UNIVERSIDADE FEDERAL DE PELOTAS
Centro de Desenvolvimento Tecnológico
Programa de Pós-Graduação em Computação

Dissertação

An Evaluation of Memory Controllers for Non-Volatile Memories

Giovane de Oliveira Torres

Pelotas, 2018

Giovane de Oliveira Torres

An Evaluation of Memory Controllers for Non-Volatile Memories

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Mestre em Ciência da Computação

Advisor: Prof. Dr. Maurício Lima Pilla
Coadvisor: Prof. Dr. Laércio Lima Pilla

Pelotas, 2018

Universidade Federal de Pelotas / Sistema de Bibliotecas
Catalogação na Publicação

T693a Torres, Giovane de Oliveira

An evaluation of memory controllers for non-volatile memories / Giovane de Oliveira Torres ; Maurício Lima Pilla, orientador ; Laércio Lima Pilla, coorientador. — Pelotas, 2018.

77 f. : il.

Dissertação (Mestrado) — Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, 2018.

1. Memory hiearchies. 2. Computer architectures. 3. Non-volatile memories. I. Pilla, Maurício Lima, orient. II. Pilla, Laércio Lima, coorient. III. Título.

CDD : 005

Elaborada por Maria Inez Figueiredo Figas Machado CRB: 10/1612

AGRADECIMENTOS

É nesta parte da dissertação que preciso agradecer às pessoas que tornaram possível, direta ou indiretamente a realização deste trabalho, pelos mais diferentes motivos. Por mais que eu acredite que seja impossível conseguir resumir em algumas palavras meus mais sinceros agradecimentos, acho importante ainda deixar registradas algumas palavras aqui neste espaço.

O primeiro agradecimento é para a família, especialmente aos meus pais, pelo suporte durante o tempo de mestrado. Ao meu pai, o agradecimento por tudo – não há outra palavra melhor para agradecer, por que senão seria uma longa lista de agradecimentos! À minha mãe (*in memoriam*), que durante nosso tempo de convivência, me deixou diversos ensinamentos que procuro sempre carregar pra minha vida.

Um agradecimento ao meu orientador, Prof. Pilla, que me teve como bolsista durante o tempo do mestrado (e antes já haviam sido quatro na graduação), com quem tive a oportunidade de aprender diversos assuntos. Outro agradecimento ao meu coorientador, Prof. Laércio, pelas diversas ajudas nas várias etapas durante o mestrado. Mais um agradecimento aos demais professores do curso, por ensinamentos, conversas e outras coisas.

Muitíssimo obrigado ao LUPS, o laboratório que continuei meus estudos para fazer o mestrado, que embora às vezes tenha sido um lugar com diversos problemas para trabalhar, me propiciou boas experiências. Indepentente disso, foi lá que desenvolvi o trabalho escrito nesta dissertação; conheci pessoas incríveis; fiz amizade com algumas poucas pessoas, que me ajudaram demais, além de proporcionar boas histórias, risadas e outras coisas, tanto em atividades dentro quanto fora da faculdade – e estes, vocês sabem quem são, não é mesmo, seus aleatórios?

Mais um agradecimento é necessário, à todas as pessoas que tive a oportunidade de conhecer neste período. Cada um, mesmo não sabendo, me ensinou alguma coisa diferente. A quem eu conheci e foi construída uma amizade neste período, um outro muito obrigado: por ajudas, conversas, e outras tantas coisas feitas. E aos amigos que eu já tinha antes de começar o mestrado, e que ainda continuam sendo meus amigos, vai um grande agradecimento, por todos os tipos de atividades feitas que, sem dúvidas, foram extremamente importantes para mim.

É vivendo que se aprende.
— FERNANDO BENINI

ABSTRACT

TORRES, Giovane de Oliveira. **An Evaluation of Memory Controllers for Non-Volatile Memories**. 2018. 77 f. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2018.

Many demands which include performance and energy consumption are present in current computational systems. In this context, actual memory technologies are critical components which affect directly both performance and energy cost of a system. Thus, memory needs improvements since they could be reaching its scalability limit. One of the alternatives to improve memory subsystems is the use of non-volatile memories (NVMs). These memories have overall low energy consumption and better scalability when comparing with volatile memories. However, NVMs still have issues which need to be overcome in order to allow large-scale use. Those issues include costly write operations (both on latency and energy) and lower material endurance if compared to current memory technologies. Parallel to that, one difficulty to adopt NVMs as main memories in computational systems is related to providing a proper scheduling of memory operations – as it should cope with the particular characteristics of NVMs. With that issues in mind, this work presents a memory controller implementation – in addition to that, a runtime analysis of memory operations in NVM-based systems is performed. To implement a memory controller, we use Gem5 and NVMain simulators, since their combination could reach runtime evaluations that this work aims. Then, the implemented controller was tested by running applications from MediaBench and MiBench benchmark set. With that, the controller was analyzed under different configurations. Tests were performed using the three most well-known and studied NVMs (PCRAM – Phase Change Random Access Memory, RRAM – Resistive Random Access Memory and STT-RAM – Spin Transfer Torque Random Access Memory). The main observations that could be highlighted in this work were: (i) while running the benchmarks in isolation, the time spent in serving memory requests is very low, not surpassing 2% of the total execution time of any application tested, (ii) when running the memory controller under different NVM technologies, latencies of read and write operations in overall were mostly impacted by the different type of memories used, (iii) when using queues with variable sizes to hold memory requests, it made negligible difference in overall performance, due to applications having small busy periods, i.e., generating and serving a small number of memory requests in detriment of other operations. Lastly, a comparison between the implemented memory controller and NVMain default memory controllers was performed, which pointed out that in the majority of the studied cases, the proposed

memory controller may need extra techniques to get better performance.

Keywords: memory hiearchies; computer architectures; non-volatile memories

RESUMO

TORRES, Giovane de Oliveira. **Uma Avaliação de Controladoras de Memória para Memórias não Voláteis**. 2018. 77 f. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2018.

Existem atualmente diversas demandas nas questões de desempenho e consumo energético em sistemas computacionais. Dentro deste contexto, as tecnologias de memórias usadas atualmente são consideradas componentes críticos que afetam diretamente tanto desempenho quanto consumo energético dentro em um sistema. Com isto, existe a necessidade de que memórias apresentem melhorias, já que estas talvez estejam chegando no seu limite de escalabilidade. Uma das alternativas para melhorar o subsistema de memória é o uso de memórias não voláteis (NVMs). Estas memórias têm em geral baixo consumo energético e melhor escalabilidade ao comparar com tecnologias voláteis. Porém, NVMs apresentam problemas que precisam ser superados para que estas possam ser utilizadas em larga escala. Estes problemas remetem ao alto custo de operações de escrita em memória tanto em latência quanto em consumo energético, além de durabilidade do material usado nas NVMs sendo inferior se comparado à tecnologias de memória atuais. Paralelo a isto, uma das dificuldades para adotar uma NVM como memória principal em sistemas computacionais também inclui dificuldades em propor um escalonamento de operações de memórias, já que torna-se necessário lidar com as características de NVMs. Com estes problemas em mente, esse trabalho apresenta uma implementação de controladora de memória, juntamente com uma análise em tempo de execução em sistemas baseados em NVMs. Para implementação de uma controladora de memória, são utilizados os simuladores Gem5 e NVMain, visto que a combinação destas atende ao desejo do trabalho de efetuar avaliações em tempo de execução. Com isto, a controladora implementada passou por testes com aplicações do conjunto de *benchmarks* MediaBench e MiBench. Assim, a controladora foi analisada sobre diferentes configurações. Testes foram feitos usando as três tecnologias de NVMs mais conhecidas e estuadas (PCRAM – *Phase Change Random Access Memory*, RRAM – *Resistive Random Access Memory* e STT-RAM – *Spin Transfer Torque Random Access Memory*). As principais observações feitas por este trabalho foram: (i) executando os *benchmarks* isoladamente, o tempo gasto em atender requsições de memória foi baixo, não superando 2% do tempo total de execução de qualquer aplicação, (ii) ao executar a controladora de memória com diferentes NVMs, as latências de operações de memória foram impactadas fortemente pelo tipo de tecnologia utilizada, (iii) ao utilizar diferen-

tes tamanhos de filas para atender requisições de memórias, o impacto causado por esta variação foi considerado desprezível, já que as aplicações executadas geraram e atenderam poucas operações de memórias em detrimento de outras operações. Por fim, uma análise de desempenho da controladora implementada com as controladoras de memória fornecidas pelo NVMain, sendo observado que na maioria dos casos a controladora proposta e avaliada talvez necessite de inclusão de outras técnicas para a extração de melhor desempenho.

Palavras-Chave: hierarquias de memória; arquitetura de computadores; memórias não voĺáteis

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

BL	Base Line
DRAM	Dynamic Random Access Memory
FCFS	First Come First Served
FRFCFS	First Ready First Come First Served
FRFCFS-WQF	First Ready First Come First Served with Write Queue
FERAM	Ferroelectric Random Access Memory
HRS	High Resistance State
LRS	Low Resistance State
MIM	Metal-Insulator-Metal
MTJ	Magnetic Tunnel Junction
MRAM	Magnetic Random Access Memory
NVM	Non-Volatile Memory
PCM	Pulse-code modulation
PCRAM	Phase Change Random Access Memory
RRAM	Resistive Random Access Memory
SL	Source Line
SRAM	Static Random Access Memory
STT-RAM	Spin Torque Transfer Random Access Memory
WL	Word Line

CONTENTS

1 INTRODUCTION

Nowadays, there is a constant need for more computational power. In this context, it is always important to find technological innovations that allow upgrades in both performance and energy consumption. One of the areas that is the target of those updates is the current memory technologies, due to multiple reasons: (i) memories are reaching their scalability limit (POREMBA; XIE, 2012; YOUNG; NAIR; QURESHI, 2015; OUKID; KETTLER; WILLHALM, 2017), (ii) due to the way memories are constructed, leakage current is becoming more of a problem (WANG; ALZATE; AMIRI, 2013; LI et al., 2015; AWAD et al., 2016) and (iii) memory is considered to be a critical component in computational systems (PEREZ; DE ROSE, 2010; ZOU et al., 2015; POURSHIRAZI; ZHU, 2016). With that considered, in order to allow progress in computational systems, it is essential that memories also get improvements.

One possibility to perform upgrades in memories lies in the replacement of current memory technologies used – those being DRAM (Dynamic Random Access Memory) and SRAM (Static Random Access Memory) – by non-volatile memories (NVMs). These memories can provide low energy consumption, as well as better scalability and higher density by memory cell (MEENA et al., 2014; YOUNG; NAIR; QURESHI, 2015; ZHAO et al., 2015). The non-volatility feature of memory allows data to be retained for a long period of time, which varies accordingly to the material used in a memory cell. On the other hand, this does not occur in volatile memories, where refresh operations are necessary to retain data. When performing continuous refreshes over time, the cost to maintain data in memory cells grows. However, NVMs also have issues that must be overcome so that they could be used in large scale. One of the biggest problems are related to NVM endurance: Due to the material used to make NVM cells, they have lower endurance if compared with volatile technologies. Also, both energy consumption and time spent to perform write operations in NVM cells are considered highly costly (MEENA et al., 2014).

In this work, three NVMs were studied: PCRAM (Phase Change Random Access Memory), STT-RAM (Spin Transfer Torque Random Access Memory) and RRAM (Resistive Random Access Memory). These specific memories were picked according

to a study done on NVMs which analyzed that most NVM related work discusses these three memories (MITTAL; VETTER, 2016). In addition to that, these NVMs have the potential for building high-density and power-efficient memory systems, thanks to enhanced scalability and non-volatility properties (YOUNG; NAIR; QURESHI, 2015). NVMs are applicable for a multitude of different computational areas, showing promising results. These include: embedded systems (WANG; ALZATE; AMIRI, 2013; CHANG et al., 2014), real-time systems (DASARI; NELIS; MOSSE, 2013; ZHANG et al., 2013), systems with NVMs as main memory (KÜLTÜRSAY et al., 2013; CHI; LEE; XIE, 2014; ZHANG et al., 2016), hybrid memory architectures (WANG et al., 2014), among others.

The studied NVMs can be inserted at different parts of a memory architecture. PCRAM is considered to be a future candidate for replacing DRAM as main memory (LI et al., 2014; ARJOMAND et al., 2017), plus large PCRAM chips are already available (CHUNG et al., 2011; CHOI et al., 2012). On the other hand, RRAM is tested mainly in cache memories (KOTRA et al., 2016; LI et al., 2017), and STT-RAM has most studies focused also in cache memories (YAZDANSHENAS et al., 2014; KIM; KIM; LEE, 2017). When developing large chips, these memories also present issues regarding reliability, (KANG et al., 2015; MUTLU; SUBRAMANIAN, 2015). Nevertheless, there is work showing that using both STT-RAM and RRAM as main memories show promising results (KÜLTÜRSAY et al., 2013; CHI et al., 2016).

One of the key challenges to improve performance of memories resides in attending read and write instructions in the best instance of time possible. Memory scheduling tends to be a complex problem, due to having to cope with multiple issues (MARTINEZ; IPEK, 2009; KIM et al., 2010; GOOSSENS et al., 2016). In some categories of systems, such as real-time systems, a set of memory operations needs to complete within a fixed deadline (FUJITA, 2014), however that may come with more costs with hardware mechanisms, which leads to more energy consumption. Hence the use of an NVM as a main memory in computational systems is seen as an alternative to improve energy consumption. When dealing with systems with NVM as main memories, a reasonable number of memory operations may not accomplish their deadlines, which can result in unacceptable delays in execution times of memory operations (DASARI; NELIS; MOSSE, 2013). With these exposed problems, memory controller scheduling policies may need to mitigate these issues (ZHOU et al., 2011; DASARI; NELIS; MOSSE, 2013; HU et al., 2014), by taking into account the natural issues found in NVMs – those being asymmetry of read and write operations, plus limited write endurance.

Considering the importance of memory operation scheduling and the use of NVMs, this work explains one of its main contributions: The implementation of a memory controller aware of NVM issues. This controller was based on the work of Dasari, Nelis and Mosse (2013) – in that work, a memory controller was proposed and a static anal-

ysis with traces of already-run benchmarks was performed. Here, the implementation extends the idea to implement the memory controller in a general-purpose simulator, so that it can evaluate the potential performance of memory controller during runtime. In order to do that, two tools were used: NVMain and Gem5, where the first one simulates NVMs, while the second one is a general-purpose simulator. These tools were chosen since their combination allows the runtime analysis this work aims. The main contribution of this work is the behavioral analysis of a different memory controller implementation, plus an evaluation its potential performance.

To provide results for this work, applications were chosen to run over the modified simulator. For this, the sets of benchmarks MediaBench and MiBench were chosen. The first result presented on this work tested the memory controller over three distinct NVMs as main memories (PCRAM, STT-RAM and RRAM), where it was seen that write memory latencies had expected results accordingly to each NVM used. On the other hand, read latencies suffered from great variations, according to the NVM and benchmark simulated. Then, both periods of benchmarks were analyzed – these periods were based on the concept of busy and idle states of the implemented memory controller, where the first one is designed to attend memory requests, while the other one waits for memory requests to come. Performed analysis show that the mean of time spent in busy periods was small, not surpassing 2% of the total execution time. Additionally, the use of queues in the implemented memory controller was analyzed, showing that queues were in overall underused, accompanying the low time spent in busy periods. On average, applications do not keep more than one request in queue when the memory controller is active, which was probably a consequence of benchmarks running in isolation – hence, this not allow the generation of a large number of memory requests. Even though this occurred, it was possible to detect benchmarks which generated bursts of memory operations, which could potentially put more pressure under the memory controller buffers in a full-system simulation.

The implemented controller was compared with already-established controllers provided by NVMain – results were presented evaluating performance on latencies of read/write operations. Regarding write memory operation latencies, when simulating with PCRAM as main memory, the custom memory controller matched these latencies of NVMain memory controllers. On the other hand, in RRAM and STT-RAM simulations write latencies tended to be worse if comparing with already-implemented memory controllers. When evaluating read latencies, on overall the results presented many variations, depending directly of the executed benchmark. This work then concludes that (i) the proposed controller may need improvements or consider other memory scheduling techniques so that better performance can be extracted and (ii) the controller may be used in hybrid memory systems in order to extract the best features each technology of memory can offer.

This work is organized as follows: Chapter 2 makes a brief study of current NVM technologies which are considered by this work (PCRAM, STT-RAM and RRAM), depicting their individual features. Chapter 3 explains the importance of memory scheduling, linking with NVM-based systems. Also, this Chapter exposes a schema for a memory controller, where the implementation is based. Chapter 4 summarizes the main contribution of this work, explaining how it was done, exposing the tools used and how the memory controller implementation was performed. Chapter 5 exhibits the main results reached by this work, and lastly, Chapter 6 explains the main conclusions achieved, also presenting potential future work.

2 NON-VOLATILE MEMORIES

NVMs are memory technologies that feature the potential to retain data for long periods without needing constant refresh operations as seen in volatile memories. NVM technologies work due to the materials used as memory cells because they have the possibility to change their state by applying an electric current.

These technologies hold potential to both consume very low power and provide much higher density than current volatile technologies such as DRAM and SRAM. However, NVMs have some issues that need to be overcome, which include poor write endurance (being several orders of magnitude lower than conventional memories) and high latency and energy costs when performing write operations (MITTAL; VETTER, 2016).

The NVM technologies studied for this work are presented in the following Sections: STT-RAM in Section 2.1, PCRAM in Section 2.2 and RRAM in Section 2.3. Then, Section 2.4 makes a short discussion of the three non-volatile technologies. Lastly, Section 2.5 concludes this Chapter.

2.1 STT-RAM

STT-RAM (Spin Transfer Torque Random Access Memory) is an NVM that is listed to be a future candidate to replace SRAM (THOMAS et al., 2014; ZHAN et al., 2016), which is mostly used as cache memories, even though some related work shows that it could be used as main memory (KÜLTÜRSAY et al., 2013; EWAIS et al., 2016). STT-RAM is considered to be an improvement over MRAM (Magnetic Random Access Memory), as STT-RAMs exert the base platform established by MRAMs to enable highly scalable memory, smaller cell sizes and better read and write latencies (MEENA et al., 2014).

An STT-RAM cell is composed of two magnetic storage devices that are different from each other. The first device has fixed magnetic orientation, while the second one holds the possibility to change its orientation. Both devices are also called ferromagnetic layers, where the first one is called **reference layer** and the second one is defined

as **free layer** (ZHOU et al., 2009). A third device is also placed between these two layers, which is called tunnel barrier layer. The grouping of the three layers is defined as the magnetic tunnel junction (MTJ), which is the key element to store information. The effect known as **spin transfer torque** occurs when a relatively strong electric current is applied in the MTJ that can flip the free layer magnetic orientation, which happens when an electron with a misaligned spin passes into a magnetized material – that mismatch gives rise to a torque between the electron and the magnet (COALMON, 2009). Figure 1 depicts an example of a basic STT-RAM cell structure. Besides the MTJ, the STT-RAM cell includes a bit line (BL) and a source line (SL). Depending of the current applied over these lines, it performs different memory operations. The word line (WL) is used to connect multiple memory cells, allowing to read an entire row of cells. Lastly, the transistor placed below the MTJ is used to allow the memory operations in a STT-RAM cell.

