call the **pinit** function to map the persistent data to its address space. The **pinit** call from the program will perceive that the data is already recovered by ZSIM, and will map them, using **mmap**, to its address space, and will not need to proper recover them with **fread**.

3.4 NVMalloc limitations

NVMalloc was based on another API called Atlas [13][1]. Table 3.1 shows NVMalloc limitations when compared against Atlas. NVMalloc, for now, treats NVM data like in a big chain of data, that starts from the root pointer until whatever data it can reach. It does not allow to split NVM into several chains, having more than one root pointer, like it could be done with a file system. If it is desirable to have more than one root pointer in NVMalloc, there must exist a structure that points to all the desired root pointers, and the pointer to its structure will actually be the root pointer.

	NVMalloc	Atlas
NVM data allocation	Yes	Yes
Multiple root pointers	No	Yes
Multithread execution	No	Yes
Failure resilient	No	Yes
ZSIM integrated	Yes	No

Table 3.1: Comparison between NVMalloc and the API it was based on, Atlas.

Atlas allows several root pointers. It works with the concept of persistent regions, where each region has its own root pointer. A program can work with several persistent regions, and each of them can be invoked with a function that works like the `pinit` function from NVMalloc. This treatment of the NVM resembles the interface that exists to work with shared memory regions. Each persistent region has its key ID, just like shared memories.

Persistent regions are implemented in Atlas with files in a temporary file storage (TMPFS), that resides in main memory, while NVMalloc implements it with files in the secondary storage. The way that Atlas implements it makes it easier to share persistent regions between processes, since each process just needs to know the key from each persistent region, and not where it is actually stored. NVMalloc does not support multiprocessing, and cannot provide functional allocation to more than one process or even for a multithreaded process.

Energy failures may corrupt the persistent data that resides in NVM. This is mainly for the fact that there are still volatile memory (caches) between the CPU and the NVM.

[1] Atlas repository in: https://github.com/HewlettPackard/Atlas

For example, in the code from Listing 3, where a linked list is persisted using NVMalloc, imagine that only half of the list is at NVM, and the other half is only at caches. If a power failure occurs, half of the list will be lost. To make matters worse, imagine that the root pointer from that list is still not in NVM. If a power failure occurs, all the list will be lost. Atlas provides functions to prevent data corruption in a power loss, and recovery routines to when its powered up. NVMalloc does not provide such routines, so it is still not ready to deal with power failure.

The main characteristic that outlines NVMalloc against Atlas is its integration with ZSIM, facilitating the exploration, with quantitative results, of the hybrid main memory scenario. NVMalloc provides the proper allocation of persistent data to NVMs, and we give examples of its usage in Chapter 4.

Chapter 4

Simulator case studies - Experimental results

In this chapter, we show examples of usage of the simulator. In the first experiment, we remove DRAM from the system and use a main memory composed only by NVM. As NVMs still cannot match DRAM's access time, we demonstrate the slowdown that will occur using only NVMs. After that, we analyze the case of the hybrid main memory, using the NVMalloc API, showing advantages and disadvantages from this approach when compared with the traditional persistence in the storage memory.

In total, there are 3 models of NVMs provided by NVMain presented in our tests: a PCM model based on [15], a Resistive RAM (RRAM) model based on [48], and a STTM model based on [31]. We also use a DRAM model from NVMain, that represents a DDR3 model working at 1600 MHz. There are two CPU models that we used in our experiments. One for the first experiment, described in Section 4.1, and another for the experiments in the hybrid main memory scenario, described in Section 4.2. After we have done the first experiments, we decided to change the CPU model to reduce the LLC size. Our intention was to increase the impact that the main memory performance has by using a smaller LLC. Despite this difference in CPU models, this does not affect our conclusions, since each experiment is done independently from each other, and we do not compare them directly.

All the tests were executed in a cluster of computers managed by Condor [56]. All nodes were running Linux 64 bits (Ubuntu 14.04) with kernel 3.13. ZSIM used PIN 2.14, and the NVMain version was the most recent obtained from the repository at July 1st, 2016. The compiler utilized was the GNU Compiler Collection (GCC) in version 4.8.5. It is possible that two executions from the same program in ZSIM shows different results. Unfortunately, we could not determine a worst-case scenario to these differences. However, empirically, in tests that we executed more than one time, this difference showed to be within a maximum of 1% of the highest value.

4.1 DRAM vs. NVMs

In this first experiment, where we explore the total substitution of DRAM by NVM, we utilized the out-of-order (OOO) core model, with 3 levels of cache: 64KB of L1 (32KB for data + 32KB for instructions), 256KB of L2 and 12MB of L3. The CPU frequency was set to 2.27 GHz. These parameters are based on the Intel Xeon L5640 processor.

Table 4.1 shows the slowdown obtained when executing the STREAM [61] benchmark using NVMs in comparison with using DRAM. The STREAM benchmark is a software that has an irregular memory access pattern with a big memory footprint. Big here means that it uses more memory than the last level cache can hold. We configured STREAM to work with data of 8 bytes distributed in 3 vectors, each with 6M positions, totalizing a memory footprint of 144MB, 12 times greater than LLC (12MB).

	STTM	RRAM	PCM
Slowdown	1.5	1.5	5.3
Slowdown AMAT	1.7	1.8	6.8

Table 4.1: Slowdown executing the STREAM benchmarks with NVMs compared to the execution with DRAM

There are two kinds of slowdown in Table 4.1, the slowdown from the program and the slowdown in the average memory access time (AMAT) of the main memory. PCM had the highest slowdowns, with 6.8 slowdown in AMAT and 5.3 in the execution speed. The other 2 NVMs had almost the same slowdowns, giving only 1.5 slowdown in execution speed. The slowdown in AMAT shows how slower these memories are when compared to DRAM, and the slowdown in execution speed shows how slower this program can be when using NVMs.

STREAM is a benchmark that is made to be memory bound, so the higher latencies in main memory should affect directly its performance. Programs that are less dependent from the memory should have a smaller impact in overall performance when utilizing slower memories. To also analyze programs that were not made to be memory bound, we ran a few benchmarks from SPEC2006 [40], with the reference input, to evaluate how they are affected by a slower main memory.

We did not execute all the benchmarks from SPEC2006 due to errors that we could not resolve and execution times that were too long and we decided to not complete the simulation. Hence, the benchmarks here presented from SPEC2006 are the ones that we could execute, and were not chosen for any peculiarity from them when compared with the others SPEC2006 benchmarks. In total, we took between 1 and 2 months to run this experiment.

Figure 4.1 shows the AMAT from main memory, in cycles of CPU, from SPEC2006 benchmarks. They follow the pattern from STREAM, with DRAM being the fastest and PCM the slowest.

The relation of slowdown in execution speed and in AMAT to main memory for SPEC2006 benchmarks is shown in Figure 4.2. Besides not having the highest values of AMAT slowdown between the benchmarks, **soplex** had the highest slowdown, of 1.4, 1.4

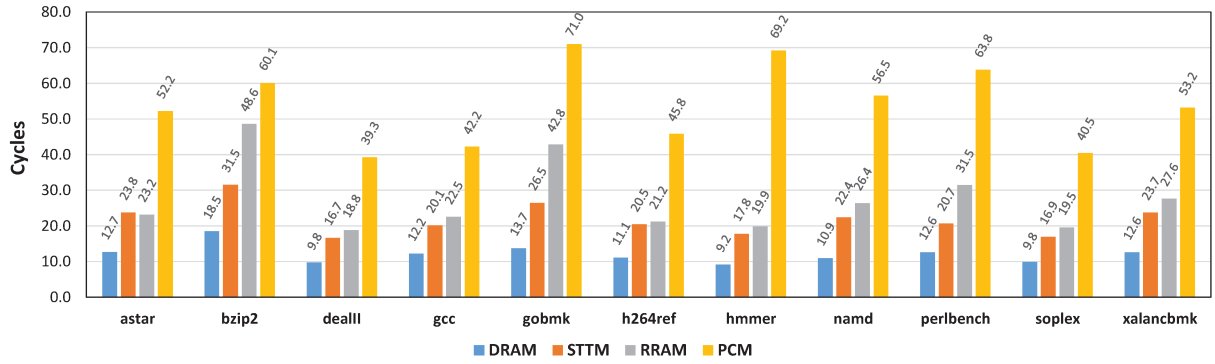


Figure 4.1: AMAT from main memory in SPEC2006 benchmarks

and 2.8 when executing with STTM, RRAM and PCM respectively. Note that **hmmer**, the one with the highest AMAT slowdown, did not have any slowdown at all in execution speed. Actually, almost half of all the benchmarks that we used from SPEC2006 had a maximum slowdown of 1.1. This shows that these applications probably can make good use of the cache memories, having small dependency from the performance of the main memory.