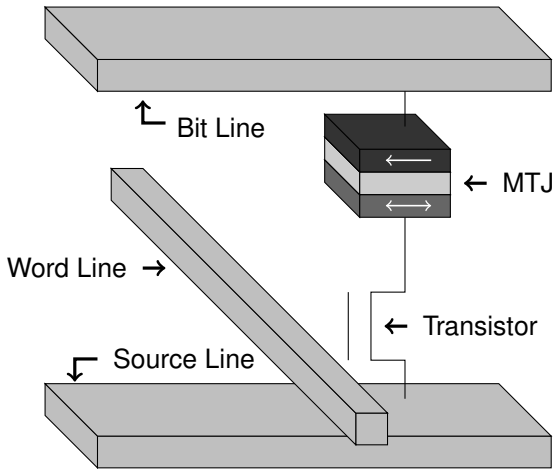


Figure 1 – Basic STT-RAM cell structure (MEENA et al., 2014)

In order to represent a bit value (logical 0 and logical 1), the magnetization difference of the reference layer and free layer is used – which is translated into a resistance difference in the MTJ. When the magnetic field of the free layer and reference layer is parallel, the MTJ resistance is low and can be interpreted as the logical zero. However, when these two layers have opposite magnetic orientations the MTJ resistance becomes high, thus the logical one can be represented. Figure 2 shows a conceptual view of the MTJ and its possible magnetic orientations.

In order to do operations in STT-RAM cells, an NMOS is connected to the WL to allow selection of a row of cells. Thereafter, a voltage is applied between the bit line BL and source line SL. Depending of the voltage applied, a different operation is performed (ZHOU et al., 2009):

- Read: A small voltage is applied between BL and SL. The amount of current that flows through the cell depends of the resistance of MTJ, which may be low or

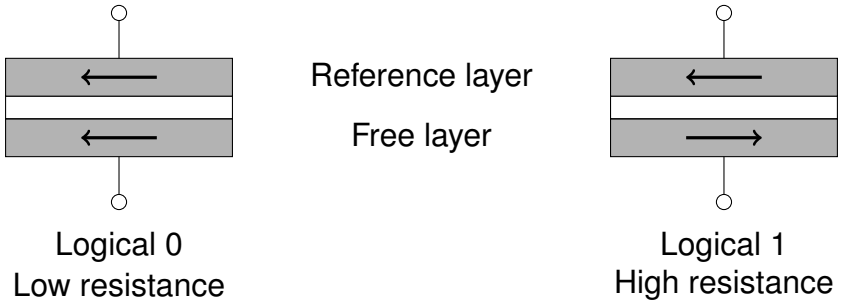


Figure 2 – Conceptual view of the MTJ structure (ZHOU et al., 2009)

high depending of the magnetic orientation of the free and reference layers. The resistance is sensed by an amplifier to output the value stored in the cell;

- Write (Reset): A larger, positive voltage is applied between SL and BL, creating a current flow from SL to BL;
- Write (Set): A larger but negative voltage is applied between SL and BL, creating a current flow in the opposite direction.

The advantages of using STT-RAM are the same as described in general NVMs: low power consumption and smaller cell sizes. When comparing STT-RAM to a conventional technology such as SRAM, it could certainly achieve 4 times greater density (ZHOU et al., 2009; ZHANG et al., 2015), but it is possible that it can reach even higher densities (KANG, 2010; YAKOPCIC; HASAN; TAHA, 2015).

However, STT-RAMs present issues regarding the asymmetry in latencies and energy consumption in read and write operations in memory cells. When performing a read operation, both latency and energy consumption are comparable to a SRAM cell read. That does not occur in a write operation, where the latency and energy consumption are much higher when comparing to the same SRAM cell (KÜLTÜRSAY et al., 2013).

2.2 PCRAM

Phase-Change Random Access Memory (PCRAM) is other type of NVM that is also considered as a technology for future memories (OIKE et al., 2015; BURR et al., 2016), which is seen as an alternative to the main memory of a computer architecture that offers better density per cell if compared to conventional memories (DU et al., 2013);

PCRAM cells are composed of two electrodes separated by a resistor, plus a phase change material – which is commonly a chalcogenide (RAOUX et al., 2014; PIROVANO, 2018). The $Ge_2Sb_2Te_5$ composite (Germanium-Antimony-Tellurium) is the most commonly used (MENG et al., 2016; BURR et al., 2016), although there is related

work that study various doping elements in the Sb_2Te_5 composite that can use titanium, aluminum, gallium, among other metals.

The phase change material layer is responsible for information storage in PCRAM, and it can be in two different states: **amorphous** and **crystalline**. By using these two distinct states it is possible to represent the two logical values of a bit, because each state holds a different resistance which differentiates from the other one. The amorphous state has high resistance, and it is used to represent logical 0, while the crystalline one has low resistance and it represents logical 1. Figure 3 depicts an example of a PCM cell. A layer of phase change material is placed between both electrodes. The lower electrode has a heat-resistant element (resistor) which makes contact with the chalcogenide layer.

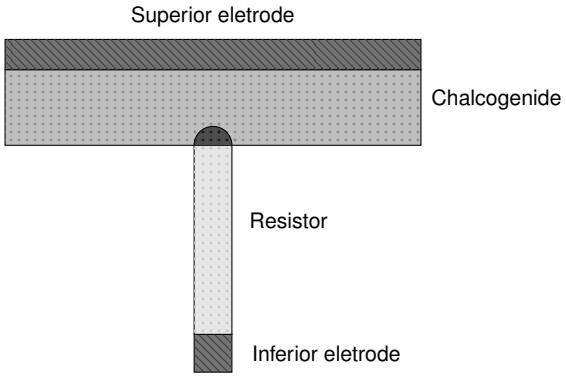


Figure 3 – Example of a basic PCM cell structure (NUMONYX, 2007)

The action of applying a current in the junction of resistor and chalcogenide denotes that an operation over a memory cell is going to be performed. Both intensity and time of an electric current define which memory operation will occur. This is shown in Figure 4, which summarizes the time and current intensity necessary to do read and write (set and reset) operations.

Each memory operation has well-defined parameters of time and current intensities. Below, every operation is explained:

- Read: A current of small intensity and time is applied, so that the phase change material is not changed in process. Then the material resistance is measured, which is different according to the current cell state;
- Reset: In this case a short time but intense current must be applied. Due to abrupt interruption of the current, the phase change material resistance increases – and by quickly stopping heat generation, the material turns into an amorphous state;
- Set: A current with longer time and moderate intensity is applied. By reducing the current, the material will reduce its resistance as well. The phase change material will also cool down slowly, turning into a crystalline state.

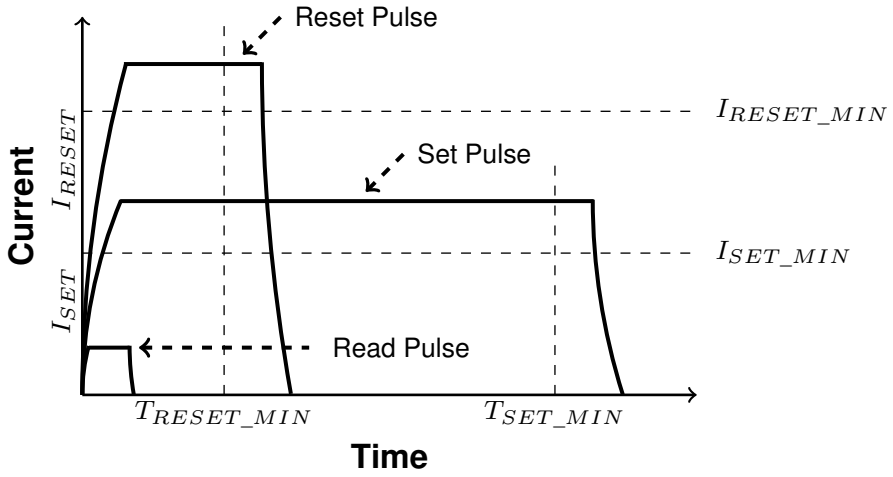


Figure 4 – Time and current intensity necessary for each memory operation in a PCM cell (WANG; WU, 2009)

Advantages found in PCRAMs can also be seen in those offered by NVMs, which include high scalability and lower energy consumption. PCRAM is also affected by the main problems found on non-volatile technologies, thus performing a write operation in a memory cell is costly on both energy consumption and time. As seen in Figure 4, the maximum time spent in a memory operation is defined by the set operation – and energy consumption peak occurs when a reset operation is done. Constant write operations in PCRAMs tend to deteriorate the phase change material more quickly if compared with conventional technologies – the write endurance of PCRAMs is orders of magnitude lower than DRAMs and SRAMs.

2.3 RRAM

The Resistive Random Access Memory (RRAM) is pointed to be another future candidate for general-purpose memory due to some of its features. Those include (i) excellent potential to miniaturization – its cell size can reach $4F^2$ (where F stands for line width), (ii) possibility to replace DRAM as main memory, (iii) and natural integration to 3D memories (ZHOU; KIM; LU, 2014; PAN et al., 2014; SONG et al., 2017).

A RRAM cell is usually composed of a device with an isolating or semiconductor material layer placed between two conductive metals – this structure is called Metal-Insulator-Metal (MIM). Figure 5 shows an example of a RRAM cell. The middle layer of an MIM is used as storage of memory information, while top and bottom layers are used as electrodes.

The middle layer of an MIM is commonly made of an oxide due to its natural feature of undergoing resistance change effects. In particular, metal oxides are the most studied alloys in related work (MEENA et al., 2014; IELMINI, 2016). On the other hand, the materials of electrode layers studied are quite varied, since its range can start

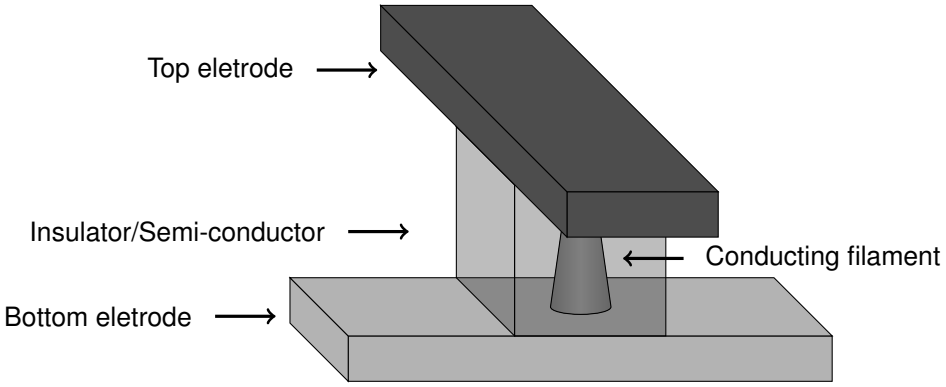


Figure 5 – Basic structure of an RRAM cell (PAN et al., 2014)

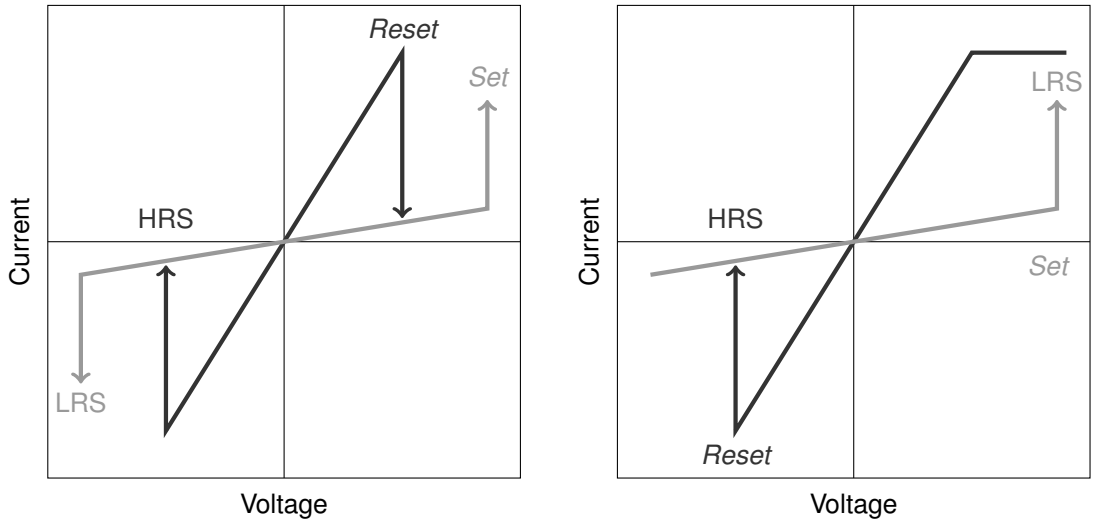
from pure elements that could be used up to complex alloys, and even some kinds of oxides (PAN et al., 2014).

In order to perform memory operations over RRAM cells, current must be applied in one of the electrodes. Similar to what occurs in a PCRAM, current intensity is the key to determine what operation will be run.

- Read: A small intensity current is applied, which needs to be little enough so it does not change the state of insulator layer. With that current, it is possible to get the material resistance and thus getting its value stored;
- Set: A current with more intensity than one used to read from a cell is applied – which needs to be intense enough to reduce to turn the insulator layer resistance into a low resistance state (LRS);
- Reset: In this case, the current needs to be sufficiently intense to increase resistance to switch the insulator layer resistance to a high resistance state (HRS).

Another important aspect that must be analyzed on metal-oxide RRAMs relates to switching modes. These can be split in two general modes called **unipolar** and **bipolar**. Figure 6 shows a schema explaining how write operations are performed depending of switching modes. Unipolar switching means the switching direction depends on only the amplitude of an applied voltage, and it does not depend on the polarity. Thus, both write operations can occur at the same polarity. On the other hand, bipolar switching means the switching direction depends of the polarity of an applied voltage, which means the set operation can only occur at one polarity, and reset operation must occur at the reverse polarity.

Even though RRAMs are candidates to be largely used future memories, there are several issues that need to be addressed. As seen in previous NVMs, high latency and energy consumption of write operations and low material endurance are problems seen in RRAM cells. Studies on increasing reliability and maintenance of the materials



(a) Unipolar (b) Bipolar

Figure 6 – Relation between current and voltage on different switching modes of RRAM (WONG et al., 2012)

used in RRAMs are essential to their future (MEENA et al., 2014). One of the main issues to adopt the use of commercial RRAMs is the lack of a thorough understanding of the switching mechanisms: the effect of switching states is hard to locate (because of the tiny device) and it is of random nature (PAN et al., 2014). Also, write endurance problem tends to be one of the long-term challenges to allow RRAMs to be used as large-scale memories (AKINAGA; SHIMA, 2012).

2.4 Discussion

With all of the features presented by each NVM, they are promising candidates to replace actual technologies. The features presented by NVMs allow them to be used in many different situations, which is seen in the diversity of related work (MITTAL; VETTER; LI, 2015; MITTAL; VETTER, 2016). The studied memories on this work are composed of different materials implies in unequal characteristics and different ways of performing read and write operations in each NVM. Table 1 shows electrical and physical features of both presented NVMs and actual volatile technologies.

Memory cell sizes are usually measured by the F unit, where it represents the minimum feature size accessible by lithography (IELMINI, 2016). Cell sizes are also measured considering the use of single-level cells (SLC) – which means that each memory cell can represent two different states: logical zero (false) and logical one (true). This description is necessary because NVMs have the possibility to store more than simply two different states in a single memory cell, which can be achieved by using multi-level cells (MLC). This feature greatly increases the storage capacity of memories with the penalty of increased write and read latencies (DONG; XIE, 2011; KHWA et al.,

Table 1 – Comparing NVM and current volatile memories (CHI; LEE; XIE, 2014; ENDOH et al., 2016; LEE, 2016)

	PCRAM	STT-RAM	RRAM	DRAM	SRAM
Minimal cell size (SLC) (F^2)	4	6	4	6	160
Write latency (ns)	≈ 100	< 10	≈ 50	$20 - 50$	≤ 2
Read latency (ns)	5	5	5	$20 - 50$	≤ 2
Operation voltage (V)	$1.5 - 1.8$	$0.8 - 1.8$	$3.3 - 6.5$	$1.35 - 1.65$	$0.6 - 1.1$
Write current (A)	10^{-4}	10^{-5}	10^{-4}	10^{-5}	10^{-5}
Retention (years)	> 10	> 10	> 10	-	-
Endurance (no. of writes)	$\leq 10^9$	$\leq 10^{12}$	$\leq 10^6$	10^{15}	10^{15}

2015).

The values described in Table 1 present many variations, since different related work shows different values. Firstly, PCRAM and RRAM cells could be developed under the smallest sizes. On the other hand, performing a write operation in these memories is costly in terms of energy and specially time. Also, RRAM is classified as a more recent technology (ENDOH et al., 2016) which has the main issue of its endurance – it is inferior in comparison with other NVMs.

As pointed out and seen in the table, all NVMs suffer from poor write endurance. Regarding energy costs, STT-RAM cells have power consumption values which are comparable to DRAM cells. Thus, in comparison with other non-volatile technologies, STT-RAM shows the best overall electrical/physical features, and it meets features that can replace both main and cache memories (KÜLTÜRSAY et al., 2013; YAZDANSHENAS et al., 2014).

Many solutions have been proposed in order to mitigate the issues found on different NVMs (DASARI; NELIS; MOSSE, 2013; MITTAL; VETTER; LI, 2015; MITTAL; VETTER, 2016), which include (i) addition of volatile technology buffers (DRAM and SRAM), (ii) relaxing retention of NVMs, (iii) adopting helpful mechanisms to analyze memory scheduling operations taking in account the asymmetry of read and write operation latencies, (iv) compiler-oriented techniques, among other techniques.

2.5 Conclusion

This chapter presented an overview of NVMs. Three main non-volatile technologies (the NVMs that are most promising future memories) were depicted, showing their unique features, strong and weak points. Lastly, a brief discussion was done in the sense of comparison between these NVMs plus DRAM and SRAM, highlighting the issues presented by non-volatile technologies. Lastly, some solutions proposed to mitigate NVM problems were cited, based on studied related work.

3 MEMORY OPERATION SCHEDULING

One of the key challenges to get better performance from actual memory technologies resides in serving read and write memory instructions in the best instance of time possible. Scheduling is considered to be a complex problem, since it needs to cope with various issues, which include (MARTINEZ; IPEK, 2009; KIM et al., 2010; GOOSSENS et al., 2016):

- Requiring circumventing access scheduling constraints;
- Prioritizing memory requests properly;
- Avoiding conflicts over memory chip devices, such as banks, row-buffers and buses;
- Efficiently deal with changes in application states, i.e., they do not generate memory operations all the time.