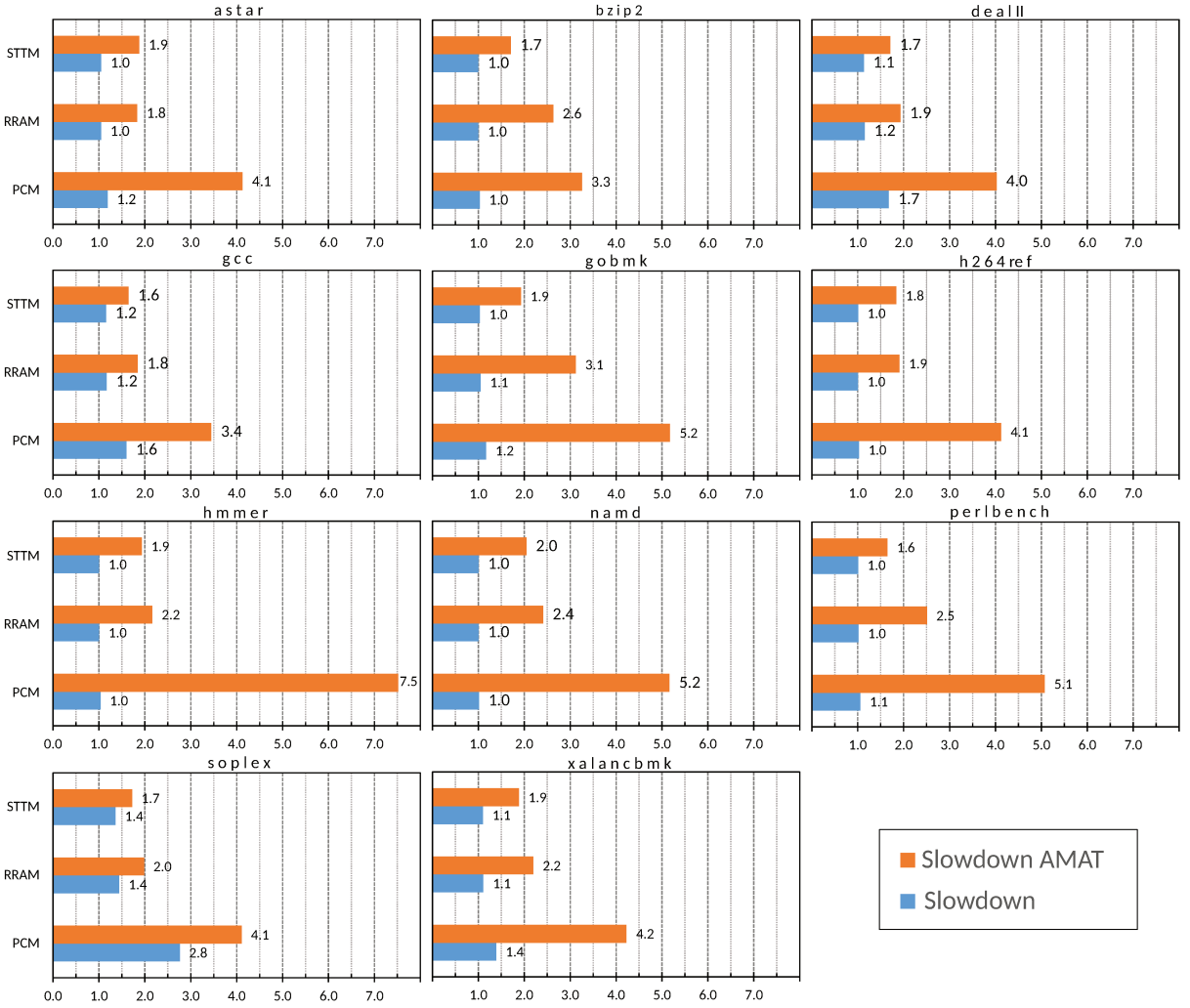


Figure 4.2: Slowdown from SPEC2006 when using NVMs as main memory

These results represent two extremes in a scenario where the main memory is made entirely by NVM. One, with the STREAM, showing a memory bound software that has its performance tightly related with the main memory AMAT. The other, the SPEC2006 benchmarks, that are more CPU bound, and have their performance less dependent of the main memory AMAT. We expect that most of the applications stays between these two extremes. And those that are closer to the SPEC2006 behavior are more tolerant to the usage of NVMs as main memory, showing that might exist opportunities for main memories composed entirely of NVMs.

4.2 Hybrid main memory

While NVMs are still unable to match the latency of DRAM, they can be used in a hybrid main memory composed by both technologies (DRAM + NVM). In this approach, DRAM is still used for transient data, while NVM gives the possibility of having persistent data in main memory. It is expected that the access to persistent data in the NVM to be faster than accessing persistent data in secondary storage, but it is not expected it to be faster than accessing transient copies of persistent data in the DRAM. For this reason, a software needs to weight how is the frequency of use of each persistent data, to know if it is worth or not to bring it to the DRAM. In this scenario, DRAM acts like a cache even for the NVM.

To test the hybrid main memory, we created two applications whose execution is described by Figure 4.3. The programs have basically two steps. In Step 1, they read data from a file, that is stored in the storage memory, and translates it to its volatile form, storing it in a data structure in the main memory. In step 2, the program consults the data structure to obtain some information. A consult to the data structure is called **query**, and the programs can make several queries. If the main memory is capable of persist data, then we might be able to leave the persistent data in its transient form, persisting it without a file. If that is the case, future executions of the programs can skip the Step 1, accessing the data that is already in the main memory.

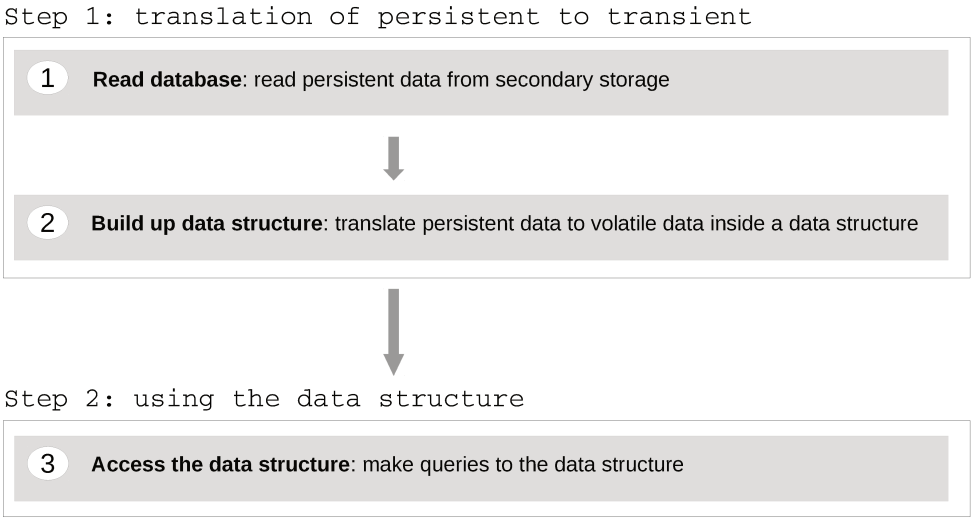


Figure 4.3: Steps performed by the programs to test the hybrid main memory scenario.

Since the total execution time of both programs is the sum of execution time of step 1 and execution time of step 2, if we are able to skip step 1, the execution time becomes only the execution time of step 2. This is where we expect that the hybrid main memory shows its advantage, since it will be able to skip step 1 after its first execution.

The speedup expected from the hybrid main memory is proportional to how representative the time of step 1 is in the total execution time. We utilized only one set of persistent data for each program, resulting in step 1 having a constant time. We vary the amount of queries, always doubling this amount. Each time we increased the amount of queries, we reduced the relevance from step 1 to the total execution time. Figure 4.4 exemplifies this fact. In this example, Step 1 takes 10 units of time (10T), while each query takes 1 unit of time (1T). With only one query, the relevance of step 1 is 90%. With 8 queries, the relevance of step 1 is 55%, being really close to lose the dominance in execution time. With 16 queries, the relevance of step 1 is only 38%, resulting in not being responsible for the majority of the execution time.

As NVMs are slower than DRAM, we expect that both step 1 and step 2 to perform slower in the hybrid main memory, but from the second execution and beyond, the hybrid main memory scenario will be able to skip step 1, and gain advantage from the current main memory (that has only DRAM). As step 2 is also slower in hybrid main memory, we expect that there will be a point where skipping step 1 will not translate in speedup anymore. The cost of each query in hybrid main memory is higher than the current main memory.

Considering the example of Figure 4.4, we expect that the hybrid main memory to have speedup over the current main memory until 8 queries, where it still is the majority of the execution time. Between 8 queries and 16, it will probably start to lose, giving slowdowns when compared to the current main memory.

Two versions of each program were created. One without the NVMalloc API, targeting the current main memory (DRAM only), and another with the NVMalloc API, targeting the hybrid main memory (DRAM + NVM). As our simulator still cannot account the secondary memory access, we make an estimation to be used in the tests. In Section 4.2.1 we explain how this estimation was done. In Section 4.2.2 and Section 4.2.3, we explain with more details these programs, and in Section 4.2.4 we show the results.

4.2.1 Secondary memory access time estimation

Step 1 involves the access to the secondary storage, something that our simulator is still unable to account. We estimate the time spent with accesses to secondary storage, outside of ZSIM, using the versions without the NVMalloc API and with 0 queries. Our estimation is based on the difference between real time and user time from the output of the command **time** of the Unix system. Figure 4.5 shows a typical output of the **time** command. User is the time spent in CPU with instructions from user space. Sys is the time spent in CPU with instructions from the kernel, and real is the program execution time. Access to the secondary storage are made by kernel code, so it is in the sys time. The sum between sys and user time may not give real time due to the multiprogramming implemented by the OS.

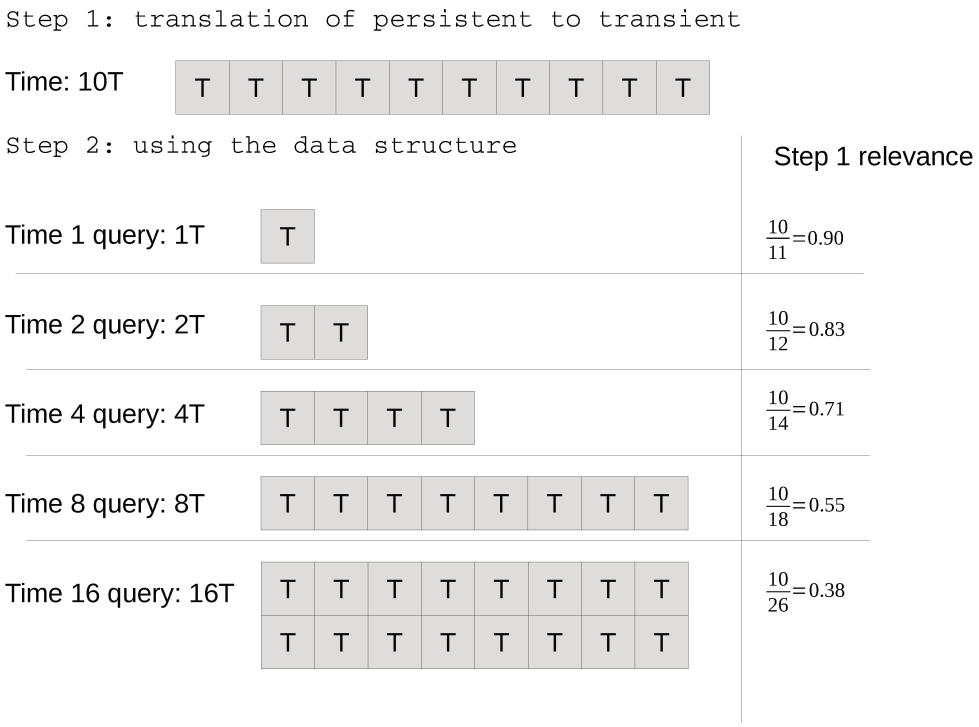


Figure 4.4: Step 1 relevance.

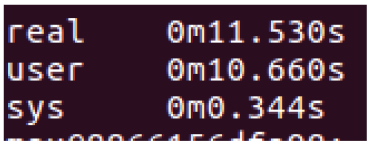


Figure 4.5: Output of command **time** from the Unix system.

We considered in our secondary storage access estimation that the difference between real time and user time represents the time spent in accessing the secondary storage. This difference represents the percentage of the total execution time relatively to accesses to the secondary storage, that we will use to estimate the secondary storage access in ZSIM. This percentage can be obtained by Equation 4.1

$$\%DiskAccess = \frac{Rt - Ut}{Rt} \tag{4.1}$$

where Rt is the real time and Ut is the user time.

With the percentage in hands, we executed the same programs (versions without NVMalloc API and with 0 queries) in ZSIM. Considering that the ZSIM's output would represent the portion of the user time, we find out what should be the total number of cycles if it accounted the secondary storage access with a simple rule of three, represented by Equation 4.2

$$\frac{ZSIMout}{Tcycles} = \frac{1 - \%DiskAccess}{1} \tag{4.2}$$

where $ZSIMout$ is the output from ZSIM in cycles of CPU and $Tcycles$ is the total number

of cycles with the estimated secondary memory access time. Since we want to give this estimation to all our executions about the hybrid main memory, we need the number of cycles spent accessing the secondary memory. We can easily obtain this subtracting the output of ZSIM from the total number of cycles, just like indicated by Equation 4.3.

$$Disk\ cycles\ estimation = Tcycles - ZSIMout \tag{4.3}$$

We can sum the estimated disk cycles to all our experiments about hybrid main memory that used the secondary storage (the ones that only uses DRAM). Since step 1 is constant, all the secondary memory access time is the same in all executions. It is important to point out that each program has its own disk access time estimation. Hence, this estimation was made for both the programs.

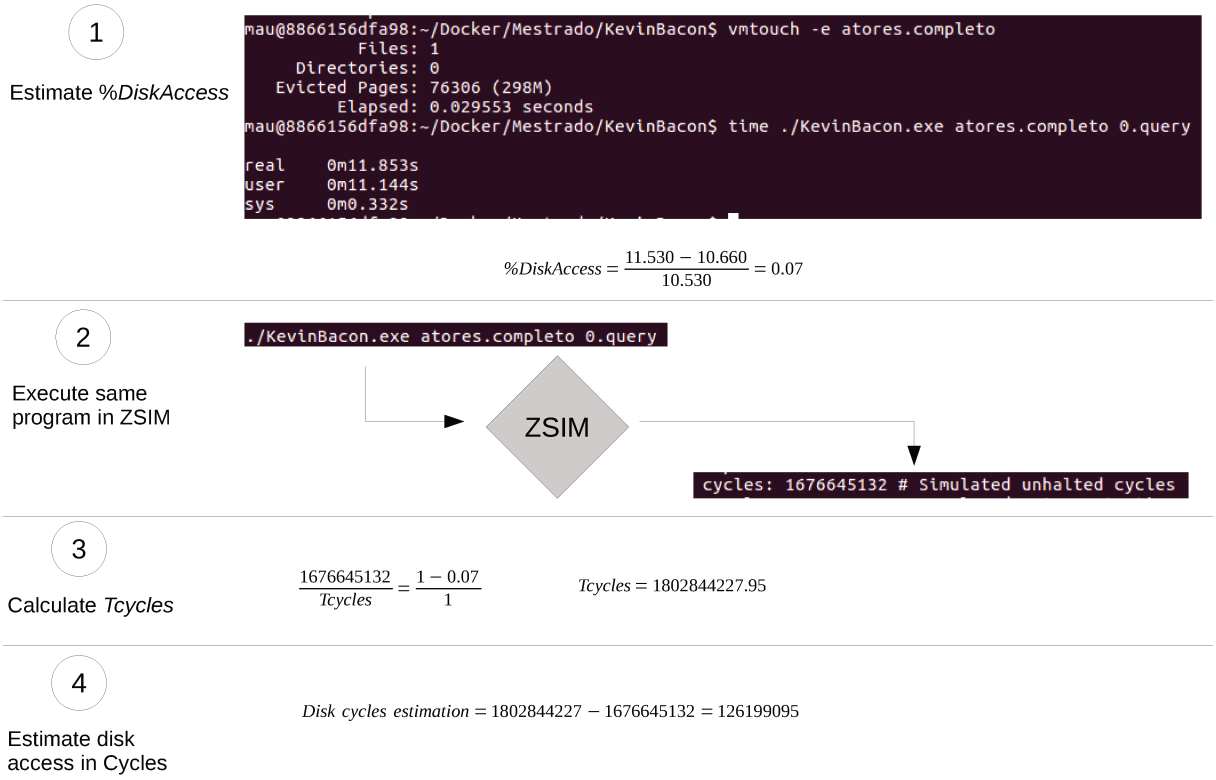


Figure 4.6: Estimation of the secondary memory access time.

Figure 4.6 summarize the steps taken to estimate the secondary memory access time and also exemplifies it. For each program, we executed this estimation 10 times, and we considered the arithmetic average from these 10 executions to be the disk cycles spent on secondary memory access. Note in Figure 4.6, in the first step, where we estimate the $\%DiskAccess$, that before we execute the program with the **time** command, we execute a program called **vmtouch**[1], that is used to clean the caching of the files in the main memory by the OS. If it was not done, in the second execution and so on, the file would already be in the main memory and the secondary memory would not be accessed.

The computer that was used to estimate the secondary storage access has the following configuration:

[1]vmtouch can be found at the following url: https://hoytech.com/vmtouch/

CPU: Intel core i7 3770k
Motherboard: Asus Maximus V Formula
Main Memory: 24GB Corsair Vengeance CML16GX3M2A1600C9
Hard Disk Drive: 1TB Seagate Barracuda ST1000DM003-1CH162
OS: Ubuntu 14.04 kernel 4.4.0-83-generic

4.2.2 Case study 1 - Adaptive Radix Tree

Our first program to explore the hybrid main memory is about the usage of an enhanced version of a trie [34]. A trie is a data structure that serves to store pairs of keys and values in an efficient way, acting as an index for fast content retrieval. The term trie was conceived because of the word reTRIEval, that is just what this structure is made for. A trie is a tree that stores keys on its edges, and values on its nodes. The keys of a trie are sequences of symbols, and the set of these symbols that can compose a key is called alphabet.

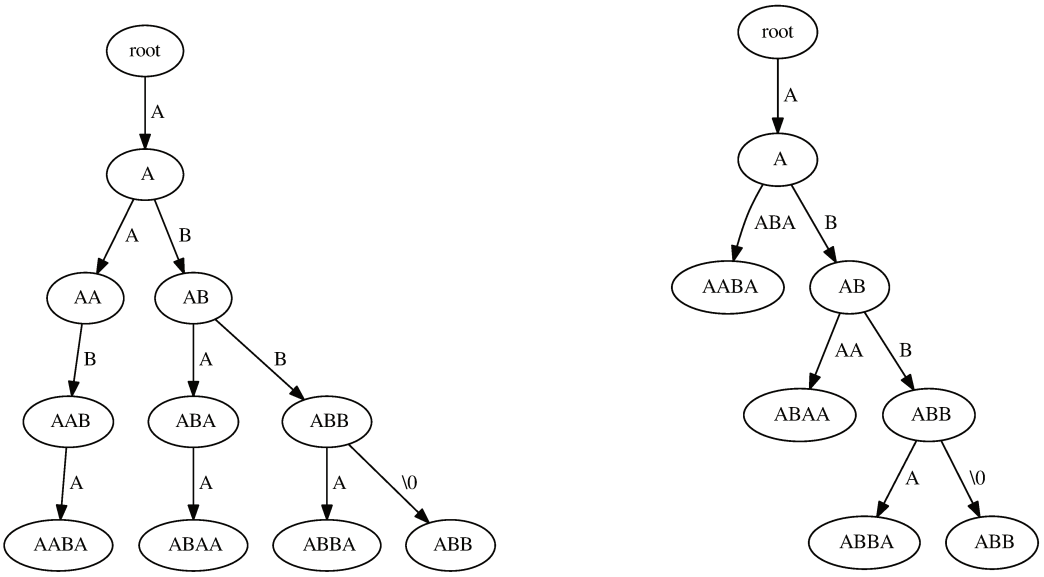
Trie and Radix Tree

Figure 4.7a shows and example of a binary trie, that contains the keys AABA, ABAA, ABBA and ABB. It is called binary because its alphabet contains only two symbols: **A** and **B**. Like already stated, the keys are stored on the edges. They are written inside the nodes only to illustrate the path that was followed to reach its node, but they are not actually inside these nodes. To check if a trie contains a certain key, the key is used to follow a path in the trie, using its symbols from the leftmost symbol to the rightmost symbol, just like can be seen in Figure 4.8.

In Figure 4.8a, the key ABB is used in a query. It follows the edges, beginning in the root of the tree, until it reaches a leaf. The last character from ABB is consumed in step 3. When the last character is consumed, it checks if the current node is a leaf or not. If it is a leaf, then the key is currently in the trie. If it is not a leaf, then it checks if it has a path for the terminator symbol. A terminator symbol serves to delimitate the end of the key. It is an extra symbol to be accounted in the trie, as it cannot compose a key. If it has a path for the terminator symbol, it follows it and reaches a leaf, concluding that the key is currently in the trie[2]. In Figure 4.8b is shown an example of search where it is not needed to make use of the terminator symbol. In Figure 4.8c, the key AAA is used in a query. In step 3 of the search, it cannot follow any path, being unable to continue, and concluding that this key is not in the trie. Note that a search for a key that is not in the trie tends to be faster than a search for one that is, because it does not need to reach a leaf, and can end in the inner nodes.

Every key has some value associated with it, and it can be any data. For example, imagine that a trie contains names of people as a key, and telephone number as value. Considering the trie from Figure 4.8, the leaf where terminates the search for the key AABA would contain the telephone number of a person called AABA.