These issues occur over all categories of memory controllers considering actual technologies. When considering using an NVM as main memory, problems found in these memories are also issues that need to be considered by memory controllers – those include asymmetry of read and write operations, plus limited write endurance.

In order to approach these problems, some related work have proposed memory controller scheduling policies and designs that mitigate these issues, thus making it more feasible to adopt an NVM as main memory (ZHOU et al., 2011; HU et al., 2014). Other solutions include increasing of cache sizes which absorb more writes and reads that would be done over the main memories. Other solutions include the adoption of hybrid architectures, i.e., by using an NVM as main memory, there may be additional buffers (usually with current technologies) used to mitigate the impact of write operations.

Other possible solutions include the programming of memory controllers that are aware of the particular characteristics of each NVM. Thus, these controllers may

perform different scheduling decisions when dealing with unequal operations (reads/writes). When modifying memory controllers considering the use of current technologies, fixed latencies for both write and read operations are assumed – a method that is appropriate in DRAM-based-systems (DASARI; NELIS; MOSSE, 2013). However, on NVM-based systems, memory write operations are more costly on both time and energy. Hence, better timing control of scheduling memory operations may make it feasible to adopt NVM as a main memory in real-time systems. These type of systems have strict timing requirements, which include meeting operational deadlines (BURNS et al., 2015; KUMAR; KARSAI, 2015). In addition, NVMs could be also used in embedded systems, since they are usually have energy constraints (SALEHI; EJLALI, 2015). Therefore, better arrangement of memory requests is seen as an option to aid NVM-based systems. Dasari, Nelis and Mosse (2013) proposed a mechanism to help analysis of memory requests timing, which has a primary objective of estimating tight timing for memory requests taking in account of the asymmetry of read and write operations. This method is better detailed in the section below.

3.1 Analysis of memory requests timing mechanism

The proposed method of memory requests timing goals to make it practical to deploy real-time applications over PCRAM main memory based systems, taking in account the differences between latencies on read and write operations (DASARI; NELIS; MOSSE, 2013). In order to perform a better and more detailed analysis, some constraints on evaluated tasks (which are simply sets of one or more memory requests) have been imposed. Those include:

- Tasks cannot migrate from one core to another, in case of multiple core architectures;
- Tasks have a fixed level of priority;
- Once a task is dispatched, it must complete its execution without being interrupted or preempted;
- If a task is designed to run at a certain time, even though it completes its execution earlier, the core remains idle until that time – no matter if other jobs are waiting execution.

Including these constraints over the sets of memory requests, some limitations on individual memory operation requests were also forced.

- **Read requests:** Once a read request has been issued by a single core, that core cannot issue a new read request until it receives the response of the previous request;

- **Write requests:**
  - Once a write request is issued, it is directed to write buffers, so that a task can proceed without waiting for the operation finishing;
  - If the write queue is not full, the controller serves pending read requests – in the case of having only write requests waiting to be issued, those are also served;
  - When the write queue is full, all pending requests are sorted according to priority (includes both writes and reads) – and then the controller serves them until the write queue is not full.

The imposed restraints over both read and write operations dictate the logic to the scheduling mechanism that is proposed. With that rationale, the memory controller can have two different states, being called the **busy** and the **idle** periods. These states can be visualized in two automata: In the **busy** automaton (shown in Figure 7), the algorithm iterates as long as memory requests can be generated. When no further requests can be generated, the algorithm switches to the **idle** automaton (depicted in Figure 8), where it waits for a memory request to be generated so that it can switch back to the busy automaton. Additionally, Table 2 contains all notations used in both automata.

Table 2 – Notations in busy and idle automata

Notation	**Meaning**
wqcap	Write queue capacity
wqlen	Number of slots used in write queue
inRd inWr	Number of incoming read and write requests, respectively
k curtime	Iteration index and the current time respectively
BP^k ID^k	Current time after the k^{th} iteration in busy period and idle period, respectively
StartBusy(w) EndBusy(w)	Stores the time at which the w^{th} busy period starts and ends, respectively
StartIdle(w) EndIdle(w)	Stores the time at which the w^{th} idle period starts and ends, respectively
LengthBusy(w) LengthIdle(w)	Length of the w^{th} busy and idle period, respectively
TR TW	Upper bounds on the time to serve a read and a write request by the memory module

The flowchart shown in Figure 7 models the controller when in busy period, where at least one read or write request needs to be issued. Once the busy period starts, the moment this happens is stored. After that, the main iteration of this period begins: It is

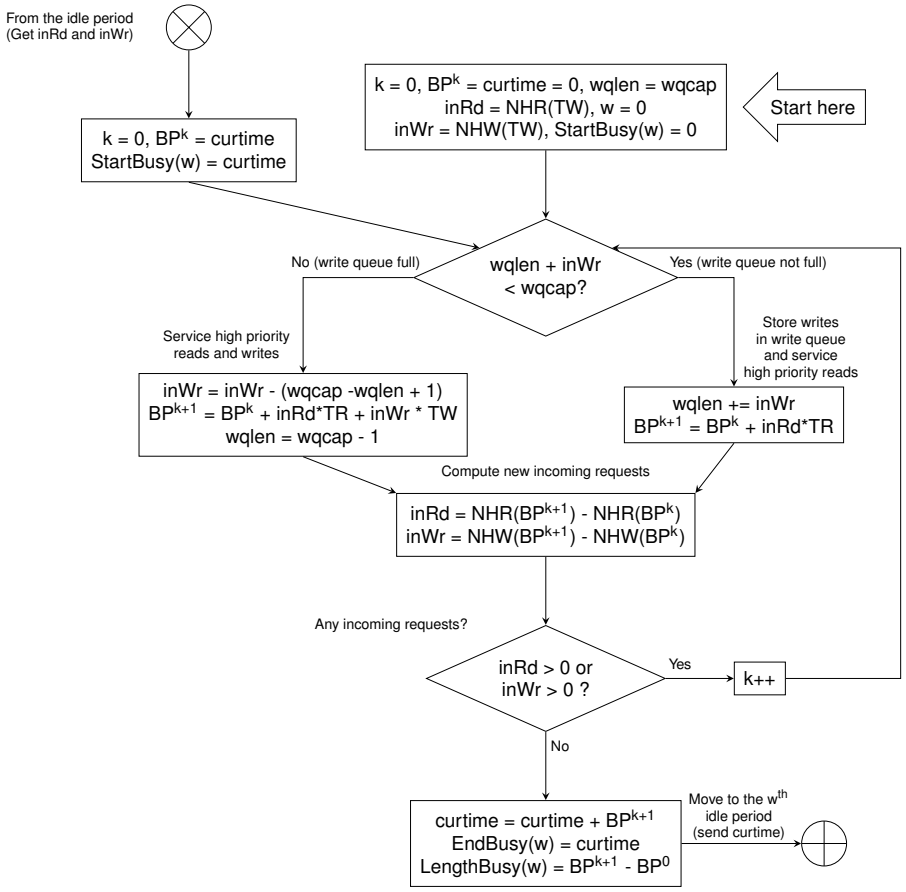


Figure 7 – The busy automaton (DASARI; NELIS; MOSSE, 2013)

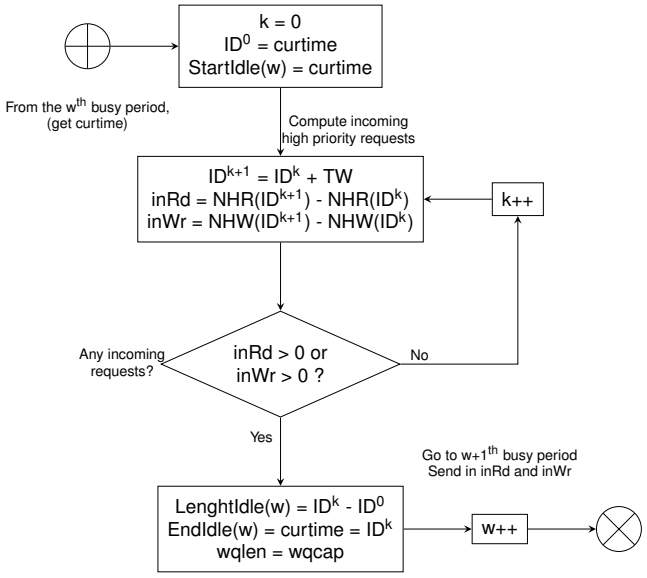


Figure 8 – The idle automaton (DASARI; NELIS; MOSSE, 2013)

checked if the write queue is full, including the arriving of one or more incoming write requests. Two independent actions can be done depending on the condition:

- **Write queue not full:** The controller simply deals with pending read requests. This branch of the algorithm prioritizes read over write operations;

- **Write queue full:** The controller will serve both read and write pending requests according to their priority – once the write queue has available slots, the controller switches back to the reads-over-writes schema.

Taking either action, the algorithm then computes if there are any more incoming read or write memory requests. Once again, two actions can occur:

- **Any incoming memory requests arrived:** The controller simply increments the iteration of the busy period, and returns back to the main iteration of the algorithm;
- **No incoming memory requests arrived:** This denotes the end of the busy period. Some variables are set here, which include the time the busy period ends, and how long the busy period was.

When the busy period ends, then the algorithm switches to idle period, where its flowchart is seen in Figure 8.

The idle period automaton has a simple iteration. Firstly, the start of the period is stored (this occurs at the same time the busy period ends). Then, the main loop of the automaton starts, where it checks for any incoming memory requests. Two independent actions can occur depending on the condition:

- **Any incoming memory requests arrived:** This denotes the end of the idle period. Some variables are set here, which include the time the idle period ends, and how long the idle period was.
- **No incoming memory requests arrived:** The controller simply increments the iteration of the idle period, and returns back to check if any memory requests have arrived.

In short, the memory controller implementing this algorithm works by switching its state from busy to idle, and vice-versa – trying to schedule memory requests as they come, taking in account the discrepancy of write and read latencies. All of the working logic of this memory controller is essential and dictates the logic to the implementation presented on this work, which is explained in the next chapter.

3.2 Conclusion

This chapter showed the importance of memory controllers in computing systems, citing their main issues. Assuming the use of a NVM as a main memory, new problems in memory controllers arose – which makes scheduling of memory requests being more challenging due to natural characteristics found in NVMs.

An explanation on a method that proposes a memory controller aware of the NVM issues (specially the increased latency when performing a memory write operation) was done. This method is the base to the development of this work, as it will be seen in the next chapter.

4 MEMORY CONTROLLER FOR NON-VOLATILE MEMORIES

Based on both subjects studied, NVMs and scheduling of memory operations, this work shows an implementation of a memory controller in simulation for NVM-based systems. In order to do this, a controller was implemented and tested considering previous analyses of memory controllers and NVMs. This Chapter explains the main contribution of this work (Section 4.1), then showing how it was done: Section 4.2 shows the tools used for this work (NVMain and Gem5) and Section 4.3 explains the implementation of a memory controller directed for NVMs. Lastly, Section 4.4 concludes the Chapter.

4.1 Main Contribution

Previous analysis show that taking the NVM characteristics in account could reach better results. However, these studies were done based on memory operation traces of applications, which means the analysis were done following two steps:

1. Applications were run in order to generate traces of memory operations, keeping track of its type (read or write) and instance of time that it executed;

2. Based on the type and time of operation, a static analysis over the traces was done so that it could be possible to evaluate where it was possible to get improvements in memory operations.

Hence, it was important to extend that analysis to the runtime, i.e., letting memory controller decide the scheduling while an application is being executed. By implementing a memory controller, it was possible to perform the runtime analysis which previous work (DASARI; NELIS; MOSSE, 2013) did not have. With dynamic analysis, it was possible to evaluate the behavior of the controller as the requests arrived – allowing for more realistic results on simulations.

Additionally, with three different NVMs studied, the implemented memory controller had to be tested with different non-volatile technologies. Each NVM has its particu-

lar characteristics, hence the impact of memory controllers in a specific NVM can be different if comparing with another NVM.

Different behavioral analyses of the implemented memory controller were done. The evaluated features included (i) time spent in generating memory requests, (ii) use of memory request queues, (iii) latencies of read operations considering different non-volatile technologies and (iv) comparison with NVMain default memory controllers.

4.2 NVMain and Gem5

To make this work feasible, two tools were used. An NVM simulator was necessary in order to implement the different memory controller that is being proposed. For this, NVMain (POREMBA; XIE, 2012) was chosen. The Gem5 (BINKERT et al., 2011) simulator was also chosen to aid the development of the work, because NVMain can be used in conjunction with Gem5 to provide closer to full system simulation.

The NVMain simulator (POREMBA; XIE, 2012) is a tool that is introduced to help the community on the modeling of both commodity DRAMs but also emerging memory technologies, which include NVMs. It is an architectural-level simulator for both types of main memory, which can model energy plus cycle-accurate operation of DRAMs and NVMs – including hybrid designs. It is considered more of a higher-level simulator, since it is more interested in the set of cells that compose the main memory, which is the hierarchy of columns, rows, banks, ranks and channels.

This tool was programmed using C++, and it has its source code available in an online repository (POREMBA, 2012), which can be given permission to get if one contacts the author (POREMBA, 2015). The current version of NVMain (2.0) added more features to support both main memory simulations, which include fine-grained memory bank model, MLC support, more flexible address translation and hooks to encourage users to explore new memory system designs (POREMBA; ZHANG; XIE, 2015).

The reasons to choose NVMain as the NVM simulator include its continuous updates over the years that keeps itself up-to-date with technology improvements. NVMain also has well documented code, and it is flexible enough to implement different memory controllers, hierarchies, prefetchers, and other memory objects used in the tool. Another key factor to choose NVMain resides in its possibility to merge itself with the general-purpose simulator Gem5, where it can provide closer to full system simulation – by using only NVMain itself, it is only feasible to perform trace simulations.

Gem5 (BINKERT et al., 2011) is a general-purpose simulator which merges some aspects of two tools: M5 and GEMS simulators – as both tools were used in many previous publications. Gem5 aims to be a community tool focused on architectural modeling, providing flexibility on different CPU, memory and interconnect models.

This tool is implemented mostly under the C++ language, with some parts pro-

grammed using Python. The source code of Gem5 is also freely available on an online repository. (BINKERT, 2012). Besides the integration with NVMain as an important factor to choose Gem5, its constant updates and fixes are also significant to pick this tool. Gem5 is also used in a myriad of published work, which makes itself relevant to the academia.

4.3 Memory Controller Implementation

With the chosen tools, a memory controller was coded in order to evaluate it, and then comparing with already implemented NVMain memory controllers to check its performance. Firstly, an understanding of the NVMain architecture is presented. An overview of the high-level design of NVMain is shown in Figure 9. Each box presented in the figure represents a memory base object, which represents either functions or classes in the source code of NVMain. In order to develop a different memory request scheduler, the boxes with **thicker** borders are considered to be the key objects.

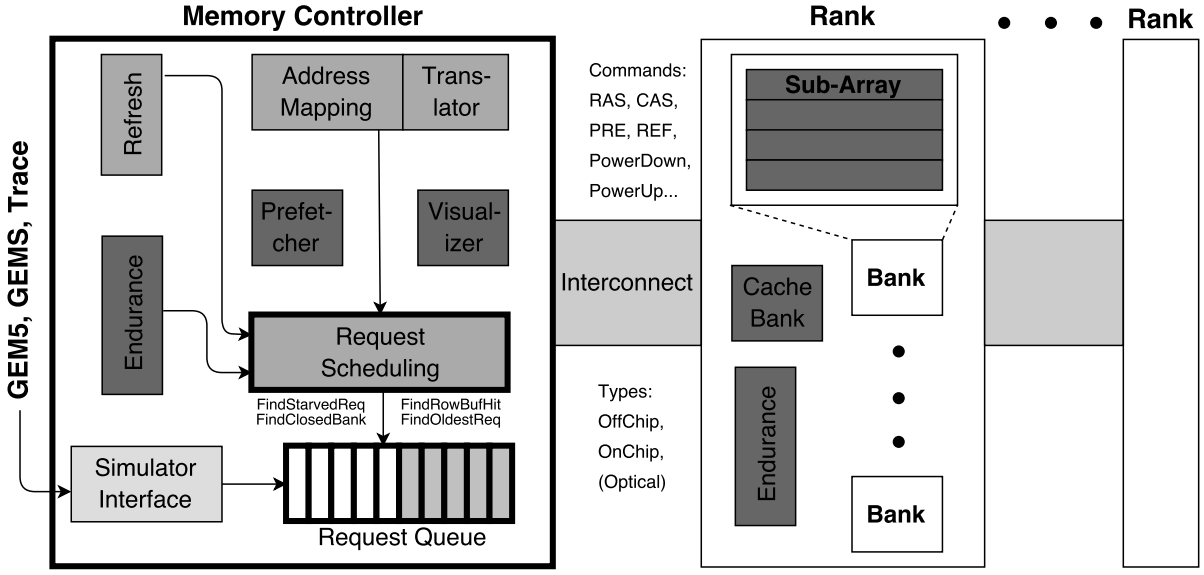


Figure 9 – Overview of NVMain Architecture: one memory controller for one memory channel (POREMBA; ZHANG; XIE, 2015)

In the NVMain source code context, Memory Controller is a base class with pre-implemented methods, called `MemoryController`. This class also extends from a superclass called `NVMObject`, which portrays a memory base object. In order to create a new memory controller, it is necessary to create a new class based on `MemoryController`. NVMain already comes with some main memory controllers for testing purposes, which are based on the simple schema of FCFS (First come first served) algorithm, which shows the basic idea of how a memory controller can deal with memory requests.

The first important step to set up a new memory controller is its internal variables. Table 3 shows the variables used in the implementation. Variable `state` determines what state is the memory controller, working like the two possible states a memory controller can have in the mechanism explained in Section 3.1. – when busy, it is in a state where memory requests are being generated, which means there is still a memory request waiting in a queue to be issued. When the state is idle, the memory controller does not have requests in a queue. When a state changes, the instance of time of this event is registered, using the variables `startBusy`, `endBusy`, `startIdle` and `endIdle`. Their variable types are paired with the one used to register instances of time (in NVMain they are called cycles), so an unsigned, 64-bit integer is used.

Table 3 – Internal parameters used in custom memory controller

Type	Variable	Purpose
bool	`state`	Controls what state memory controller is (Busy or idle).
uint64_t	`startBusy`	Instance of time that the busy state started.
uint64_t	`endBusy`	Instance of time that the busy state ended.
uint64_t	`startIdle`	Instance of time that the idle state started.
uint64_t	`endIdle`	Instance of time that the idle state ended.
int	`queueNumber`	How many queues memory controller has. This is always 3.
int	`writeQueueSize`	Maximum size of write request queue, in slots.
int	`readQueueSize`	Maximum size of read request queue, in slots.
int	`extraQueueSize`	Maximum size of extra request queue, in slots. By default, this is the sum of `writeQueueSize` and `readQueueSize`

Variable `queueNumber` has a constant value – it is always 3, since it denotes how many queues are used in the memory controller. There are read and write queues, which simply are reserved to keep read and write memory requests respectively. In addition, there is an extra queue that is able to receive both read and write requests, only used when the write queue becomes full. The role of this queue will be discussed in detail further.