[2]It is possible to implement a trie that does not need to reach a leaf to conclude that it has certain key. The only requirement is to have some variable that marks if is acceptable a search to end in a certain inner node



(a) Binary trie. (b) Binary radix tree.

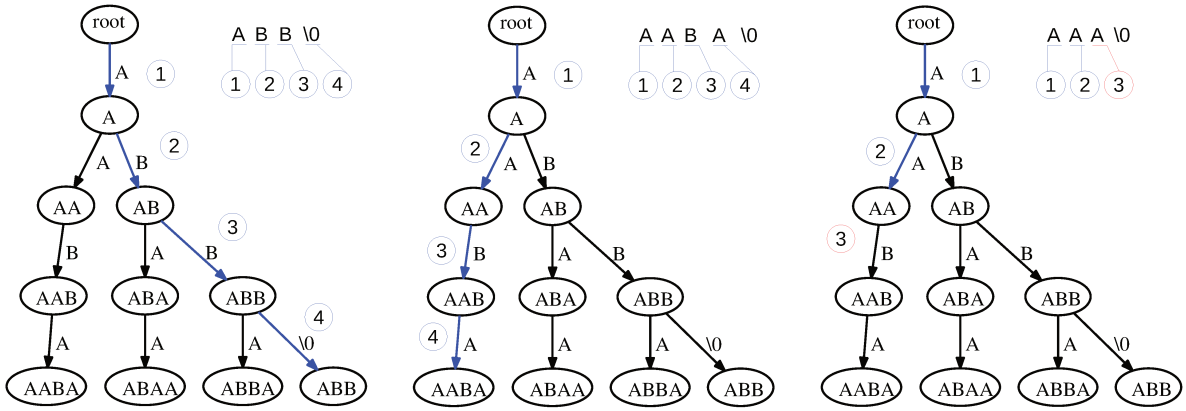
Figure 4.7: Examples of binary tries with the following keys: AABA, ABAA, ABBA and ABB.

An issue with the trie structure is when you have long keys that do not share a long prefix with other keys presented in the trie. Take as an example the key AABA in the trie from Figure 4.8. It has only the first letter in common with another key. A search for AABA must walk three steps after the first letter (for its ABA suffix) until it realizes that it is in the trie. An optimization in the trie structure is to group these suffixes, which do not belong to any prefix from other keys, in a single edge. Tries with this optimization are called radix trees[3]. Figure 4.7b shows an example of a radix tree. Note that a search for the key AABA can walk only one step after comparing its first character.

The node from a trie usually consists of a vector of pointers, where each position of this vector corresponds to a possible path to be followed. The number of possible paths is equal to the size of the alphabet used in the trie. In the case of binary trie, each node has a vector with two pointers, one for each symbol. In the case of Radix tree, this limitation in the number of possible paths is the same. If a node in a radix tree already have a number of paths equal to the size of its alphabet, a new insertion of path on its node certainly have a common prefix with the path of another key, and therefore a new node should be created instead of a new path to the already saturated node. Figure 4.9 exemplifies this fact from Radix trees.

A way to reduce the number of steps performed in a search in a trie is to extend its alphabet. Symbols can be grouped together to form new symbols. The extended alphabet is not used to compose the keys, but is used to store it. Figure 4.10 shows an example from a binary trie with extended alphabet. Each key in the trie is stored in groups of 4

[3]They are also known as Radix trie, Compact trie and Patricia tree. This last one due to the Practical Algorithm To Retrieve Information Coded In Alphanumeric (PATRICIA) presented in [63].



(a) Query: ABB. (b) Query: AABA. (c) Query: AAA.

Figure 4.8: Examples of queries in a binary trie.

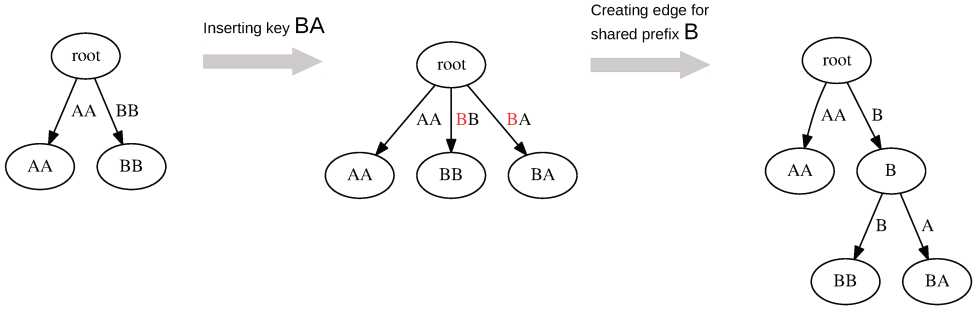


Figure 4.9: Example of insertion of a new key in a node from a binary Radix tree. Note that the possible paths from each node is also 2, just like the binary trie.

symbols. The possibilities of keys are still the same, but the possible paths to follow in each node are increased[4] from 2 to 80. Note that the key ABB has less than 4 characters, making necessary the use of the terminator symbol in the extended alphabet.

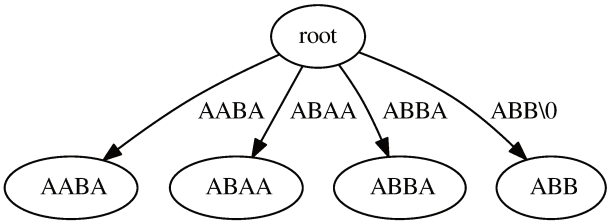


Figure 4.10: Example of a trie with extended alphabet, using a sequence of 4 symbols to compose a new symbol.

An increase from 2 to 80 pointers in each node represents an increase of 40 times in the number of pointers per node. From an implementation point of view, this increase in the number of pointers translates to an increase in memory usage. That may not be a problem if those pointers are all being used, but if they are not, they represent an overhead in the total memory usage. Considering the example from Figure 4.10, the root pointer has only 4 possible paths that takes to a key, the other paths are not being used.

[4]Perceive that we are disconsidering the path to the terminator symbol in both accounts. Counting it, it would be 3 and 81 possible paths.

In other words, there are 76 pointers not being used in the root pointer, just occupying memory space.

The radix tree, not only can decrease the tree height, but also can decrease the memory space overhead caused by unused pointers, since it may reduce the number of nodes when compared with the trie (as seen in Figure 4.7). One characteristics from the radix trees is that every inner node, except the root node, has at least 2 children. Figure 4.11 shows different overheads of a node from a radix tree, depending on the used alphabet. In the case of 2 symbols (binary radix tree), there is not any overhead, since every inner node has at least 2 children. This guarantee of 2 children in the inner node is maintained in bigger alphabets. Observe that for an alphabet with 256 symbols, like the American Standard Code for Information Interchange (ASCII) for example, that can be represented with 8 bits, the overhead can reach 254 null pointers. If we consider a pointer with a size of 8 bytes, that is a usual size for 64 bits systems, 254 null pointers translate to around 2KB of memory overhead.

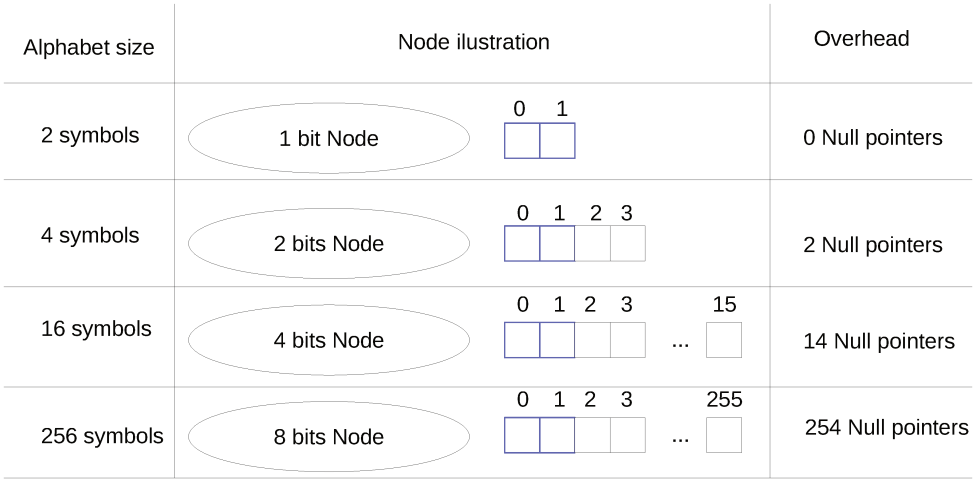


Figure 4.11: Examples of possibles overheads in a node from a radix tree depending on its alphabet size.

ART - Adaptive Radix Tree

The memory overhead, besides spending memory space, increases the sparseness of the useful data, what probably harms the spatial and temporal locality of the program that uses the radix tree. The Adaptive Radix Tree [53] (ART) tries to solve this problem. It is a radix tree that counts with 4 different kinds of nodes, each one for a certain number of possible paths. It works with an alphabet of 8 bits, that is, that has 256 symbols, resulting in having a maximum of 256 possible paths per node. As the number of possible paths change in a certain node, it can be replaced by one that counts with more pointers or by another that has fewer pointers, resulting in having nodes more adapted to the number of possible paths that they really have, decreasing the memory overhead.

We adapted an existing implementation of the ART[5] with the NVMalloc API. Our

[5]The source code from ART that we used can be found in the following repository: https://github.com/armon/libart

adaptation is able to build an ART, fill it with a set of keys and persists it using the NVMalloc API. From the second execution of the program and beyond, if the ART is the same, it does not need to build it again, since it is persisted from a previously execution.

There is also an adaptation of ART to work with the **libvmem**, that is a library that provides an API with the same functionality of NVMalloc. **libvmem** is a library that composes, along with another 7 other libraries, the NVM library[6], that is a set of libraries to work with NVMs in the main memory.

4.2.3 Case study 2 - Kevin Bacon Game

Our second program to explore the hybrid main memory is an application that resembles the Kevin Bacon Game[7]. In this game, actors are nodes of a graph, and they are connected with each other if they already participated in the same movie. The player provides the name of an actor as input and the program answers with the shortest distance between the given actor and Kevin Bacon. The objective is to dare who ever plays it to provide an actor that has a certain minimum distance from Kevin Bacon. We utilized a more generalized approach that does not need to focus only on the actor Kevin Bacon. Two actors must be provided to the program, and it will answer with a shortest path between these actors (one of the shortests if more than one exists).