The next step of implementation is the definition of the working logic of the memory controller. In this work, some diagrams were drawn in order to show this logic, using some notations to show the actions the controller takes. Table 4 summarizes all notations used on Figures 10–12.

The next step of implementation is based off from already established memory controllers – a method called `IsIssuable`. This method simply receives a memory request and returns a boolean value that indicates whether the request can be issued or not. Figure 10 shows a diagram of the `IsIssuable` method. If the request that arrives is a read from memory, it checks if the read queue is not full. In case this is

Table 4 – Notations used in the diagrams of the memory controller

Notation	Description
m_r	A main memory request
q_r	Queue used to hold read main memory requests
q_w	Queue used to hold write main memory requests
enqueue(q_x, m_r)	Method to enqueue a main memory request in a queue, where x = {r,w}
state	The state the controller is currently in. Can be busy or idle
q_e	Extra queue which is used when q_w is full
findRequest(q_x)	A method that tries to retrieve a memory request from a queue, where x = {r,w,e}

true, the request can be issued. However, when that queue is full, it cannot accept more requests, thus returning false. The same logic works for the arrival of a write request – it checks for the write queue instead. This method is implemented here since the base method `IsIssuable` from `MemoryController` always returns true, i.e., a memory request can be always issued. However, since memory requests need to be put in a queue, it may be already full. Thus, this method can inform that the memory request cannot be issued.

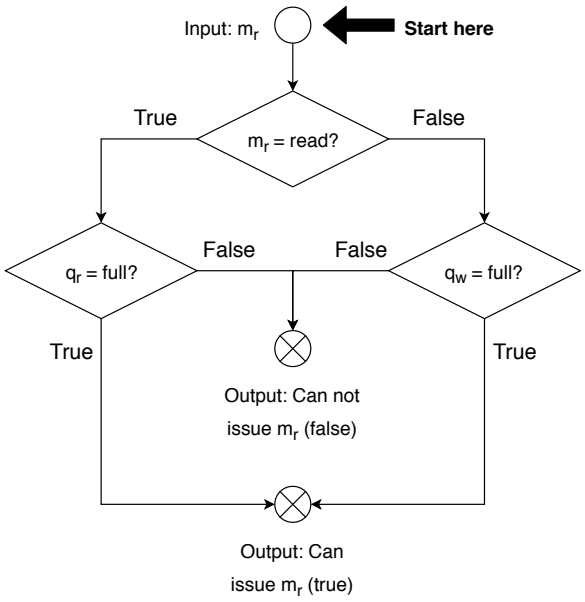


Figure 10 – Diagram to check if a memory request can be issued based on the request queue capacity (`IsIssuable` method)

When a request can be issued, it has to be put in its respective queue so that the scheduler can organize what memory requests will turn into operations. Method `IssueCommand` is responsible for these actions, and its diagram is depicted in Figure 11. The first action taken by `IssueCommand` is call the previously explained method `IsIssuable` in order to check if a memory request can be issued. If it cannot be issued, then the methods stop its execution. Else, the controller is ready to put a request in its

respective queue, depending of the type of the request.

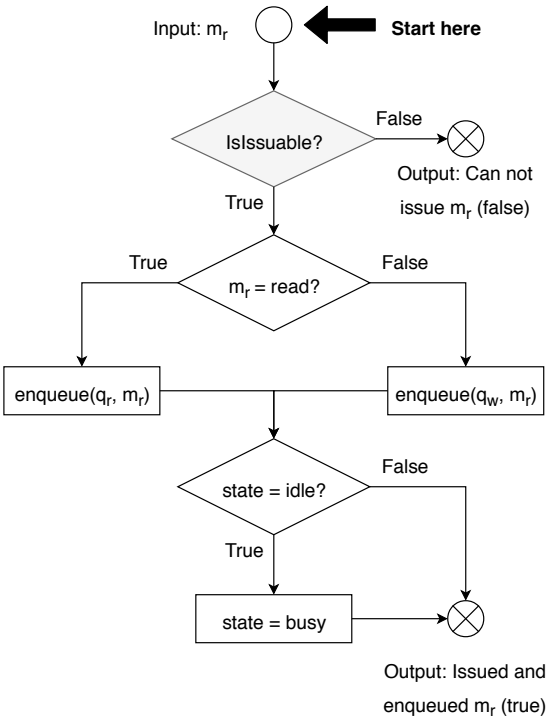


Figure 11 – Diagram that checks if a memory request will be issued (`IssueCommand` method)

Additionally, here the memory controller can change its state. This can occur when the controller is in idle state, which represents no memory requests being generated. However, if this method is called and a memory request was enqueued, it is necessary to change the state to the busy one. The instance of time this occurred is registered – where the busy state started and the idle state ended. After changing the state, `IssueCommand` ends its execution successfully.

The next implemented method, named `Cycle` is also extended from the `MemoryController` class, and it looks for enqueued memory requests. Its diagram is shown in Figure 12. The first step of this method is to check if the extra queue is not empty, which is always true when the controller receives the first main memory request. In this case, the controller will then check if the write queue is full of requests – firstly, it is assumed that this case does not occur. Considering that, the controller then tries to get a request from the read queue. If not possible, the code then tries to retrieve a memory request from the write queue. Thus, the scheduler prioritizes read requests over write requests.

Eventually, when attending many read requests the write queue may be filled with requests. When this occurs, all stored write requests have to be attended. In this case, the method called `queueTransfer` is executed. This takes all requests from both read and write queues to the extra queue, following this procedure:

1. All requests from both write and read queues are removed from their respective queues;
2. These requests are sorted by priority. It is considered that a request which arrived earlier in a queue has a higher priority than another request that was enqueued later;
3. The sorted requests are put in the extra queue, ending the method execution.

The last case that can attend a memory request occurs when there is at least one request that can be retrieved from the extra queue. The extra queue gets the priority over all other queues, since it represents the attending of all requests when the write queue is full.

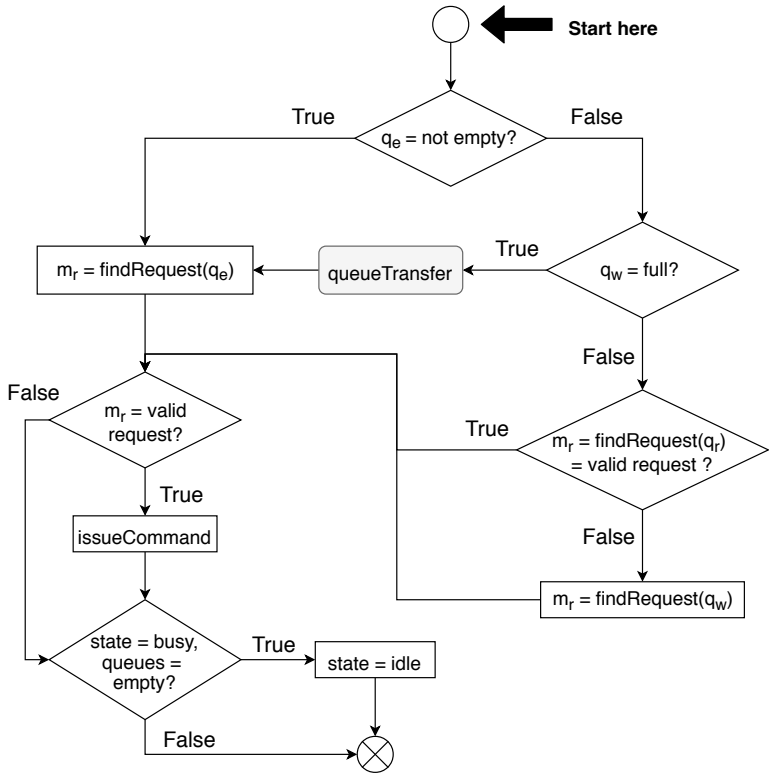


Figure 12 – Diagram that schedules requests and turn them into operations (`Cycle` method)

In any of the cases explained before, the algorithm will try to get a memory request using the method `findRequest`, which can store either a valid memory request or a null value. After trying to retrieve a memory request from any queue, the controller will check if the request gotten is valid or not. If it is valid, then the next step is to turn it into a memory operation, which is done by calling an already implemented `MemoryController` method called `IssueMemoryCommands`.

Lastly, after performing the memory operation the controller may change its state. This is possible whenever the controller is in busy state and all queues are empty. This

means no memory requests were generated to be scheduled, so the state is changed to the idle one. The time of this event is registered, denoting the start of idle state and the end of busy state. Finally, `Cycle` ends its execution.

Considering the work logic of the implemented memory controller, the code for it was developed and inserted in NVMain. It is important to point that any technology of memory can be simulated using this controller. This is possible since in order to perform an execution using a certain memory technology, a configuration file describing the memory needs to be specified. In addition to that, NVMain already comes with examples of memory technologies that can be executed.

4.4 Conclusion

This Chapter pointed the need of expanding timing analysis of memory requests in systems with an NVM as a main memory – runtime analysis is preferred over static analysis. The Chapter explained the main contribution of this work, and how it was reached – which was possible to use NVMain and Gem5 tools were chosen, since they can work together to reach the desired results.

Additionally, this chapter presented how the analysis of Section 3.1 could be translated in a practical environment – which was possible to do by coding a custom memory controller in NVMain, so that it could be attached to Gem5, turning it possible to perform a runtime behavior analysis.

5 RESULTS

This Chapter shows the main results reached by this work. With the modifications performed to include the implemented memory controller, both simulators were configured. This is explained in Section 5.1. Then, a set of applications were chosen to run in the simulator. For this job, two sets of benchmarks were picked: MiBench (LEE; POTKONJAK; MANGIONE-SMITH, 1997) and MediaBench (GUTHAUS et al., 2001). Applications were tested and selected to evaluations of the memory controller. In addition to that, some changes in the input data were performed to reflect more realistic and stressful executions – this is explained in Section 5.2. Then, a profiling of the chosen benchmarks was done and explained in Section 5.3. In Section 5.4 the behavioral analyses of the implemented memory controller were exposed. Section 5.5 compares the implemented memory controller with NVMain default memory controllers, and lastly, Section 5.6 concludes the Chapter.

5.1 Simulation configurations

In order to obtain results, it is important to highlight the configurations of the tools. On Gem5, one of the configuration relates to memory hierarchy. For all simulations it was considered the use of two levels of cache, reflecting default configurations found on Gem5. Firstly, the L1 cache is composed of two caches: a 32KB cache for instructions, plus a 32KB cache for data. Then, there is a L2 cache with 2MB. The CPU used on simulation was the `TimingSimpleCPU`, which reflects a simple model of a CPU, which uses timing memory accesses. This model connects the CPU to the cache, defining the necessary functions to handle memory accesses, which allowed to perform a better merge with NVMain memory system. Furthermore, Gem5 was compiled using the `fast` option, which allowed for code optimization and better performance and the instruction set architecture (ISA) x86 was used.

With NVMain, parameters regarding the main memory system were set. For each NVM technology, the original configuration files for each memory (PCRAM, RRAM and STT-RAM) were used. However, some changes in the general memory system config-

uration were done regarding the main memory schema. All NVMs followed the same schema found in the PCRAM configuration file – this was done in order to allow tests under the same memory system configuration, preserving its natural characteristics, which include device timing parameters and energy parameters. These parameters were extracted from (CHOI et al., 2012) (PCRAM), (KAWAHARA et al., 2012) (RRAM) and (EVERSPIN, 2015) (STT-RAM).

The implemented memory controller also had parameters which needed to be defined in order to perform tests. Since the memory controller is based on memory requests queues, they needed to have limited sizes. Observing the already implemented memory controllers on NVMain, the default queues could hold 16 requests – a value maintained in the implemented memory controller for both read and write queues. For the extra queue, it was used a queue which could hold 32 requests, as it can potentially hold all requests from both write and read queues due to the scheduling used in the memory controller.

5.2 Benchmark checking and manipulation

Firstly, a check-up of the applications of both sets of benchmarks was done – Table 5 summarized each application status, pointing out if the application could be compiled and executed. Applications that occurred in both sets of benchmarks (cases of `adpcm`, `ghostscript`, `jpeg` and `pgp`) were only shown in the MediaBench benchmark set. Some of the benchmarks have multiple execution stages, which are either decode/encode (decodification/codification stages) or a specific process (Applications `mesa` and `susan` have three different forms of execution, each one with a specific functionality). The majority of benchmarks executed normally, with some exceptions that need to be pointed out:

- `ghostscript` had some incompatibilities with some library files, which were fixed on compilation. However, it gives a segmentation fault error when running. This probably occurred due to the use of old libraries that are no longer supported;
- `pgp` gave errors when compiling auxiliary assembly files, which probably occurred because these files had 32-bit assembly instructions (a 64-bit architecture is used to compile benchmarks);
- `ispell` also compiled correctly, however it runs in an endless loop;
- `lame` also compiled without errors, but it gives a segmentation fault error;
- `tiff` applications (`tiff2bw`, `tiff2rgba`, `tiffdither` and `tiffmedian`) had

compilation problems in the `tiff` library, which is shared by all those applications.

The impediments to run properly these specific benchmarks probably occurred due to using versions of C libraries, which were used to compile and execute correctly on 32-bit architectures. When compiling applications to a 64-bit architecture, the use of more recent libraries likely lead to errors on applications. Because of these errors, these benchmarks were excluded from the analysis presented by this work.

Table 5 – Benchmark compiled and executed status

Benchmark set	Application	Stages	Compiled	Executed
MediaBench	adpcm	decode,encode	yes	yes
	epic	decode,encode	yes	yes
	g721	decode,encode	yes	yes
	ghostscript	-	yes	no
	gsm	decode,encode	yes	yes
	jpeg	decode,encode	yes	yes
	mesa	mipmap,osdemo,texgen	yes	yes
	mpeg2	decode,encode	yes	yes
	pegwit	decode,encode	yes	yes
	pgp	decode,encode	no	-
	rasta	-	yes	yes
MiBench	basicmath	-	yes	yes
	blowfish	decode,encode	yes	yes
	bitcount	-	yes	yes
	crc	-	yes	yes
	dijkstra	-	yes	yes
	fft	decode,encode	yes	yes
	ispell	-	yes	no
	lame	-	yes	no
	mad	-	yes	yes
	patricia	-	yes	yes
	quicksort	-	yes	yes
	rijndael	decode,encode	yes	yes
	rsynth	-	yes	yes
	sha	-	yes	yes
	sphinx	-	yes	yes
	stringsearch	-	yes	yes
	susan	corners,edges,smoothing	yes	yes
	tiff2bw	-	no	-
	tiff2rgba	-	no	-
	tiffdither	-	no	-
	tiffmedian	-	no	-
	typeset	-	yes	yes

In the runnable benchmarks, in order to reflect more stressful and real-world applications, some of the inputs used in applications were changed, making the benchmark data sets larger. When running applications with original data sets, it was noted that

Table 6 – Changes in input data of applications from MediaBench and MiBench

Benchmark	Input type	Old Input	Changed input
adpcm	.pcm file	Part of Bill Clinton's Speech (295,0 KB)	Die Walküre, by Richard Wagner (107,2 MB)
crc			
g721			
gsm			
epic	.pgm fle	Lena image, size 256x256 (65,0 KB)	A 1280x1024 wallpaper (1,3 MB)
susan		An office image, size 384x288 (110,7 KB)	
jpeg	.ppm file	A image with a rose, size 227x149 (101,5 KB)	A 1280x1024 wallpaper (3,9 MB)
mpeg2	.m2v file	A 4-frame video (34,9 KB)	A video with 20 seconds of length (1,3 MB)
mad	.mp3 file	A 30-second part of Now I Know Why You Wanna Hate Me, by Limp Bizkit (381,6 KB)	Die Walküre, by Richard Wagner (8,6 MB)
blowfish	.txt file	Kurt Vonnegut's commencement address at MIT in 1997, replicated 1408 times (3,2 MB)	Moby Dick by Herman Melville, replicated 10 times (12,5 MB)
rijndael			
sha			
pegwit		A simulation statistics file (91,5 KB)	
rsynth		Part of an article by Jeff Stark about David Halberstam's opinion about "Apocalypse Now" (3,3 KB)	
quicksort		A vector with 50,000 elements (1,6 MB)	A vector with 1,000,000 elements (5,3 MB)
dijkstra		A totally-conected graph with 100 vertexes	A totally-conected graph with 1000 vertexes
fft	parameters	Waves = 8, Length of Sinusoids = 32768	Waves = 16, Length of Sinusoids = 131072

cache memory absorbed most of memory requests. By making the data sets larger, more main memory requests could be generated, thus making the analysis of the impact of a different main memory controller feasible. In addition, when performing executions with memory hierarchies having smaller cache sizes, the time spent on the simulations increased, turning them impracticable. Table 6 summarizes all changes done in the original input data provided by the benchmarks. Applications that could not be executed did not have its data changed, since they would not enter in the simulations. Benchmarks `sphinx` and `patricia` did not had data changed since in test-runs they already generated a large number of memory requests, which were enough to perform analysis on main memory requests. `rasta` did not have its data changed due to technical difficulties to find proper input files.

5.3 Benchmark memory profiling

This work initially performed a memory profiling of applications that were executed from the sets of benchmarks, according to Table 5. Applications were simulated so that it could be possible to analyze patterns of generated memory requests.

Figures 13 and 14 show how many main memory requests were generated during the execution of an individual benchmark. Two charts were used to show data since the quantity of main memory requests generated greatly differs, depending on benchmark nature. The Y-axis shows how many main memory requests were generated during the execution of a benchmark, whereas the X-axis is used for the application names. In the case of a benchmark had multiple stages, an additional suffix was used. This is indicated by a hyphen and the initial letter of the stage, e.g., stage `edges` of benchmark `susan` is shown in the charts as `susan-e`.

In Figure 13 it is shown all applications that had 50,000 or less main memory read requests. The write requests were excluded from this chart since no application was capable of generate at least 1,000 write requests – they would not be visually noticeable. It is important to observe that even using larger inputs in these benchmarks, it was not possible to generate more main memory requests. In this context, a notable benchmark that needs to be highlighted is `dijkstra`, since it is a memory-intensive application that calculates all shortest distances of all nodes in a graph. It could not generate more than 30,000 main memory requests, even though it was the benchmark that generated the biggest quantity of memory requests (3.5×10^9) – however most of them were directed to the cache memories ($\approx 99.999\%$). Other applications have the same pattern, which include all of those involving cryptography (`blowfish`, `pegwit`, `rijndael` and `sha`) and PCM (Pulse-code modulation) file benchmarks (`adpcm`, `crc`, `g721` and `gsm`).

Regarding `basicmath` and `bitcount`, they are not memory-intensive bench-

marks: While the first tests mathematical calculations that include cubic function solving, integer square root and angle conversions, the second one tests the bit manipulation abilities of a processor, counting the number of bits in an array of integers, by using five different methods – thus, even increasing the amount of data to process, it will not have a significant impact on memory accesses. Applications `mad`, `stringsearch` and `rsynth` also have this similar pattern.

In short, the benchmarks shown here can be considered less representative about memory profiling, due to the low generation of main memory requests. Some applications did not even generate a single main memory write request, which occurred due to how the simulation dealt with memory requests. Benchmarks were executed in isolation, hence the dispute for hardware resources was non-existent. As the simulation starts, all memories are empty. Thus, the application begins using the fastest memory available. As a consequence of that, many memory requests do not even come to the main memory.

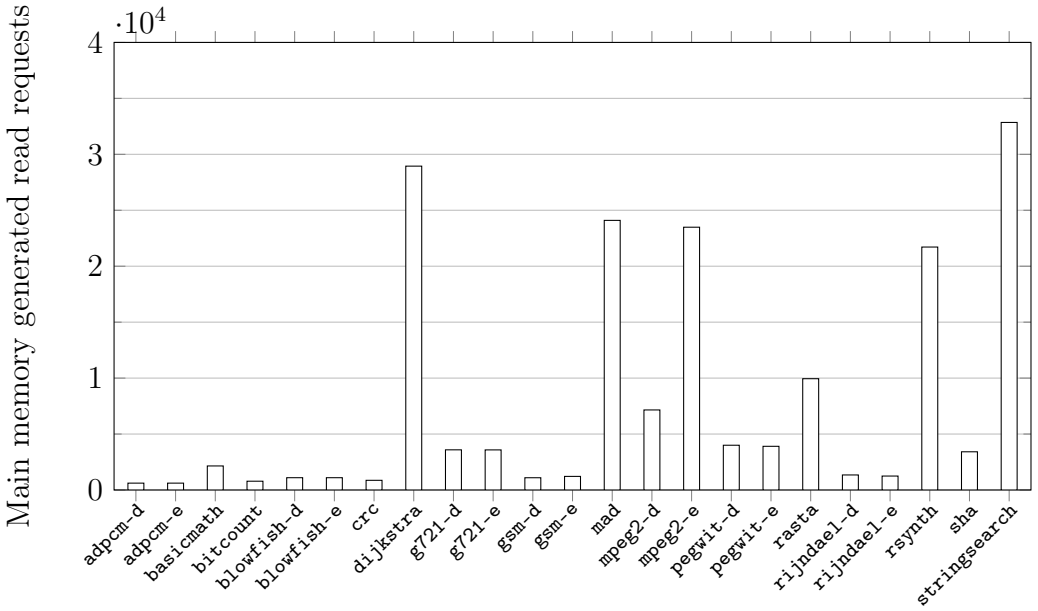


Figure 13 – Total of read main memory requests performed in each benchmark, where the number of read requests $\leq$ 50,000

Figure 14 has data of all benchmarks with more than 50,000 main memory read requests. In this case, applications with either original or bigger data inputs could generate a considerable amount of main memory requests. Considering the case of `sphinx`, the original large data set was used, and it reached the biggest amount of both read and write main memory requests. This occurred because of a combination of two factors: Firstly, it is an application that generated a great number of memory requests (8.1×10^8). Secondly, it was one of the benchmarks that had more cache misses, leading to more requests directed to main memory.

Regarding both stages of benchmark `epic`, they appear in this chart due to being

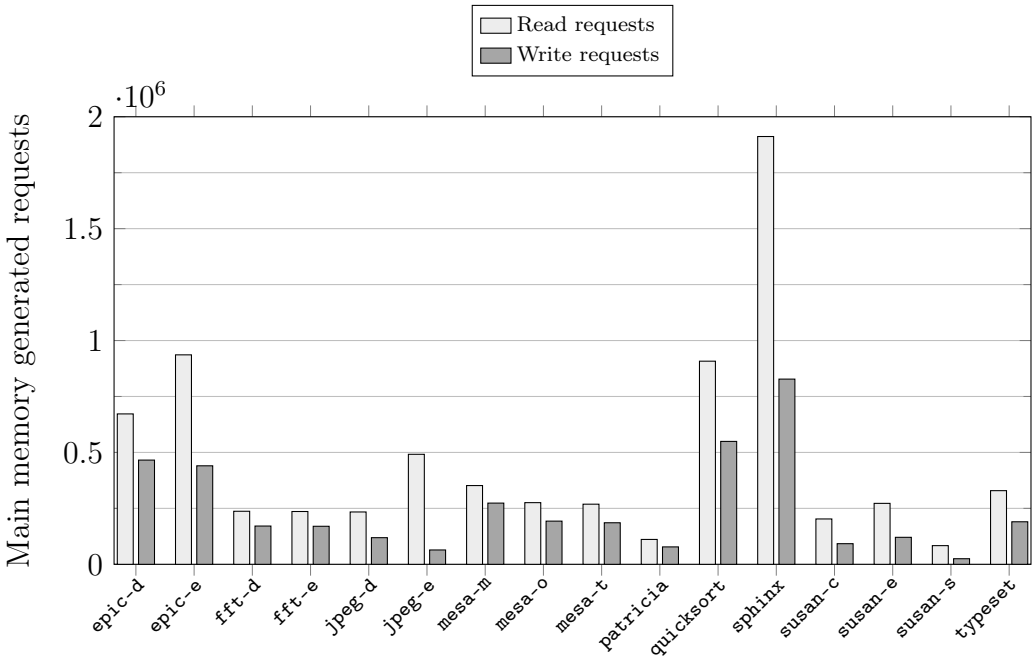


Figure 14 – Total of main memory requests performed in each benchmark, where the number of read requests $>$ 50,000

the executions that had the biggest cache miss rates, leading to substantial increases in main memory requests. On `epic decode`, $\approx 3.71\%$ of generated memory requests were directed to main memory, whereas on `epic encode` this value was $\approx 2.78\%$. The changes in the input data set also were a key factor to the increase in the overall number of memory requests (Original input was a 320x240 image, which was replaced by a 1280x1024 image). The remaining image manipulation applications (`jpeg`, `mesa` and `susan`) have the same pattern occurred in `epic`.

The remaining benchmark executions had a pattern that is similar to the one detected on image manipulation applications. On both `patricia` and `typeset` the original data inputs used generated reasonable main memory requests – however, both `fft` and `quicksort` benchmarks are memory-intensive ones, but the high number of generated main memory requests was helped by the increase of the application inputs.

5.4 Memory controller Evaluation

Knowing the behavior of main memory requests in benchmarks, the next results look to comprehend how the implemented memory controller behaves in the simulated executions.

5.4.1 Evaluating impact of different technologies of NVM as main memory

Firstly, an analysis was done by testing all benchmarks running over different NVMs as a main memory, using the implemented memory controller. These evaluations show

the average read and write latency of benchmarks using these NVM technologies as main memories. The latency includes all time spent in a main memory request: It starts counting when the memory request is generated, and it ends when the memory operation is performed.

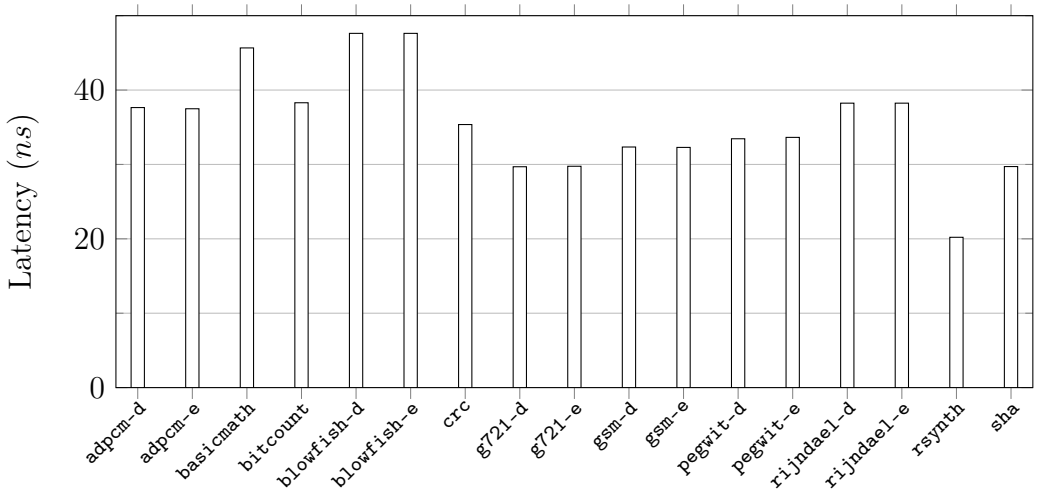
5.4.1.1 PCRAM

Firstly, a PCRAM was used as main memory in benchmark simulation. The average of read and write latencies were calculated for each application, which is depicted in Figure 15. However, due to a notable difference in latencies presented in the simulated benchmarks, they were split in two groups: Figure 15a show only read latencies of benchmarks which did not generated any main memory write requests. On the other hand, Figure 15b displays both read and write latencies of applications that generated main memory write requests.

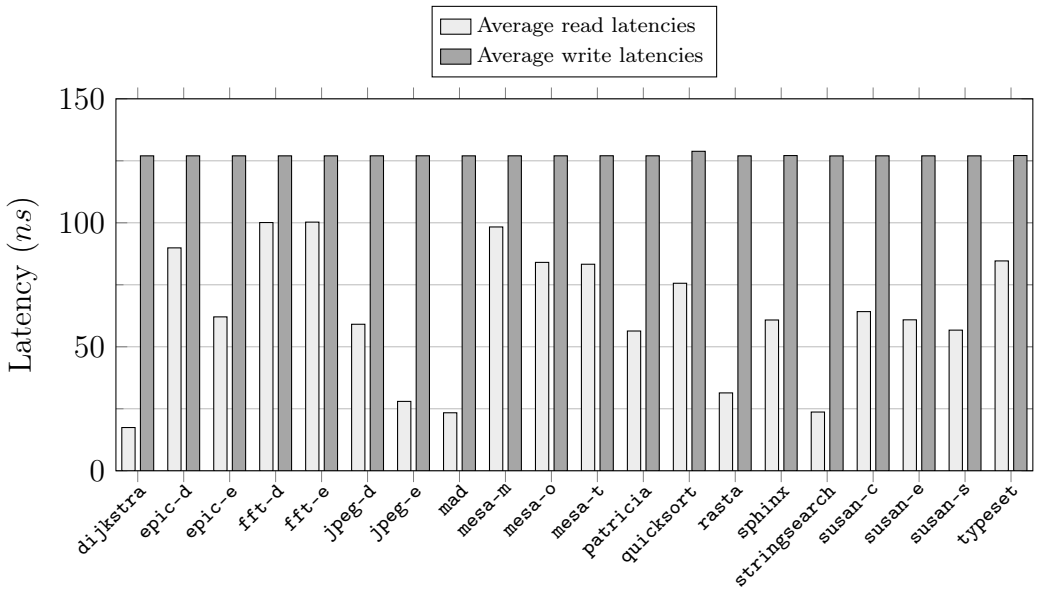
Figure 15a shows that read latencies vary from 20 to $50ns$ – which is a high value when considering current PCRAM read latencies of around $5ns$. This can be explained because the PCRAM configuration file provided by NVMain is based off from an older article (CHOI et al., 2012), and therefore, it could have a lower read latency, which is not given by the paper.

In Figure 15b, both read and write latencies are shown. Some applications had read latencies similar to the ones shown in Figure 15a – cases of `dijkstra`, `jpeg encode`, `mad`, `rasta` and `stringsearch`. This is directly related to the relatively low number of generated memory requests. However, as mainly main memory write requests grow (and so the length of busy periods), the read latencies also increases when using the implemented controller – For example both stages of `fft`, the average of read latencies surpasses $100ns$, which is considered to be a high latency for a read operation. On the other hand, benchmarks that generated less requests had similar average read latencies as applications with no main memory write requests. This occurred due to the high interference that write requests put over read requests. While a write operation to the main memory is being executed, one or more read requests can arrive to the controller. These read requests must wait for the write operation to finish so that they can be attended. Since a write operation is more time consuming than a read one, reads that are waiting for a write operation to finish needs more time to complete its execution.

Regarding average write latencies, it is observed that all simulated benchmarks had very similar results. The benchmark with best overall average write latency was `stringsearch` – $126.97ns$ – while the application with worst value was `quicksort` – $128.85ns$. This means the range (difference between highest and lowest values of a sample) of write latencies did not surpass $2ns$. The less variable values on write latencies can be explained: write main memory requests could be only influenced if a



(a) Latencies of read memory operations in benchmarks with no main memory write requests



(b) Latencies of read and write memory operations in benchmarks with generated main memory write requests

Figure 15 – Latencies of memory operations with a PCRAM as main memory

longer burst of read requests occurred, which did not happened. In this scenario, since read requests have priority to be attended over write ones, the controller would deal primarily with reads. Write requests then would spent more time on the queue, which would increase their latency.

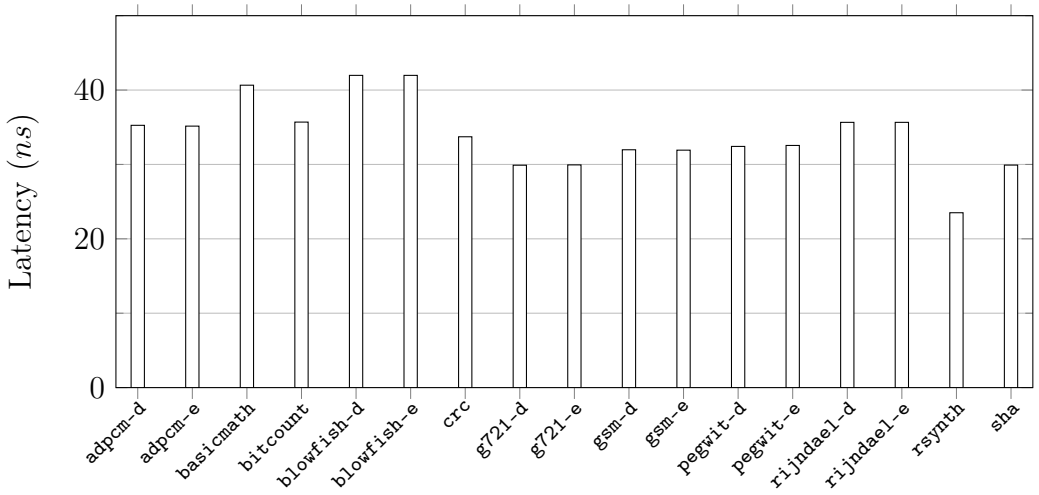
5.4.1.2 RRAM

The next simulations were executed using an RRAM as a main memory. Following the same logic shown in the evaluation of PCRAM, Figure 16 shows calculated average of read and write latencies for each application, splitting in two charts, where Figure 16a show only read latencies of benchmarks which did not generated any main memory write requests. On the other hand, Figure 16b displays both read and write latencies of applications that generated main memory write requests.

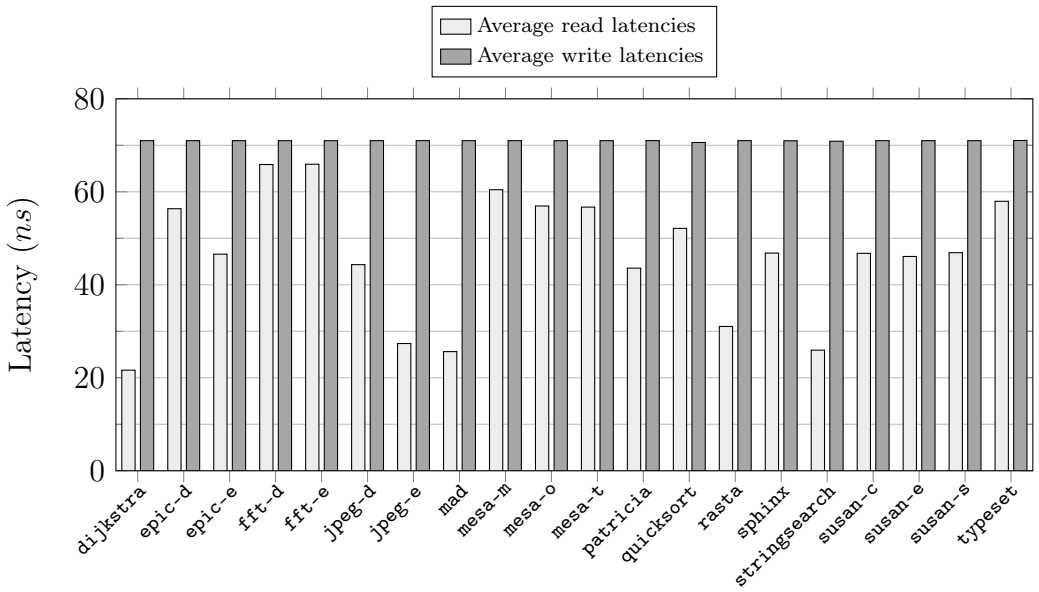
Analyzing Figure 16a, it is seen that benchmarks with no main memory write requests follow the same pattern as the PCRAM simulations – their average read latencies do not surpass $50ns$. In this case, the RRAM configuration file provided by NVMain is based off from (KAWAHARA et al., 2012). However, in this case it is specified that the RRAM cell read latency is confirmed to be $25ns$. The higher read latencies may be caused due to two factors: an overhead caused by the implemented memory controller, plus the physical memory size: While the memory shown in the article has a capacity of 8 Mb, the configuration file describes a 4 GB memory. Accessing a cell in memories with higher capacities tend to be slower than performing a memory operation in smaller memory chips.

In Figure 16b, the remaining benchmark simulations with both read and write latencies are shown. Benchmarks with less generated main memory requests tend to have better overall read latencies. On `dijkstra`, the average read latency hit around $21.6ns$, a value lower than expected. This probably occurred due to use of internal buffers which gave speed-ups on memory operations. `jpeg-e`, `mad` and `stringsearch` also had average read latencies consistent with the one given by (KAWAHARA et al., 2012).

As more main memory requests are generated, average read latencies increase, but in a smaller scale if we compare with results seen in PCRAM simulations. The worst cases continue occurring in both stages of `fft`, however, their average read latencies just above $65.8ns$. Depending on the benchmark simulated, read latencies still present variations, which can be explained by the same factor which was shown in PCRAM simulations: the interference of write requests on read requests. By using the implemented memory controller, this interference occurs independently of memory technology, as it is related to how the scheduling of memory operations is done. The notable difference comparing PCRAM and RRAM simulations is that the variation seen in read latencies occurred in a smaller scale.