In our implementation, the graph contains nodes representing movies and nodes representing actors. An actor is connected with a movie if this actor participated in this movie, and, in other words, a movie is connected with an actor if it belongs to the list of movies that this actor has worked. Therefore, there are no connections between actors, and there are no connections between movies. The graph is bipartite, with actors on a side, and movies on the other. To maintain coherence with the original Kevin Bacon Game, the length of a path is not the number of connections, or edges, that it contains, but the number of movies it contains. That ends up being the half of the number of connections. Figure 4.12 shows an example of a graph from our Kavin Bacon Game, composed by 3 movies and 5 actors.

To find the shortest path between two actors, lets say actor **A** to actor **B**, we utilize the Breadth-First Search (BFS) algorithm [21, Chapter 22, p. 594], starting in the node of actor **A**, until it reaches actor **B**. In this algorithm, it is verified every node that is a distance of 1 from actor **A**, and then every node that is a distance of 2, and so on, until it finds actor **B** or is unable to increase the distance (in this case, **B** is unreachable from **A**). If there is not a path between **A** and **B**, we say that the distance between these nodes is infinite.

Figure 4.13 shows examples of queries in the graph. Figure 4.13a ilustrates the search from the actor Hugh Jackman until the actor Kevin Bacon. First, all the nodes from distance of 1 are verified. As the graph is bipartite, all the nodes with distance 1 from the actor are referent to movies, and certainly none of them is the aimed actor. After verifying every node with distance of 1, it starts to verify the ones with distance of 2. In this step,

[6]Informations about the NVM library can be found at the following url: http://pmem.io/

[7]A deeper discussion about this game can be found at [18] and, by the time of the writing of this dissertation, it can be played at the following url: http://www.oracleofbacon.org/

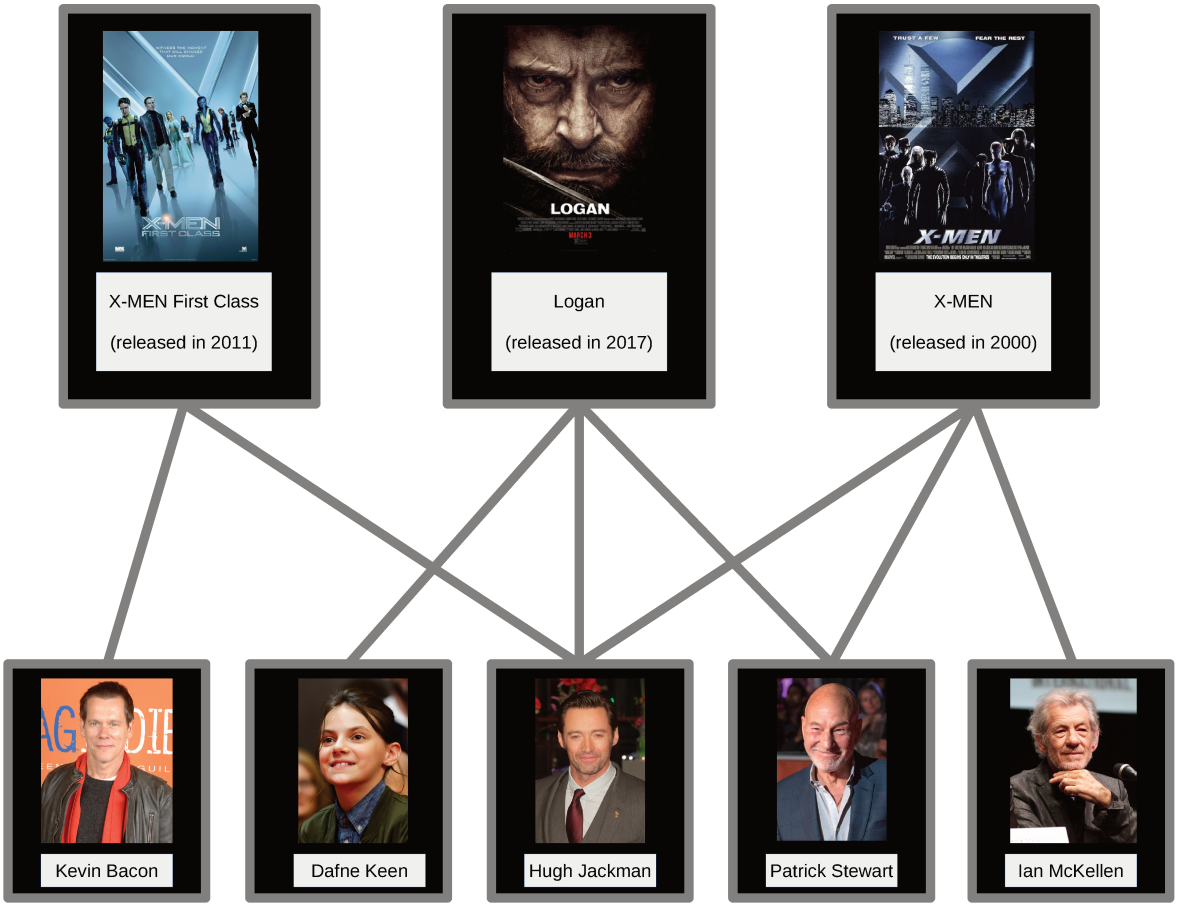


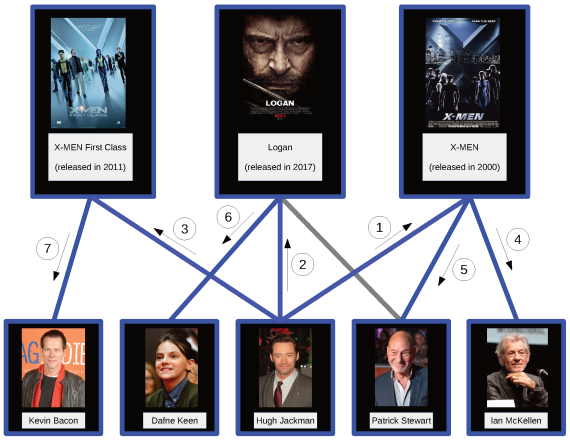
Figure 4.12: Example of a graph used in our Kevin Bacon Game[8].

it starts to look for all the actors that participated from the movie **X-MEN**, that Kevin Bacon did not work. After that, everyone from the movie **Logan** is verified, that also has not the participation of Kevin Bacon. Finally, actors from the movie **X-MEN First Class** are verified and Kevin Bacon is found, resulting in Kevin Bacon having a distance of 1 from Hugh Jackman (remember that our distance is the half of the edges of the path).

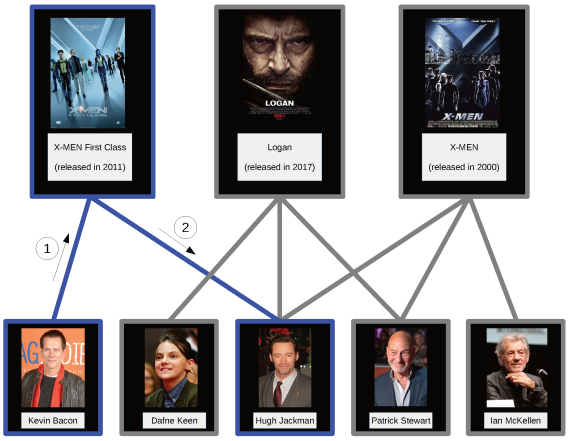
In our example, a search for the distance between Hugh Jackman and Kevin Bacon resulted in verifying all the nodes of the graph. It took 7 steps to find Kevin Bacon. The same does not occur if we invert the query. Figure 4.13b shows the search of the shortest path from the actor Kevin Bacon to the actor Hugh Jackman. This search ends with only two steps, resulting in a much faster search.

We implemented the nodes and edges that compose the graph with two structures shown in Listing 5. The node, Listing 5(a), is composed by a string **name**, that stores the name of a movie or the name of an actor, a linked list pointed by **link** that contains the information of the edges from this node, two variables, **color** and **prev**, to be used in the BFS algorithm, and a pointer to a next node to be used in a linked list for a faster localization of this node. All the nodes are included in a hash table to accelerate

[8]X-Men movies images belong to 20th Century Fox. We obtained them through the Wikipedia website. The pictures of the actors we also got through Wikipedia. We only used these images to exemplify our Kevin Bacon Game, and we do not claim to have their property. They had their resolution reduced, enforcing their use as fair use, and as *política de isenção de doutrina* in Brazil.



(a) Search from Hugh Jackman to Kevin Bacon.



(b) Search from Kevin Bacon to Hugh Jackman.

Figure 4.13: Examples of the BFS algorithm to find the shortest in our Kevin Bacon Game.

its location. The edges, Listing 5(b), contains a vector of pointers **target**, of length **EDGE_GROUP**, to store the addresses of the nodes connected with the node that contains this edge, a counter **amount** to indicate how many pointers the vector actually has (that are not NULL), and the pointer **next** to form a linked list if more pointers **target** are needed. The length **EDGE_GROUPS** is a parameter to be defined in the program, and in our implementation we set it to 10.

```
1  struct node{
2    // Name of the actor or movie
3    char *name;
4    // Edges from this node
5    struct edge *link;
6    // Color for BFS
7    int color;
8    // Previous node for BFS
9    struct node *prev;
10   // Linked list for direct search
11   struct node *next;
12 };
```

(a)

```
1  struct edge{
2    // Pointers to movies or actors
3    struct node *target[EDGE_GROUP];
4    // Number of pointers in target
5    int amount;
6    // Pointer to form a linked list
7    struct edge *next;
8  };
```

(b)

Listing 5: Code of the struct of (a) node and (b) edge in our implementation of the Kavin Bacon Game.