(a) Latencies of read memory operations in benchmarks with no main memory write requests



(b) Latencies of read and write memory operations in benchmarks with generated main memory write requests

Figure 16 – Latencies of memory operations with an RRAM as main memory

Concerning write latencies, the same logic seen in PCRAM applies on RRAM as well: They all had very similar results, with few to none variations between the application runs. This happened for the same reasons pointed in the PCRAM analysis: write requests did not suffer from situations which could potentially occur, due to not occurring bursts of memory requests that could fill the write buffer. The difference here resides in the average of all simulated benchmarks, which was $\approx 71ns$, a smaller value if compared to PCRAM. This difference can be explained by the technology, since an RRAM is considered to have better write latencies when comparing to a PCRAM.

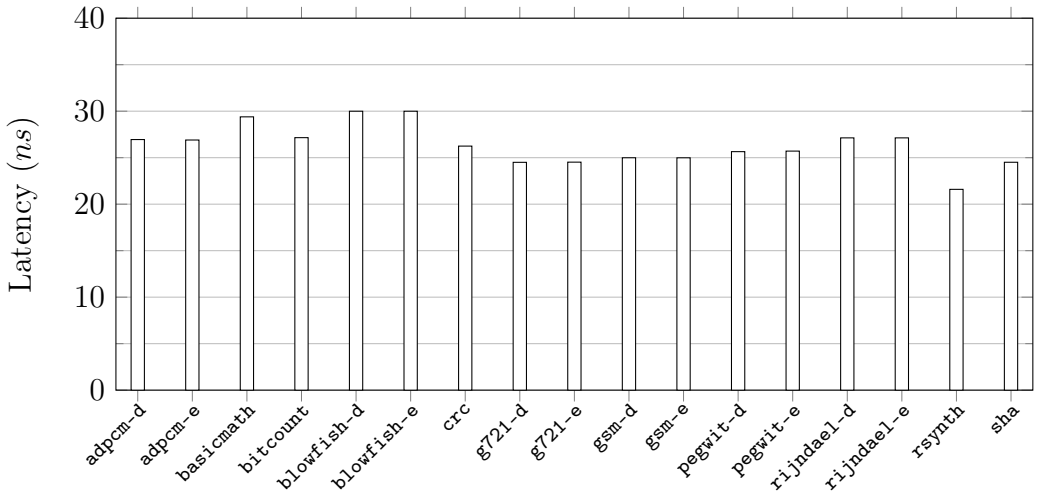
5.4.1.3 STT-RAM

The last NVM technology that was simulated was STT-RAM. Using the same organization seen in both previous NVM simulations (PCRAM and RRAM), Figure 17 has two charts with calculated average of read and write latencies for each benchmark. The first chart, shown in Figure 17a contains only average read latencies for applications that did not generate any main memory write requests. In Figure 17b shows both average read and write latencies in benchmarks that generated main memory write requests.

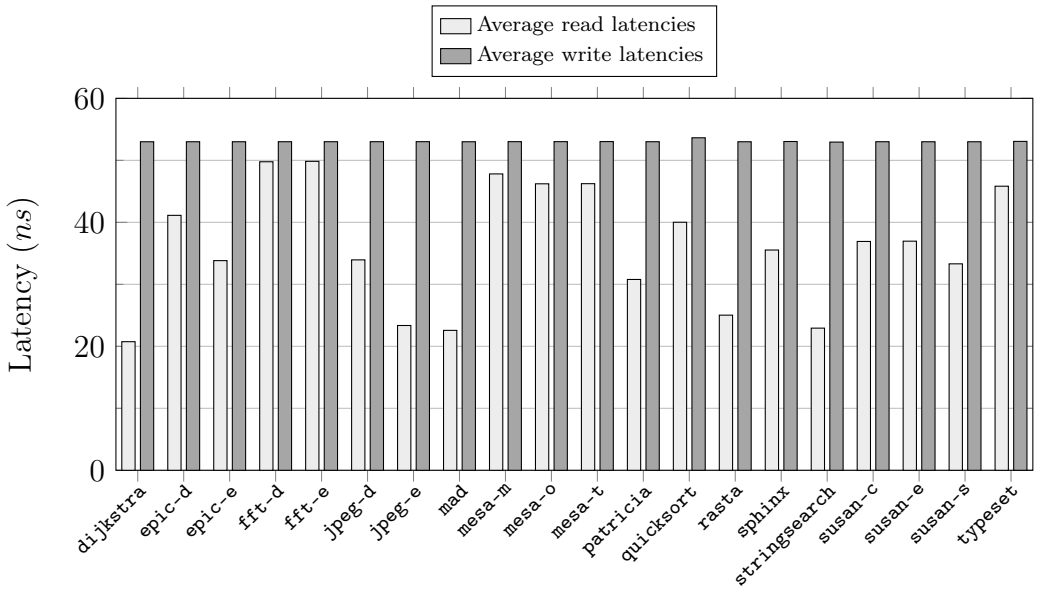
In Figure 17a, no surprises can be seen in the pattern of benchmarks with no main memory write requests: It is similar to previous NVM simulations. The key difference here is the technology used: With STT-RAM, applications do not have latencies higher than $31ns$. In this particular case, according to the datasheet that is used as base for the NVMain STT-RAM configuration file, the read latency of an STT-RAM cell is $35ns$ (EVERSPIN, 2015). That means simulated benchmarks had better performance in read latencies than expected, which can be explained by two factors: (i) the use of memory buffers, and (ii) incorrect specifications in the configuration file of the STT-RAM presented by NVMain. Both reasons reflect the same behavior of what occurred on RRAM.

Remaining benchmark simulations with both read and write latencies are shown in Figure 17b. The impact caused by writes over reads still exists, but in a lower scale. The lower impact occurred due to the technology used: writes still interfere on reads, but since a write operation in a STT-RAM cell has lower latency when comparing with PCRAM and RRAM, this interference is also lower. For example, the worst average read latency values, found in both stages of `fft`, did not surpass $50ns$, while the best average read latency (`dijkstra`) was $\approx 21ns$.

Regarding write latencies, the same pattern found in both PCRAM and RRAM also occurs when using STT-RAM as main memory. On STT-RAM, memory requests also did not suffered from possible scenarios of interference. Hence, these latencies had very few variations between benchmark runs, with an average value of $\approx 53.04ns$.



(a) Latencies of read memory operations in benchmarks with no main memory write requests



(b) Latencies of read and write memory operations in benchmarks with generated main memory write requests

Figure 17 – Latencies of memory operations with an STT-RAM as main memory

5.4.1.4 Overall analysis

Regarding different NVM technologies, lastly, Figure 18 summarizes all NVM analysis in a single chart, where it is shown the average of all read and write latencies of executed benchmarks, grouped by the NVM used as main memory in simulation. The error bars in the chart represent standard deviation. Additionally, it is important to highlight that each average was calculated differently:

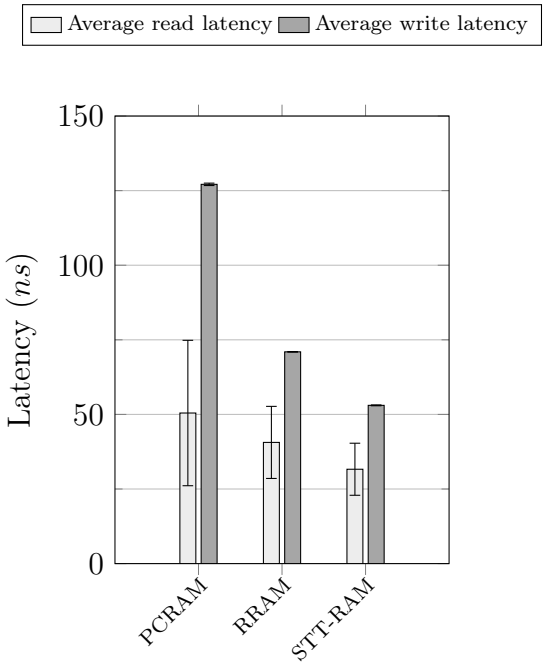


Figure 18 – Averages of read and write latencies for each NVM as main memory

- Read latency average: All applications executed were considered in order to calculate it;
- Write latency average: Benchmarks which did not generate main memory write requests did not entered in the calculation of the average.

The chart tells it is not clear which NVM holds on overall the best read latency, due to great variation presented on all simulations.By using the t test with a 95% confidence interval and comparing the means presented on the chart, it was noted that all read latency means do not have significant differences. Thus, in theory, all NVMs can have the potential to have similar latencies when perform read memory operation. However, it seems the implemented memory controller can meddle on read operations. On the other hand, the average write latencies had very small standard deviations, leading to see that STT-RAM performs the fastest write operations comparing with the other two technologies simulated – which could be checked both on simulation using the different memory controller and NVM review on this work.

5.4.2 Evaluating length of busy periods

Looking into the algorithm used to attend memory requests, an analysis of busy periods over the execution of the benchmarks was presented. Figure 19 presents the percentage of each busy period in a benchmark run – i.e., during all the execution of an application, it was observed how much time the memory controller in the **busy state**, which translates into generating and attending memory requests. Assuming the memory controller starts with no memory requests: As soon as a memory request arrives in the memory controller, this denotes the start of a busy period. When a memory request is turned into an operation and there are no requests on queue waiting to be attended, this denotes the end of a busy period. This value, (p_{busy}) was calculated using the equation shown in (1), where c_{busy} is how many cycles during a benchmark run were spent in the busy state, and c_{idle} is how many cycles in a benchmark execution were spent in the idle state.

$$p_{busy} = \frac{c_{busy}}{c_{busy} + c_{idle}} \times 100\% \tag{1}$$

For this experiment, it is presented an analysis using the STT-RAM main memory configuration was used alongside with the implemented memory controller. The reason to choose this NVM was previous results, with the best overall write latencies. In those simulations, many benchmarks had busy periods with very small values, with percentages less than 0.1% – those were excluded from this chart. It is perceptible that almost all benchmarks shown in this chart also appear in Figure 14 – with that, it is possible to point out that benchmarks that generate more memory requests in the shortest periods of time. However, due to the simulated architecture and cache memory reducing greatly the quantity of main memory requests, on average the time spent in busy state is small, not surpassing 2% of the total time.

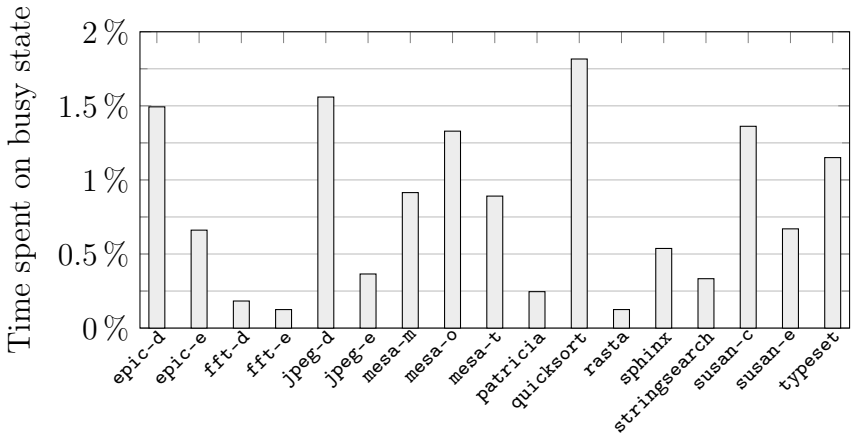


Figure 19 – Percentage of time spent in busy state during benchmark executions

Benchmark `sphinx` is a case to be highlighted. Even though it was shown that

it is the application that generated most main memory requests (both read and write ones), it is only the benchmark with the 10th longer period spent in busy state. This occurred because at the same time it generated a great number of memory requests, it also executed a large number of other instructions (branches, integer and floating point ones) – thus, the busy periods of the benchmark run were spread out alongside it.

With similar logic, it is seen why `quicksort` has the longest busy period: It generated a considerably large amount of main memory requests, however, the number of seconds simulated in `quicksort` was 1.63 seconds – in comparison, `sphinx` had 5.5 as the number of seconds simulated.

5.4.3 Evaluating use of queues

One key factor of the implemented memory controller is the presence of three different queues to store memory requests, due to how it was coded (one for read requests, other for write requests, and the extra queue – used when the write queue is full). Thus, it becomes important to evaluate how queues are used in the controller.

This analysis was directed towards the use of queues during the execution of the benchmarks. Figure 20 shows, on average, how many requests were present in queue while the state of execution in the memory controller was the busy one. This analysis was performed once again using the STT-RAM main memory configuration. Values that are exhibited in chart were calculated using Equation 2, where q represents a specific queue (read, write or extra), r_q represents how many requests are being stored in queue q in a specific cycle, and c_b is the total of cycles spent in the busy state. In short, this does an average of how many requests are being kept in queues while the application is in busy state. Calculation was done considering only the busy state since the way the controller was implemented, no memory requests are attended in the idle state.

$$m_q = \frac{\sum_{i=0}^{c_{busy}} r_q}{c_{busy}} \tag{2}$$

In this evaluation, there were some benchmarks which did not generate enough main memory requests in order to perform any analysis over it and thus they were excluded from this chart. All of these benchmarks had the pattern of generating one main memory request where as soon as it was queued, it was immediately attended. Hence, when applying Equation 2, m_q resulted in value 1 in these cases.

The applications shown in Figure 20 are, not coincidentally, the ones which had the biggest quantities of generated main memory requests – more requests to schedule, more requests which need to be queued. The low values of the average of requests ($m_q \leq 1$) in queue occur due to benchmarks running in isolation, since they do not have to share multiple resources at the same time. It is believed that within a full-

system simulation, more requests would have to be generated, occupying more slots in queues.

It is also important to highlight the use of the extra queue. In all tests run, only two benchmarks made use of it, cases of `quicksort` and `sphinx`. This was a consequence of the large number of generated main memory write requests. Since the use of extra queue triggers when write queue becomes full, these applications were more prone to occupy all slots of the write queue. The explanation for `quicksort` having the largest occupation of extra queue is explained due to more write requests being generated in a smaller period of time – the same logic that explains why it has also the longest busy period.

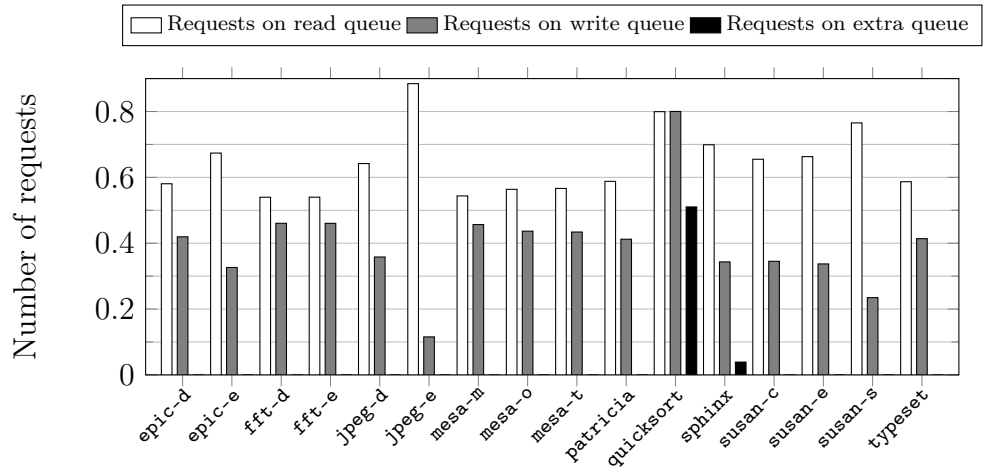


Figure 20 – Average number of requests in memory controller queues while on busy state

5.4.4 Additional remarks

With all presented behavioral analysis over the implemented memory controller, some extra considerations may be exposed. As it was noted, analysis over busy periods and queue use were done considering only the use of STT-RAM as a main memory, and fixed request queue sizes. It could be possible to perform analysis considering mixed configurations on different aspects of the memory controller. Below, some tests that were discarded from this work are explained:

- Impact of different NVMs in busy period length: Even though the absolute lengths of busy periods vary according to different NVMs used (due to different latencies when performing memory operations), its percentages which are calculated considering the total benchmark simulation time had negligible differences.

- Impact of different queue sizes: Some tests were done considering the use of smaller and bigger queues, having the default queue sizes as base (16 requests in read/write queues, 32 in the extra queue). By using powers of 2, simulations

were performed where the read/write queue sizes ranged from 4 to 128. Since all but two simulated benchmarks did not make use of the extra queue, even using 4-request read/write queues, this study was discarded. This occurred due to low generation of memory requests, which is backed up by results explained in 5.4.3 since it did not have enough applications to make a representative analysis.

5.5 Comparison between different memory controllers

After the behavioral analysis of the implemented memory controller, a comparison was performed with other memory controllers – all provided by NVMain. This was done in order to check the overall simulated performance of the implemented controller.

The implemented memory controller was compared with three different controllers already given by NVMain. All of the simulated controllers were tested using the studied NVM technologies as a main memory. The chosen controllers were:

- FCFS (First-come First-served): After each read or write is issued, the page is closed. The oldest request is attended first. It uses a unique queue that can hold both write and read requests;
- FRFCFS (First-ready First-come First-served): Prioritizes starved requests, then row buffer hits, and then tries oldest requests. The queue used can also hold both write and read requests;
- FRFCFS-WQF: (First-ready First-come First-served with Write Queue): Works similarly to FRFCFS. However, it uses two separated queues, where one is used to attend read requests, while the other one serves write requests. Write requests are attended when the write queue becomes full, emptying it so that it can keep more requests.

NVMain also includes additional memory controllers which were not considered to be included in this simulation, since they are not suitable for working as main memory controllers. The only memory controller which could be evaluated would be PerfectMemory, which would simulate a theoretical memory system with the smallest latency and energy cost possible. This controller could be used as the upper bound of performance. However, the current version of PerfectMemory does not execute properly – a segmentation fault results from the execution of any simulation.

All benchmarks were executed using the four memory controllers (FCFS, FRFCFS, FRFCFS-WQF and the implemented by this work) under the three different configurations of NVM (PCRAM, STT-RAM and RRAM). For the next charts, the Y-axis represents the latency in nanoseconds of a memory operation, whereas the X-axis shows the simulated benchmarks. Each bar represents a different memory controller used

in a simulation. The controller implemented by this work is labeled as **Custom**, while other controllers are labeled by their acronyms.

5.5.1 PCRAM

Figure 21 presents the average of read latencies over different memory controllers using a PCRAM as main memory. The chart is split in two charts, where Figure 21a shows benchmarks that did not generate main memory write requests, whereas Figure 21b displays benchmarks that generated main memory write requests.