Nodes and edges are created as soon as an entry for a movie or an actor is read from a database. Edges are created in pair of pointers. For example, if we discover that Hugh Jackman worked in the movie **Logan**, a pointer from the edge structure from the node referent to the movie **Logan** must point to the node referent to Hugh Jackman, and a pointer from the edge structure from the node referent to Hugh Jackman must point to the node referent to the movie **Logan**. The vector of edges, **target**, in the edge structure is an attempt to force a spatial locality with the pointers of the edge. Note that, besides

this attempt, every allocation of nodes and edges are made individually, in a way that may strongly take to a great sparseness of the data, resulting into a bad spatial and temporal locality of its data in our implementation of the Kevin Bacon Game.

4.2.4 Results

Listing 6 shows the code from the main function from both our ART and Kevin Bacon Game program. In the ART implementation, in Listing 6(a), the function **pinit** is called to recover the persistent data (line 7 or 9). The function **pget_root** is called to try to recover an ART from a previously execution (line 12). If there is a persisted ART, **pget_root** will return its address, and the function responsible to build the tree can be skipped (condition from IF in line 13 will be false), and the execution goes straight to the **makeSearches** function (line 16), that is responsible to execute the queries. A query in our ART program is to check if a certain key is in the ART or not.

The execution from the Kevin Bacon Game follows the same pattern as the ART program, as can be seen in Listing 6(b). It first calls the **pinit** function to recover the persistent data, then tries to get the address of a persisted graph, using the **pget_root** function (line 12). If there is a persisted graph, then the execution can skip the function that builds up the graph (condition from IF in line 13 will be false) and goes straight to the **makeSearches** function (line 17), that is responsible to execute the queries. A query in the Kevin Bacon Game is to return the shortest path between two actors. Note that, in both examples, the **pinit** function is used and the **pdump** function is commented. That is because we intended to run these applications with ZSIM, and as already explained in Section 3.3, **pinit** still needs to be called even in this case, because it still needs to map the persistent data to the programs address space.

```
1  int main(int argc, char **argv){
2    if(argc < 3 || argc > 4){
3      printf("[ERROR] Parameters\n");
4      return -1;
5    }
6    if(argc == 4){
7      pinit(argv[3]);
8    }else{
9      pinit("art.dump");
10   }
11   art_tree *artTree;
12   artTree = pget_root();
13   if(artTree == NULL){
14     artTree = buildTheArtTree(argc, argv);
15   }
16   makeSearches(artTree, argc, argv);
17   //pdump();
18   return 0;
19 }
```

(a)

```
1  int main(int argc, char **argv){
2    if(argc < 3 || argc > 4){
3      printf("[ERROR] Parameters\n");
4      return -1;
5    }
6    if(argc == 4){
7      pinit(argv[3]);
8    }else{
9      pinit("kb.dump");
10   }
11   pStruct *pRoot;
12   pRoot = pget_root();
13   if( pRoot == NULL ){
14     pRoot = buildTheGraph(argc, argv);
15   }
16   colorThreshold = pRoot->savedColorThreshold;
17   makeSearches(pRoot, argc, argv);
18   pRoot->savedColorThreshold = colorThreshold;
19   //pdump();
20   return 0;
21 }
```

(b)

Listing 6: Code of the main function from both our (a) ART program and (b) Kevin Bacon Game.

To compose the hybrid main memory in our tests, we utilized the DRAM with the

PCM. We chose the PCM because, as indicated by our experiments from Section 4.1, it was clearly the slowest of the NVMs from our tests. The utilization of the PCM represents a worst-case scenario when compared with the other 2 NVM models that we utilized. The CPU model utilized in the hybrid main memory experiments is the out-of-order (OOO) with 2 levels of cache: 96KB of L1 (64KB for data + 32KB for instructions) and 2MB of L2. The CPU frequency was set to 2 GHz. All the tests here utilized the disk access estimation explained in Section 4.2.1.

Results from ART

To test the hybrid main memory with our ART program, we utilized an ART containing 10 million unique keys. Each key having a minimum length of 39 and a maximum length of 55. The average length of the used keys is 46, and the size of the file containing these keys is 446MB. We tested the performance of ART program from executing in the current memory system (DRAM only), that must access the secondary storage, build up the ART from a file, and then process queries for it, against the second execution of the program in a hybrid main memory, where the ART is already persisted, making unnecessary to build it (since its already built).

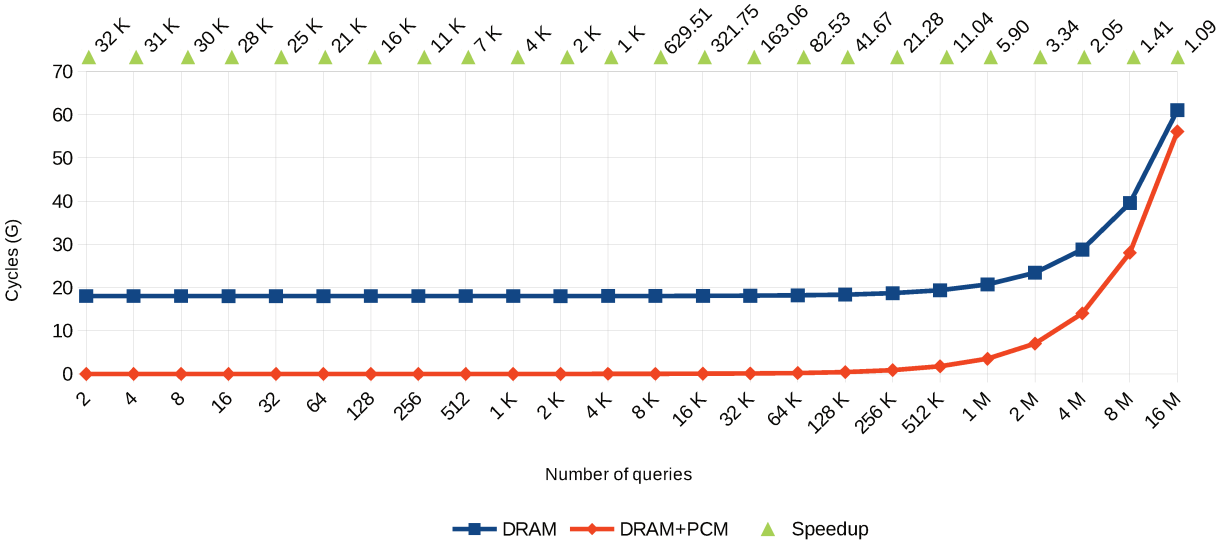


Figure 4.14: Performance from our ART program in the second execution in the hybrid main memory.

Figure 4.14 shows the results from our tests with the ART program. In the X axis there is the number of queries in the execution (in the **makeSearches** part from Listing 6(a)), and in the Y axis is time, measured in CPU cycles. Note in the upper part of the graph, there are the speedups from each execution. Each query is a search for a key that is presented in the tree. We chose to only use keys that belong to the ART because they need to reach a leaf node to end. Keys that are not presented in the tree may finish in the inner nodes, possibly representing a faster search. All the keys were shuffled to build the queries, utilizing the **shuf**[9] command presented in the Linux system. While the amount

[9]Manual for the shuf command: https://linux.die.net/man/1/shuf

of queries is inferior to the amount of keys presented in ART, every query is unique, and there is not a repeated query in the execution. The last execution, with 16M queries, has a few duplicated queries.

Perceive that the ART program outperforms the DRAM by 32K when executing only 2 queries. In fact, ART is able to sustain its performance for even larger queries. With 8M keys, the hybrid main memory provides a 41% of speedup. We decided to double this amount of queries, extrapolating the number of keys presented in ART, resulting in an execution of 16M queries. Even in this case, the hybrid main memory still got and advantage of 9% speedup when compared with the current memory system that only has DRAM.

Results from Kevin Bacon Game

To test the hybrid main memory with our Kevin Bacon Game implementation, we utilized a part of the Internet Movie Database (IMDB)[10], that covers only movies, to build up the graph. In total, this database contains around 3 million actors related with around 780 thousand productions. Figure 4.15 shows the results from our tests with our implementation of the Kevin Bacon Game. In the X axis is the number of queries in the execution (in the **makeSearches** part from Listing 6(b)), and in the Y axis is time, measured in CPU cycles.

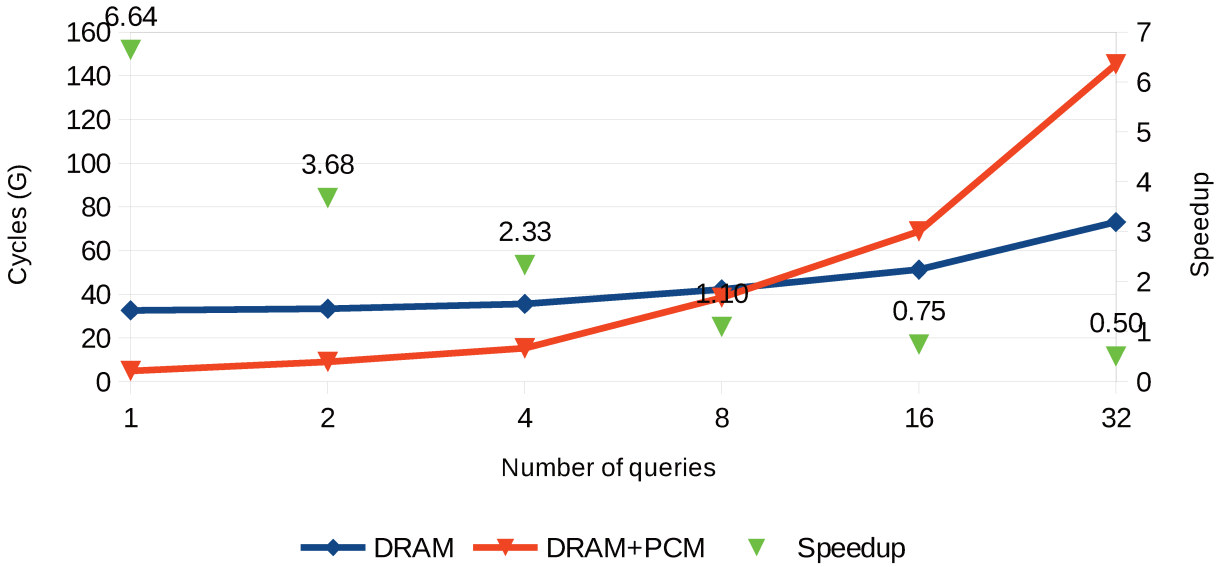


Figure 4.15: Performance from our Kevin Bacon Game implementation in the second execution in the hybrid main memory.

It is worth to point out that each query from this experiment is a pair of two actors, and we always make the search from the first actor to the second actor. As searches in the graph may have drastically differences in cost (as exemplified by Figure 4.13), each point presented in Figure 4.15 is the arithmetic average of 10 different executions. In the total there are 10 different queries for each amount of queries, and we considered the arithmetic average from the results of these 10 executions to be our final result. Just like

[10]IMDB website: www.imdb.com

in our ART queries, we utilized the **shuf** command to generate the queries for the Kevin Bacon Game.

Differently from the ART program, our Kevin Bacon Game showed much more humble results. Executing only 1 query, it has 6.64 of speedup, that is close to the speedup that we got with 1 million queries in ART. The results with 8 queries is almost the crosspoint between the hybrid main memory performance and the current main memory that has only DRAM, incurring in only 10% speedup. From 16 queries and beyond, the hybrid main memory loses its advantage, resulting in only slowdowns.

Veredict from hybrid main memory

As already stated, a program that uses the hybrid main memory must weight the frequency of usage of its persistent data to check if is worth or not to transfer it to DRAM. The ART, as stated by their authors [53], can make good use of the cache memories, something that appeared to be true with our results, reaching speedups up to 32K and hardly having a slowdown. On the other hand, our implementation of the Kevin Bacon Game utilizes data in a way much more sparse. Every node can point to nodes that can be far away from them in the memory. Even the edges from a same node may be afar from each other in memory. This results in a bad usage of the cache memories and having a tightly dependence from the performance of the main memory. Fact that takes our Kevin Bacon Game to start showing slowdowns in much more modest amounts, with 16 queries.

Besides the bad use of the cache memories, the cost of each query in Kevin Bacon Game might be much greater than the cost of a query from ART. In ART, in the worst case, the number of nodes visited will be the length of the key, that in our case had a maximum length of 55. In the Kevin Bacon Game, the number of nodes might be much higher. For example, as an initial lower bound of visited nodes, all the movies that a certain actor **A** has done will be visited in a search starting from **A**. In our database, Kevin Bacon has a degree of 135, meaning that in a search starting from him, more than 135 nodes will be visited, and that is more than the double of the worst case from ART.

The case of the ART and Kevin Bacon Game resembles our example from STREAM and the benchmarks from SPEC2006. Programs that are highly dependent on the performance of the main memory, like Kevin Bacon Game and STREAM, may have its performance highly debilitated when utilizing NVMs. For these cases, it is worth to bring the persistent data to the DRAM, making the NVM to act just like a fast storage memory.

Chapter 5

Conclusion and future work

In this work, we proposed a simulator that is capable to explore the use of NVMs in the main memory. It was based on an existing integration between the ZSIM and the NVMain simulator. We allowed the use of more than one kind of memory in the main memory, permitting the hybrid main memory to be simulated, and proposed an API called NVMalloc, to be used together with the simulator, in order for a program to be able to use the non-volatility of the main memory. Results obtained during the development of this simulator were already published in two occasions. The first as a short paper [70] and the second as a full paper [69].

To exemplify the usage of the simulator, we first explored the total substitution of DRAM by NVMS. In total, we explored 3 NVMs: PCM, STTM and RRAM. We tested this substitution running the benchmark STREAM, that is highly memory bound, and a few benchmarks from the SPEC2006, that are more CPU bound. We observed that in STREAM we had slowdowns as high as 5.3, but in SPEC2006, at least half of the benchmarks had a maximum of 10% slowdown, indicating that the NVMs performance might be tolerable in some cases.

For the hybrid main memory scenario, we presented two applications that might make use of this configuration. One, the Adaptive Radix Tree (ART), being a structure for fast content retrieval, makes good use of the cache memories and have its queries in a well defined upper bound. The other, Kevin Bacon Game, however, does not worry that much with good spatial and temporal locality of its data. It also has an undefined upper bound for its queries, allowing that two different queries might have a drastically difference in its cost. The results showed that the ART is capable of getting a great advantage in persisting itself in main memory, showing speedups as high as 32k. In the opposite direction, Kevin Bacon Game, with only 8 queries, already almost matched the execution from the current main memory (DRAM only) with the hybrid main memory. In this case, it might be advantageous to transfer the whole graph from Kevin Bacon Game to DRAM, leaving the NVM as a fast storage memory.

Future work

As future work, the first thing that we hope to do is to be able to account for the secondary memory access. Our estimation here is not very accurate, since computers already count

with hardware prefetchers and are able to overlap execution with secondary memory access (superscalar processor). We also want to simulate a page table. With the account of secondary memory access and page table simulation, we will be able to explore the effects of swapping from main memory to storage memory caused using more memory than the main memory capacity, possibly having its penalties diminished by the use of NVMs.

NVMalloc, as indicated by Table 3.1. is still very limited, having plenty space to be improved. We intend to adapt NVMalloc to be used by multithreaded programs. For its crash consistency, we can both implement it as software, in NVMalloc, or we can rely on hardware solutions. ThyNVM [76] for example utilized memory remapping in the main memory, providing a transparent checkpoint and restore for the software. We can improve this scheme with other memory remapping approaches, like page overlays [80] for example, that is used to reduce memory usage but may be adapted as a checkpoint/restore scheme.

There are several points that were not even scratched by our work here, like software protocols for accessing both main memory and storage memory, energy consumption, memory endurance, possible data errors, high bandwidth exploration. For this last one, there are new improvements with DRAM technology, like the Hybrid Memory Cube (HMC) [45], that are promising. Trying to merge the NVMs with new improvements with DRAM, and that would be the hybrid main memory, might be a good idea.

The source codes used in this work can be found in a provided repository[1]. We hope this simulator to be used in future research about the usage of NVMs and we encourage its development and enhancement in even cases that we did not mention here.

[1]Repository URL: `https://github.com/Dragonslair5/NVMExploration`

Bibliography

[1] AXLE project: Advanced analytics for extremely large european databases, 2016. https://axleproject.eu/.

[2] Jung Ho Ahn, Sheng Li, O Seongil, and Norman P Jouppi. Mcsima+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 74–85. IEEE, 2013.

[3] Krste Asanovic and D Patterson. Firebox: A hardware building block for 2020 warehouse-scale computers. In *USENIX FAST*, 2014.

[4] Malcolm P. Atkinson, Peter J. Bailey, Ken J Chisholm, Paul W Cockshott, and Ronald Morrison. An approach to persistent programming. *The computer journal*, 26(4):360–365, 1983.

[5] Malcolm P. Atkinson, Laurent Daynes, Mick J. Jordan, Tony Printezis, and Susan Spence. An orthogonally persistent java. *ACM Sigmod Record*, 25(4):68–75, 1996.

[6] Jean-Loup Baer. *Microprocessor architecture: from simple pipelines to chip multiprocessors*. Cambridge University Press, 2009.

[7] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 41–46, March 2005.

[8] Roberto Bez, Emilio Camerlenghi, Alberto Modelli, and Angelo Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489–502, 2003.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug 2011.

[10] Katherine Bourzac. Has intel created a universal memory technology?[news]. *IEEE Spectrum*, 54(5):9–10, 2017.

[11] Kirk M Bresniker, Sharad Singhal, and R Stanley Williams. Adapting to thrive in a new economy of memory abundance. *Computer*, 48(12):44–53, 2015.

[12] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 52:1–52:12, Nov 2011.

[13] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices*, 49(10):433–452, 2014.

[14] Feng Chen, Michael P Mesnier, and Seungyong Hahn. A protected block device for persistent memory. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pages 1–12. IEEE, 2014.

[15] Youngdon Choi, Ickhyun Song, Mu-Hui Park, Hoeju Chung, Sanghoan Chang, Beakhyoung Cho, Jinyoung Kim, Younghoon Oh, Duckmin Kwon, Jung Sunwoo, et al. A 20nm 1.8 v 8gb pram with 40mb/s program bandwidth. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 46–48. IEEE, 2012.

[16] Danny Cobb and Amber Huffman. Nvm express and the pci express ssd revolution. In *Intel Developer Forum, San Francisco, CA, USA*, 2012.

[17] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 105–118. ACM, 2011.

[18] James J Collins and Carson C Chow. It's a small world. *NATURE*, 393:409, 1998.

[19] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick 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.

[20] Information Storage Industry Consortium et al. 2015–2025 international magnetic tape storage roadmap, 2015.

[21] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[22] Tom Coughlin. The internet of things meets the storage of everything [the art of storage]. *IEEE Consumer Electronics Magazine*, 4(2):118–120, 2015.

[23] Tom Coughlin. Crossing the chasm to new solid-state storage architectures [the art of storage]. *IEEE Consumer Electronics Magazine*, 5(1):133–142, 2016.

[24] Tom Coughlin. How big are your dreams? gauging the size of future content [the art of storage]. *IEEE Consumer Electronics Magazine*, 6(2):108–124, 2017.

[25] Peter J Denning. Virtual memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189, 1970.

[26] Xiangyu Dong, Xiaoxia Wu, Guangyu Sun, Yuan Xie, Helen Li, and Yiran 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.

[27] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, page 15. ACM, 2014.

[28] M. Durlam, B. Craigo, M. DeHerrera, B. N. Engel, G. Grynkewich, B. Huang, J. Janesky, M. Martin, B. Martino, J. Salter, J. M. Slaughter, L. Wise, and S. Tehrani. Toggle mram: A highly-reliable non-volatile memory. In *2007 International Symposium on VLSI Technology, Systems and Applications (VLSI-TSA)*, pages 1–2, April 2007.

[29] Donald Eadie. Edvac drum memory phase system of magnetic recording. *Electrical Engineering*, 72(7):590–595, 1953.

[30] John Presper Eckert. A survey of digital computer memory systems. *Proceedings of the IRE*, 41(10):1393–1406, 1953.

[31] Everspin Technologies. *256K x 16 MRAM Memory, MR2A16A*, 11 2007. Rev. 6.

[32] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond processor-centric operating systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[33] John Fotheringham. Dynamic storage allocation in the atlas computer, including an automatic use of a backing store. *Communications of the ACM*, 4(10):435–436, 1961.

[34] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[35] Bill Gervasi. Dram module market overview. *SimpleTech, JEDEX Shanghai*, 2005.