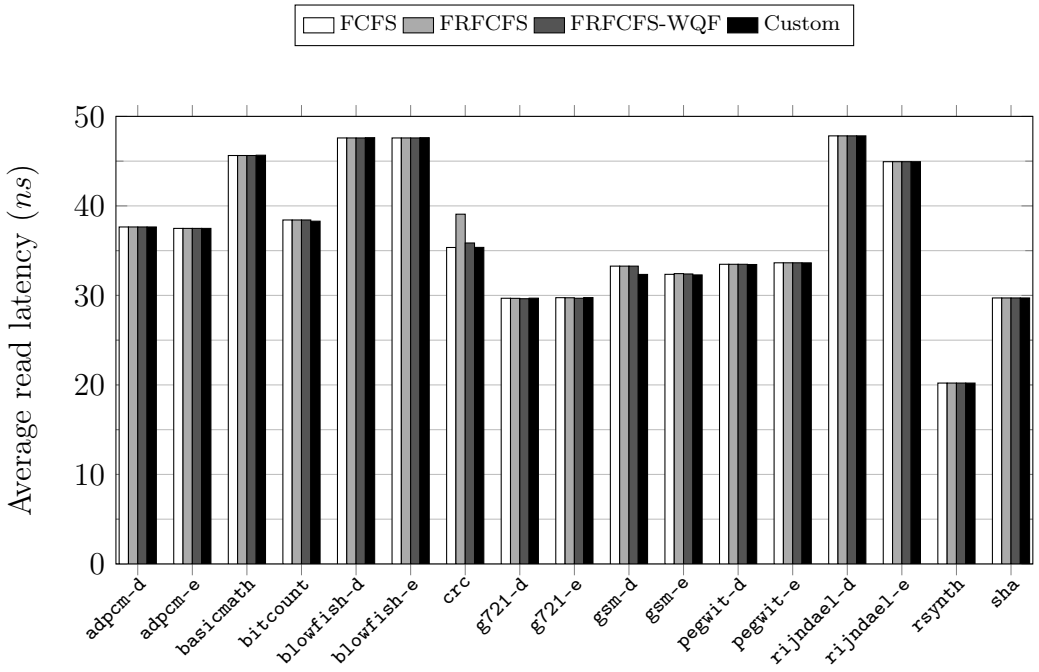
It is seen that in benchmarks with no main memory write operations, the memory controller used made negligible difference. This occurred since all applications shown in Figure 21a did not generate a large number of main memory requests ($\leq 50,000$), which translates into less work for the memory controller during the execution of these applications. Hence, the average read latency of these benchmarks depends directly by their nature.

When analyzing applications that have generated at least one main memory request, similar read latencies occur on benchmarks `dijkstra`, `mad` and `stringsearch`. These applications also have generated a lower number of memory requests, which explains the same pattern found on previous benchmarks. When observing remaining applications (which are the ones that generated more memory requests), the FRFCFS-WQF controller had the fastest read latencies. FRFCFS-WQF has a special buffer that stores write requests and serves them when the queue is full, which helped the boost in read latencies, since they do not depend of possible write requests to finish, as it occurs in the Custom controller. The remaining controllers had similar average read latencies, meaning that using the Custom controller with two different buffers for reads and writes had similar performance as controllers with unified request buffers (FCFS and FRFCFS).

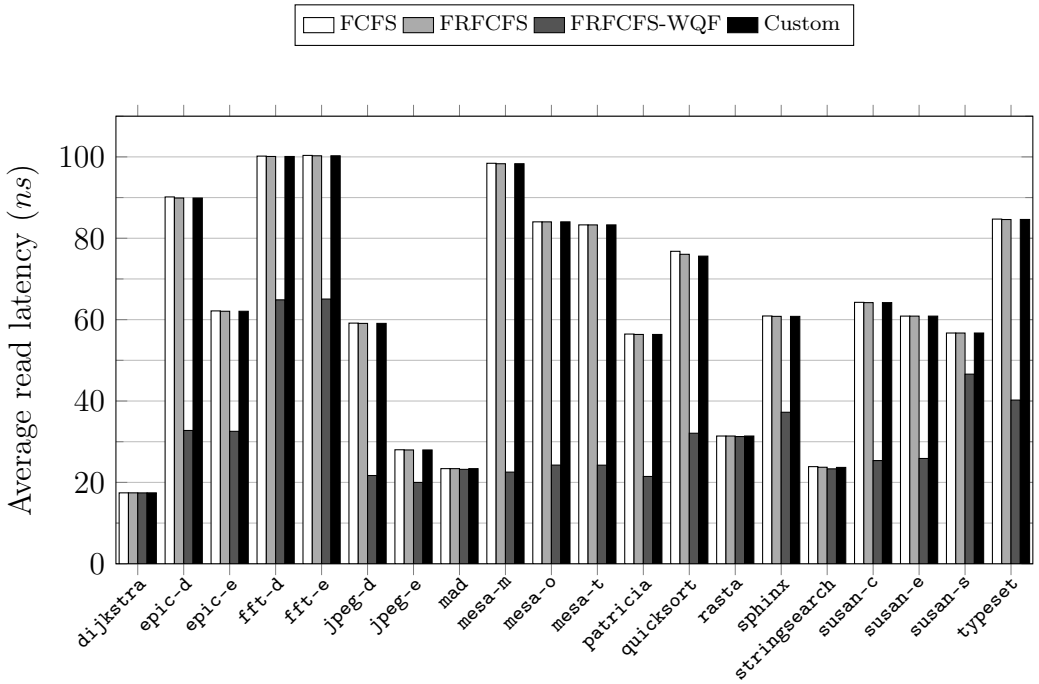
Figure 22 shows the average of write latencies over memory controller using PCRAM as main memory. When evaluating all controllers but FRFCFS-WQF, they had very similar write latencies, which is a similar behavior found in the previous analysis of read latencies. This means that even by using a different memory operation scheduling for the Custom controller, it does not make a difference when comparing with both FCFS and FRFCS controllers.

On the other hand, FRFCFS-WQF reached the slowest write latencies in all applications simulated. This occurs due to a trade-off effect: with the use of a write request queue, an improvement on read latencies over all controllers was observed. However, this comes with the cost with a increase on write latency.

By using FRFCFS-WQF as the memory controller, it is observed that the applications had an overall better performance when comparing to other controllers. Since in all simulated benchmarks the number of read operations is greater than the number



(a) Latencies of read memory operations in benchmarks that did not generate main memory write requests



(b) Latencies of read memory operations in benchmarks that generated main memory write requests

Figure 21 – Latencies of read memory operations with PCRAM as main memory in multiple memory controllers

of write operations, this strategy helped on the overall performance of the benchmark when using the FRFCFS-WQF controller.

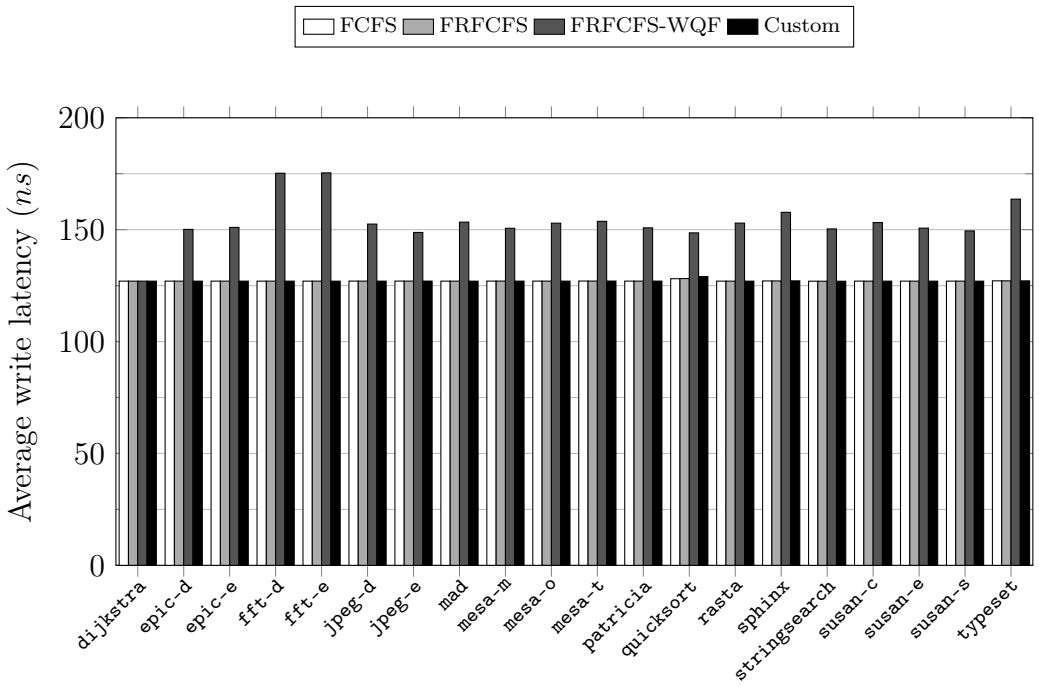


Figure 22 – Latencies of write memory operations with PCRAM as main memory on multiple memory controllers

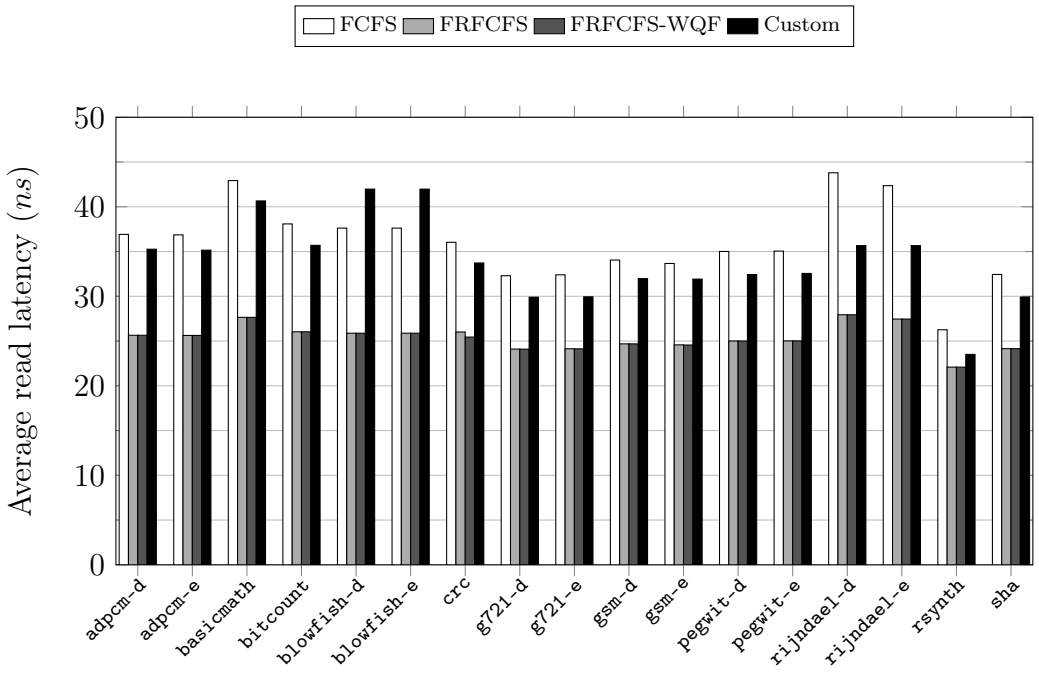
5.5.2 RRAM

The next analysis of different memory controllers was done using an RRAM as main memory. Figure 23 presents the average of read latencies over different memory controllers. This Figure is also split in two charts, where Figure 23a shows benchmarks with did not generate main memory write requests, whereas Figure 23b displays benchmarks that generated main memory write requests.

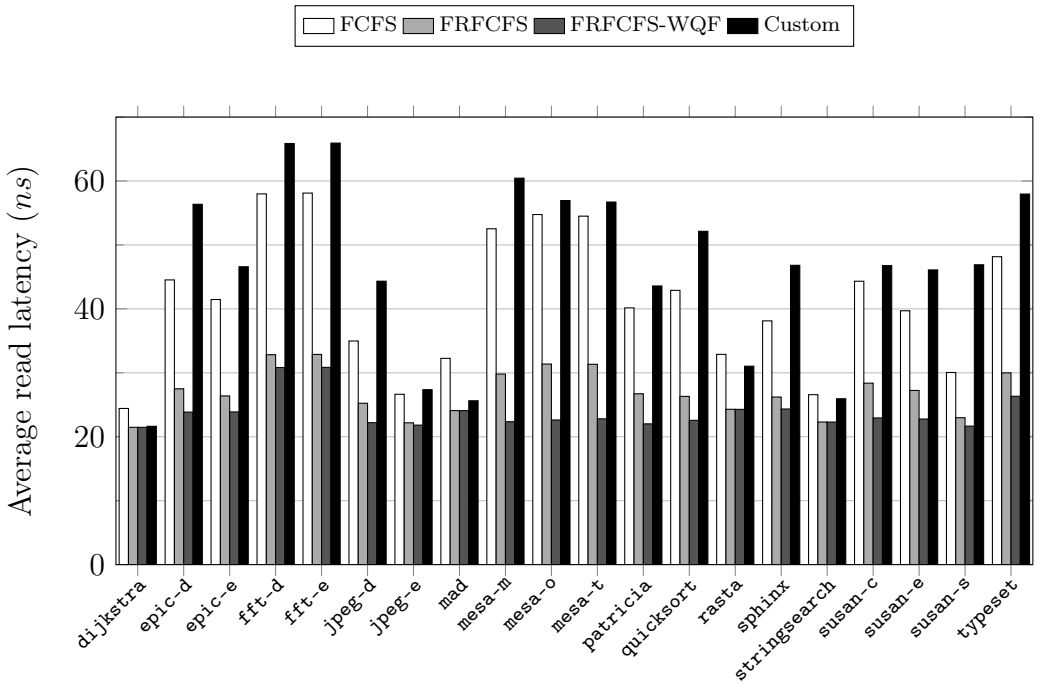
On benchmarks shown in Figure 23a, there are differences on the patterns when comparing with PCRAM. In this case, it can be observed that both controllers FRFCFS and FRFCFS-WQF had similar latencies in all benchmarks simulated. These can be explained due to the absence of write main memory requests, as these controllers have different policies for those types of requests. In addition to that, both FRFCFS and FRFCFS-WQF prioritize row-buffer hits, which occurred on RRAM more frequently than on PCRAM, whereas controllers FCFS and Custom prioritizes only oldest requests on queue.

When analyzing Figure 23b, the worst read latencies occur in both FCFS and Custom. In benchmarks `dijkstra` and `mad`, read latencies when simulating Custom controller reaches values close to FRFCFS and FRFCFS-WQF, which occurred due to being applications that generated a small number of main memory requests – this

same behavior occurred on PCRAM simulations.



(a) Latencies of read memory operations in benchmarks that did not generate main memory write requests



(b) Latencies of read memory operations in benchmarks that generated main memory write requests

Figure 23 – Latencies of read memory operations with RRAM as main memory in multiple memory controllers

In addition, the impact of FRFCFS-WQF on RRAM read latencies which was more evident in PCRAM appears on some applications here. Applications that generated more memory requests tend to be more impacted by FRFCFS-WQF, reducing read

latencies in comparison with FRFCFS. This can be seen specially on all stages of benchmark `mesa` – from FRFCFS-WQF to FRFCFS, a read latency reduction of $\approx 8ns$ can be seen.

Figure 24 shows the average of write latencies over memory controller using RRAM as main memory. In this case, Custom controller reached the slowest latencies in all simulated benchmarks, with an average value of $70.9ns$. Regarding other controllers, no pattern could be observed, since the results showed variable values depending on each benchmark simulated.

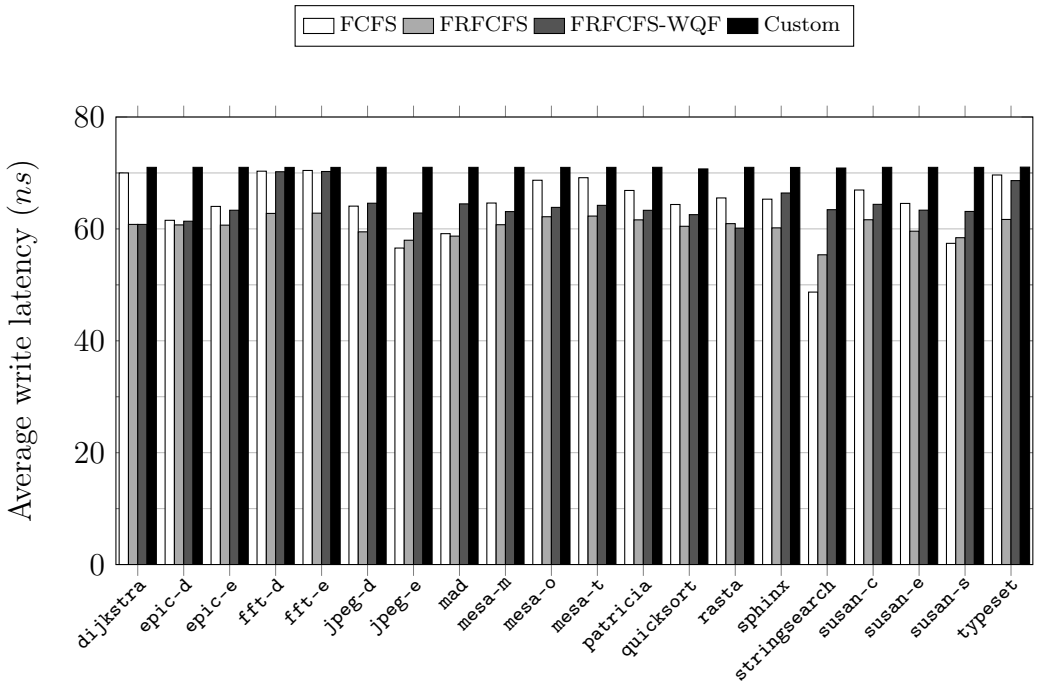


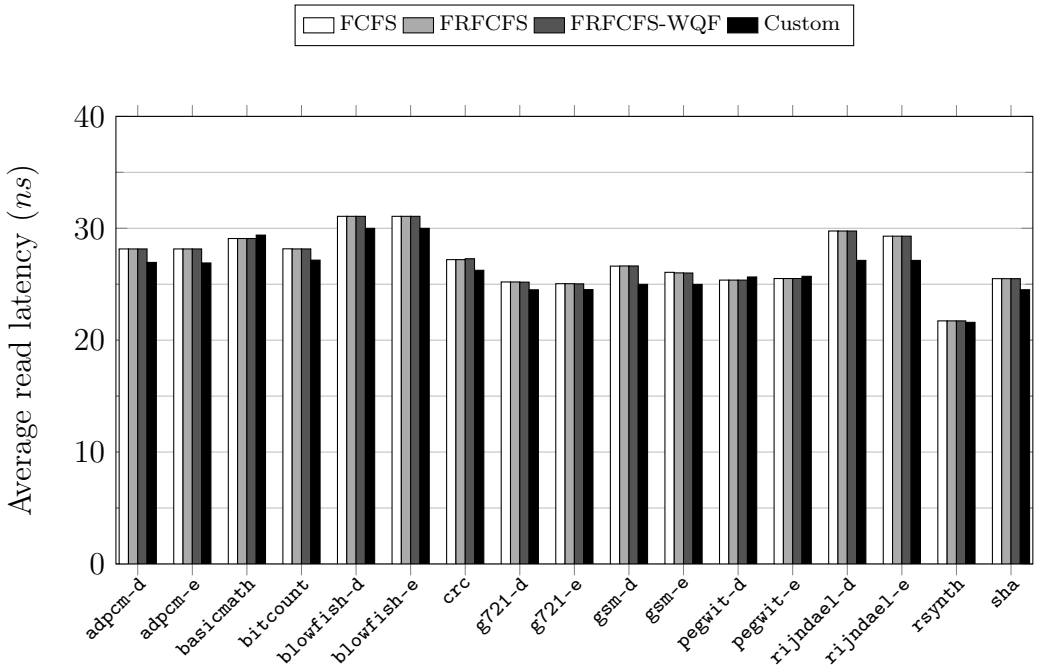
Figure 24 – Latencies of write memory operations with RRAM as main memory on multiple memory controllers