[36] S. Gill. The diagnosis of mistakes in programmes on the edsac. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 206(1087):538–554, 1951.

[37] S. E. Gluck. The electronic discrete variable computer. *Electrical Engineering*, 72(2):159–162, Feb 1953.

[38] Samuel Greengard. Better memory. *Communications of the ACM*, 59(1):23–25, 2015.

[39] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The fourth-generation intel core processor. *IEEE Micro*, 34(2):6–20, Mar 2014.

[40] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[41] M Hosomi, H Yamagishi, T Yamamoto, K Bessho, Y Higo, K Yamane, H Yamada, M Shoji, H Hachino, C Fukumoto, et al. A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-ram. In *International Electron Devices Meeting*, pages 459–462, 2005.

[42] Cisco Visual Networking Index. The zettabyte era–trends and analysis. *Cisco white paper*, 2013.

[43] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., 2007.

[44] J Janesky, ND Rizzo, Dimitri Houssameddine, R Whig, FB Mancoff, M DeHerrera, JJ Sun, Markus Schneider, HJ Chia, Suhas Aggarwal, et al. Device performance in a fully functional 800mhz ddr3 spin torque magnetic random access memory. In *Memory Workshop (IMW), 2013 5th IEEE International*, pages 17–20. IEEE, 2013.

[45] Joe Jeddeloh and Brent Keeth. Hybrid memory cube new dram architecture increases density and performance. In *VLSI Technology (VLSIT), 2012 Symposium on*, pages 87–88. IEEE, 2012.

[46] Ju-Young Jung and Sangyeun Cho. Memorage: Emerging persistent ram based malleable main memory and storage architecture. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 115–126. ACM, 2013.

[47] Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. Reducing the cost of persistence for nonvolatile heaps in end user devices. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 512–523. IEEE, 2014.

[48] A. Kawahara, R. Azuma, Y. Ikeda, K. Kawai, Y. Katoh, K. Tanabe, T. Nakamura, Y. Sumimoto, N. Yamada, N. Nakai, S. Sakamoto, Y. Hayakawa, K. Tsuji, S. Yoneda, A. Himeno, K. i. Origasa, K. Shimakawa, T. Takagi, T. Mikawa, and K. Aono. An 8mb multi-layered cross-point reram macro with 443mb/s write throughput. In *2012 IEEE International Solid-State Circuits Conference*, pages 432–434, Feb 2012.

[49] Kimberly Keeton. The machine: An architecture for memory-centric computing. In *Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 2015.

[50] Karl F Kempf. *Electronic Computers within the Ordnance Corps*. Aberdeen Proving Ground, 1961.

[51] Tom Kilburn, Dai BG Edwards, Michael J Lanigan, and Frank H Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, (2):223–235, 1962.

[52] Nasser A Kurd, Subramani Bhamidipati, Christopher Mozak, Jeffrey L Miller, Timothy M Wilson, Mahadev Nemani, and Muntaquim Chowdhury. Westmere: A family of 32nm ia processors. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 96–97. IEEE, 2010.

[53] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 38–49. IEEE, 2013.

[54] Shuangchen Li, Ping Chi, Jishen Zhao, Kwang-Ting Cheng, and Yuan Xie. Leveraging nonvolatility for architecture design with emerging nvm. In *Non-Volatile Memory System and Applications Symposium (NVMSA), 2015 IEEE*, pages 1–5. IEEE, 2015.

[55] YiChing Lin, SH Kang, YJ Wang, K Lee, X Zhu, WC Chen, X Li, WN Hsu, YC Kao, MT Liu, et al. 45nm low power cmos logic compatible embedded stt mram utilizing a reverse-connection 1t/1mtj cell. In *Electron Devices Meeting (IEDM), 2009 IEEE International*, pages 1–4. IEEE, 2009.

[56] Michael J Litzkow, Miron Livny, and Matt W Mutka. Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111. IEEE, 1988.

[57] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. Nvm duet: Unified working memory and persistent store architecture. In *ACM SIGPLAN Notices*, volume 49, pages 455–470. ACM, 2014.

[58] Tz-yi Liu, Tian Hong Yan, Roy Scheuerlein, Yingchang Chen, Jung Keun Lee, Ganesh Balakrishnan, Gordon Yee, Haijun Zhang, Alex Yap, Jun Ouyang, et al. A 130.7-2-layer 32-gb reram memory device in 24-nm technology. *Solid-State Circuits, IEEE Journal of*, 49(1):140–153, 2014.

[59] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[60] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[61] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, dec 1995.

[62] Justin Meza, Yixin Luo, Samira Khan, Jishen Zhao, Yuan Xie, and Onur Mutlu. A case for efficient hardware/software cooperative management of storage and memory. 2013.

[63] Donald R Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.

[64] Ravi Nair. Evolution of memory architecture. *Proceedings of the IEEE*, 103(8):1331–1345, 2015.

[65] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. *ACM SIGARCH Computer Architecture News*, 40(1):401–410, 2012.

[66] John von Newmann. First draft of a report on the edvac. 1945.

[67] Anurag Nigam, Clinton W Smullen IV, Vidyabhushan Mohan, Eugene Chen, Sudhanva Gurumurthi, and Mircea R Stan. Delivering on the promise of universal memory for spin-transfer torque ram (stt-ram). In *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, pages 121–126. IEEE Press, 2011.

[68] Shuichi Oikawa. Integrating memory management with a file system on a non-volatile main memory system. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1589–1594. ACM, 2013.

[69] Mauricio G Palma, Emilio Francesquini, and Rodolfo Azevedo. Simulação de arquiteturas de hardware com memórias não-voláteis. In *Anais do XVII Simpósio em Sistemas Computacionais de Alto desempenho (WSCAD)*, pages 264–275, 2016.

[70] Mauricio G Palma, Emilio Francesquini, and Rodolfo Azevedo. Simulação e avaliação de soluções de software para arquiteturas com memórias não-voláteis. In *Anais da 7a Escola Regional de Alto Desempenho de São Paulo (ERAD-SP)*, pages 113–116, 2016.

[71] Avadh Patel, Furat Afram, and Kanad Ghose. Marss-x86: A qemu-based microarchitectural and systems simulator for x86 multicore processors. In *1st International Qemu Users' Forum*, pages 29–30, 2011.

[72] David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[73] Steven Pelley, Peter M Chen, and Thomas F Wenisch. Memory persistency. *ACM SIGARCH Computer Architecture News*, 42(3):265–276, 2014.

[74] Matthew Poremba, Tao Zhang, and Yuan Xie. Nvmain 2.0: A user-friendly memory simulator to model (non-) volatile memory systems. *IEEE Computer Architecture Letters*, 14(2):140–143, 2015.

[75] A Proctor. Non-volatile memory & its use in enterprise applications. *Viking Technology Understanding Non-volatile Memory Technology Whitepaper*, pages 1–8, 2012.

[76] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutiu. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. In *Microarchitecture (MICRO), 2015 48th Annual IEEE/ACM International Symposium on*, pages 672–685. IEEE, 2015.

[77] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 10(1):16–19, 2011.

[78] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. *SIGARCH Comput. Archit. News*, 41(3):475–486, jun 2013.

[79] Joachim W Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems (TODS)*, 2(3):247–261, 1977.

[80] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, Todd C Mowry, and Trishul Chilimbi. Page overlays: An enhanced virtual memory framework to enable fine-grained memory management. *ACM SIGARCH Computer Architecture News*, 43(3):79–91, 2016.

[81] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[82] R. L. Snyder. The input-output system of the edvac. *Transactions of the American Institute of Electrical Engineers*, 70(1):507–509, July 1951.

[83] Andrew S Tanenbaum. *Modern operating systems*. Pearson Education, 2009.

[84] Avadis Tevanian, Richard F Rashid, Michael Young, David B Golub, Mary R Thompson, William J Bolosky, and Richard Sanzi. A unix interface for shared memory and memory mapped files under mach. In *USENIX Summer*, pages 53–68, 1987.

[85] Dan Vesset, CW Olofson, A Nadkarni, A Zaidi, B McDonough, D Schubmehl, S Bond, Sh Kusachi, Q Li, and Ph Carnelley. Idc futurescape: Worldwide big data and analytics 2016 predictions. *International Data Corporation*, 2015.

[86] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. *ACM SIGPLAN Notices*, 46(3):91–104, 2011.

[87] Maurice V Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, (2):270–271, 1965.

[88] Maurice V Wilkes. The memory gap and the future of high performance memories. *ACM SIGARCH Computer Architecture News*, 29(1):2–7, 2001.

[89] Stuart A Wolf, Jiwei Lu, Mircea R Stan, Eugene Chen, and Daryl M Treger. The promise of nanomagnetics and spintronics for future logic and universal memory. *Proceedings of the IEEE*, 98(12):2155–2168, 2010.

[90] HS Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

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

[92] Xiaojian Wu and AL Reddy. Scmfs: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 39. ACM, 2011.

[93] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[94] Matthias Wuttig. Phase-change materials: towards a universal memory? *Nature materials*, 4(4):265–266, 2005.

[95] Wei Xu, Hongbin Sun, Xiaobin Wang, Yiran Chen, and Tong Zhang. Design of last-level on-chip cache using spin-torque transfer ram (stt ram). *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(3):483–493, 2011.

[96] J.J. Yi and D.J. Lilja. Simulation of computer architectures: simulators, benchmarks, methodologies, and recommendations. *Computers, IEEE Transactions on*, 55(3):268–280, March 2006.

[97] Shen Yin and Okyay Kaynak. Big data for modern industry: challenges and trends [point of view]. *Proceedings of the IEEE*, 103(2):143–146, 2015.

[98] Hung-Chang Yu, Kai-Chun Lin, Ku-Feng Lin, Chin-Yi Huang, Yu-Der Chih, Tong-Chern Ong, Joana Chang, Sriraam Natarajan, and Luan C Tran. Cycling endurance optimization scheme for 1mb stt-mram in 40nm technology. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2013 IEEE International*, pages 224–225. IEEE, 2013.

[99] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 421–432. ACM, 2013.

[100] Jishen Zhao, Cong Xu, Ping Chi, and Yuan Xie. Memory and storage system design with nonvolatile memory technologies. *IPSJ Transactions on System LSI Design Methodology*, 8(0):2–11, 2015.