5.5.3 STT-RAM

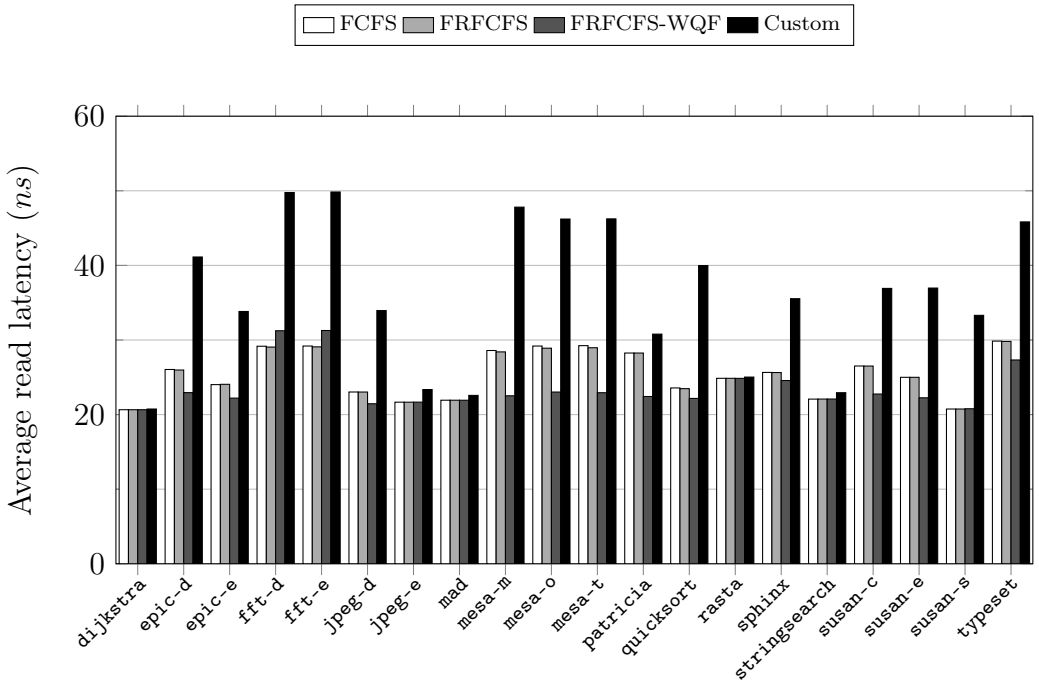
The last evaluation of different memory controllers was done using an STT-RAM as main memory. Figure 25 presents the average of read latencies over different memory controllers. The chart is also split in two, where Figure 25a shows benchmarks that did not generate main memory write requests, whereas Figure 25b displays benchmarks that generated main memory write requests.

In Figure 25a, the default controllers of NVMain had similar latencies on all applications with no main memory write requests. Since the simulated benchmarks generate less than 50,000 memory requests, the impact of the memory controller used is negligible – a small but notable variation is only observed when using the Custom memory controller.

This same pattern is seen on benchmarks that generated a small number of mem-



(a) Latencies of read memory operations in benchmarks that did not generate main memory write requests



(b) Latencies of read memory operations in benchmarks that generated main memory write requests

Figure 25 – Latencies of read memory operations with STT-RAM as main memory in multiple memory controllers

ory requests in Figure 25b (`dijkstra`, `mad` and `stringsearch`). In benchmarks that generated more main memory requests, the Custom controller reached the worst read latencies, while the other controllers had similar latencies – best results occurred on FCFS or FRFCFS-WQF, depending on the benchmark.

Figure 26 shows the average of write latencies over different memory controllers using STT-RAM as a main memory. In this case, FCFS and FRFCFS had similar results independently of the application simulated, and they reached the fastest write latencies. When simulating either FRFCFS-WQF or Custom controllers, slower write latencies occurred. This could be a consequence of the specific treatment for write operations on both controllers – STT-RAM was the technology that had the smallest gap between read and write latencies.

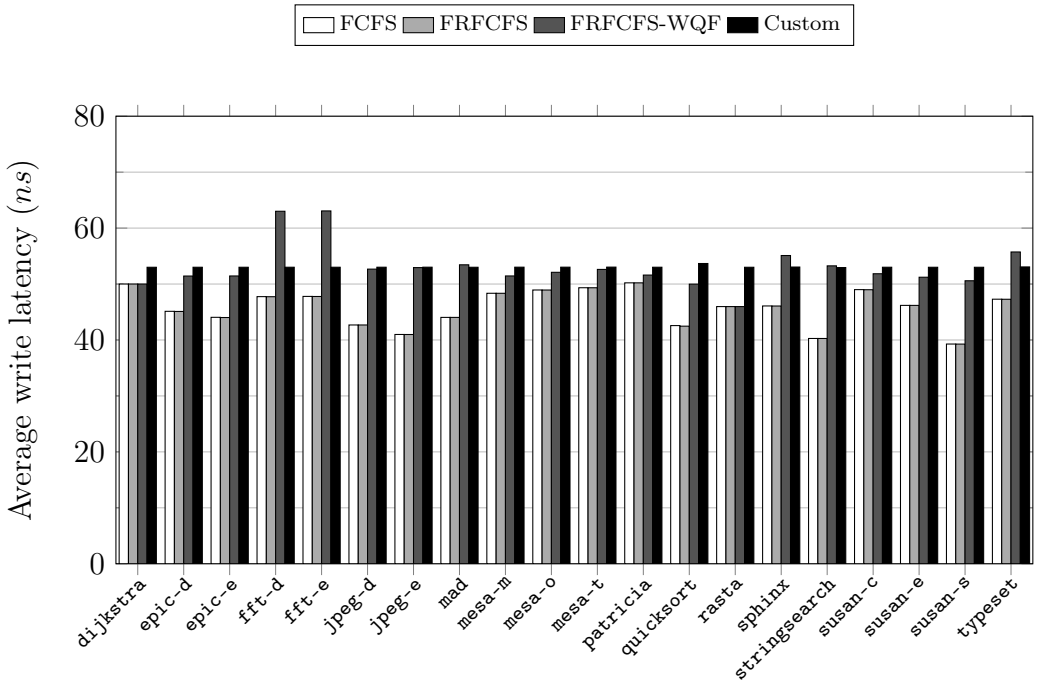
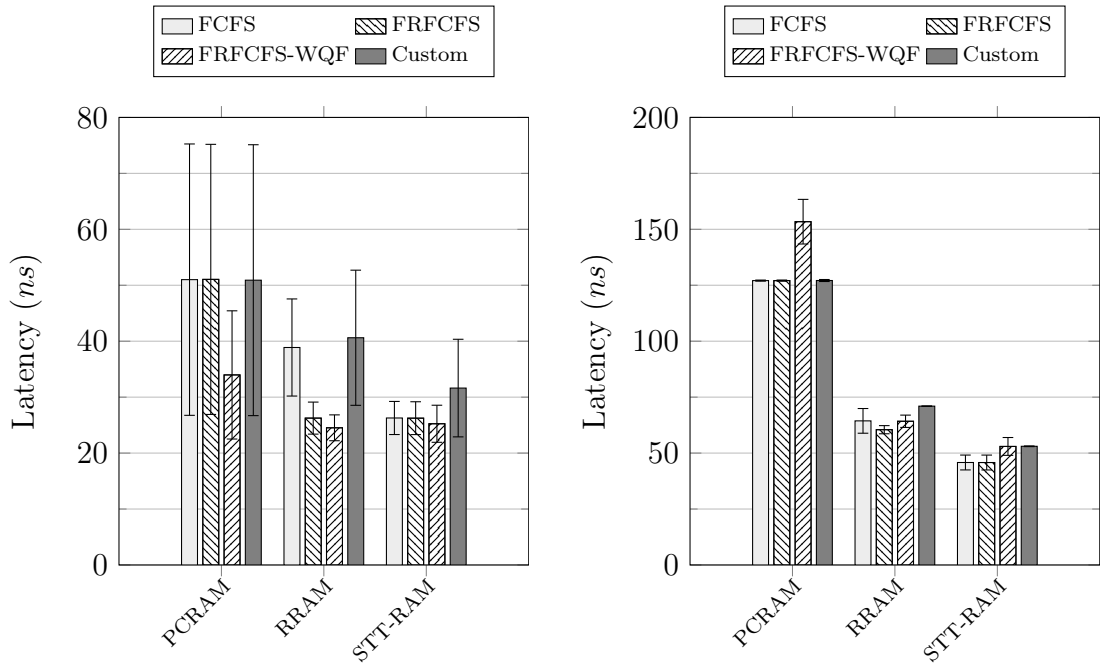


Figure 26 – Latencies of write memory operations with STT-RAM as main memory on multiple memory controllers

5.5.4 Overall Analysis

Lastly, Figure 27 presents the average of both read and write total latencies over different memory controllers, considering the use of different NVMs. In Figure 27a, it is shown the calculated average of all read memory operation latencies over all simulated applications, while in Figure 27b the average of write memory operation latencies of remaining applications (which generated at least one main memory write request) is shown.

In overall, read latencies in any NVM/memory controller had very variable values between applications, as it can be seen by its calculated standard deviations and pre-



(a) Average read latencies

(b) Average write latencies

Figure 27 – Latencies of memory operations in different NVM technologies and memory controllers

vious charts. On PCRAM, the FRFCFS-WQF controller scheduling policy helped to boost read latencies of many benchmarks in detriment of write latencies. However, on average no controller can be defined as the best – by running a t test with a 95% confidence interval and comparing the means presented on the chart, it was observed that all PCRAM read latency means do not have significant differences.

As seen previously, the Custom memory controller interfered in read latencies, independently of the NVM used as main memory. This variation is seen in a minor scale when simulating RRAM with both FRFCFS and FRFCFS-WQF, and the already established memory controllers in STT-RAM. Hence, the addition of more scheduling techniques to the Custom Controller can help boost read latencies and reduce their variations between benchmarks which got worsened when comparing with other memory controllers.

Regarding write latencies, it is seem that in PCRAM the Custom memory controller have at least matched the best overall results, which are found in FCFS and FRFCFS. It also surpassed FRFCFS-WQF, that was the worst average write latency – which occurred due to the scheduling that attended read requests more quickly. On the other hand, when observing RRAM and STT-RAM latencies, it did not achieve best latencies. However, it is remarkable that average write latencies along the benchmark runs were more uniform than in already established controllers, with small standard deviations in the implemented memory controller.

5.6 Conclusion

This chapter presented all results obtained from multiple experiments performed by this work. In order to extract results, sets of applications were chosen, being MediaBench and MiBench. Sanity checks were performed to ensure that benchmarks could be compiled and executed. Then, a profiling of memory requests on benchmarks which could be executed was done.

Then, a behavioral analysis on the implemented memory controller was done, by evaluating length of busy periods, tests over different NVMs and use of queues. A comparison between the implemented controller and NVMain default controllers was performed, where both read and write latencies were analyzed. This comparison was done considering each benchmark simulated. A final analysis was done calculating the average and standard deviation of all benchmarks on each NVM and each memory controller – in this case, also both read and write latencies were the comparison factors.

6 CONCLUSION

This work proposed a evaluation of different memory controllers directed to NVMs. In order to cope with asymmetry of write and read memory of NVMs, it is important to schedule memory requests properly. In order to do that, firstly a study over NVM and memory controllers was performed. This was done in a sense of knowing current NVMs which were most studied on the academia, and how memory controllers could aid to extract best performance from NVM-based systems. With that considered, an implementation of a memory controller using an already-proposed scheduling policy was showed in this work.

For the purpose of implement the controller, it was decided to use two tools: Gem5 and NVMain, which are, respectively, a general-purpose simulator and a NVM simulator. They were chosen since the combination of both tools can provide cycle-accurate and near-full-system simulation. With tools chosen, a different memory controller was implemented in NVMain memory-system. Its algorithms were detailed as they were coded inside the memory simulator.

After coding the controller into NVMain, the next step of this work was to perform its evaluation. For this part, sets of benchmarks were chosen to run and test the custom controller, which were MediaBench and MiBench. Their applications were tested and got their data inputs expanded in order to generate more main memory operations, which allowed to perform analysis over them. With that, a benchmark profiling was done over memory request generations, finding out what applications generated small or big quantities of memory requests.

Then, a behavioral analysis of the memory controller was done. Firstly, the memory controller was tested using three different NVMs as main memories, being those PCRAM, RRAM and STT-RAM. The tests run had expected results accordingly to the distinct NVM technologies used, even though read latencies had great variations between benchmark executions. Afterwards, an evaluation of the busy periods of the benchmarks in the memory controller was performed, where the main result found was that on simulated benchmarks, the average of time spent in busy state was small, which did not surpass 2% of the total execution time. Thus, the time spent attending

and generating memory requests was considered small in executed benchmarks.

Another analysis presented by this work was regarding the use of queues in the implemented memory controller, one of its key elements. Simulations run showed that queues were, in overall, underused. In average, the applications do not keep more than one request in queue when the memory controller is active. This probably happened due to benchmarks running in isolation, which did not let generation of large number of memory requests, and that resource sharing was nonessential. Additional analysis were also done: When simulating NVMs, the length of busy periods in executed benchmarks had negligible differences; Simulation of queues with different capacities made no difference, since queues in average were underused, as already observed.

Lastly, a comparison between the implemented memory controller with already-established controllers provided by NVMain. The results were done evaluating performance on latencies of read and write operations, which are important factors of NVMs. In overall, read latencies in tested memory controllers had very variable results, while write latencies tended to more uniform values, independently of the benchmark executed.

When using the custom memory controller with STT-RAM as main memory, write latencies reached the best results, matching the latencies of already-established NVMain controllers. While that STT-RAM showed on overall the best write latencies, it is important to point that using it on main memory requires larger chips – and bigger STT-RAM chips have worse properties when comparing with smaller chips. For instance, this impact seen on both STT-RAM and RRAM is less seen on the developing of large PCRAM chips.

On other cases, this work concludes that the custom controller may need improvements or consider other memory scheduling techniques in order to obtain better performance. Other possibility lies in the adoption of hybrid architectures, which combines both volatile and non-volatile technologies to compose a memory system, which can potentially lead to extract the better features that each type of memory can offer.

6.1 Future work

There are plenty of future work to do starting from this work. One may include the evaluation of a full-system simulation using the powerful emulation of Gem5, combining it with NVMain main memory subsystem. This could extend this work, since all of the presented analysis here was done with benchmarks running in isolation. In order to do that, many other challenges and variables will need to be considered, turning this evaluation a more powerful and complex one. With the full-system simulation, an operational system needs to be present in the executions, which can potentially insert more pressure on the main memory controller. In addition to that, more processes will

have to dispute for resources, which may reflect a more realistic scenario.

This analysis could be extended to use test with other sets of benchmarks. These could be aimed to test a specific type of applications, e.g., image and multimedia applications. Other tests can be performed with applications that have different intentions – for example, general-purpose benchmarks, such as SPEC-CPU (SPEC, 2017). In a similar context, other evaluations can be accomplished by testing other emerging memory technologies which show promising results. Analysis could be performed in the sense of evaluating the impact of different memory controllers on different emerging technologies, which can include 3D memories (HOUDT, 2017), racetrack memories (ZHANG et al., 2016), among others.

Since the use of NVMs on memory systems show issues, another possibility lies in the adoption of hybrid architectures. These combine the use of both memory technologies (volatile and non-volatile) in a memory system so that it can extract the best of each type of memory. The development of a memory controller in a hybrid system must be efficient and adaptive to different scenarios, e.g., it should detect when it is more advantageous to use a non-volatile memory instead of a volatile one.

Other possible future work includes the improvement of memory controller directed to NVMs. Related work shows different ways of dealing with memory operations (specially write ones), which could aid the controller implemented here in order to obtain an overall better performance. Examples of helpful techniques can include write cancellation and write pausing (QURESHI; FRANCESCHINI; LASTRAS-MONTAÑO, 2010) and increasing of write concurrency in multiple banks of a memory (LAI et al., 2015).

REFERENCES

AKINAGA, H.; SHIMA, H. ReRAM technology; challenges and prospects. **IEICE Electronics Express**, Tokyo, Japan, v.9, n.8, p.795–807, 2012.

ARJOMAND, M. et al. HL-PCM: MLC PCM Main Memory with Accelerated Read. **IEEE Transactions on Parallel and Distributed Systems**, New York, NY, USA, v.28, n.11, p.3188–3200, Nov 2017.

AWAD, A. et al. Silent Shredder: Zero-Cost Shredding for Secure Non-Volatile Main Memory Controllers. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.50, n.2, p.263–276, March 2016.

BINKERT, N. **public/gem5 – Git at google**. 2012, Available at: <https://gem5.googlesource.com/public/gem5>. Access on: March, 2018.

BINKERT, N. et al. The Gem5 Simulator. **SIGARCH Comput. Archit. News**, New York, NY, USA, v.39, n.2, p.1–7, Aug. 2011.

BURNS, A.; GUTIéRREZ, M.; RIVAS, M. A.; HARBOUR, M. G. A Deadline-Floor Inheritance Protocol for EDF Scheduled Embedded Real-Time Systems with Resource Sharing. **IEEE Transactions on Computers**, New York, NY, USA, v.64, n.5, p.1241–1253, May 2015.

BURR, G. W. et al. Recent Progress in Phase-Change Memory Technology. **IEEE Journal on Emerging and Selected Topics in Circuits and Systems**, New York, NY, USA, v.6, n.2, p.146–162, June 2016.

CHANG, C. W.; YANG, C. Y.; CHANG, Y. H.; KUO, T. W. Booting Time Minimization for Real-Time Embedded Systems with Non-Volatile Memory. **IEEE Transactions on Computers**, New York, NY, USA, v.63, n.4, p.847–859, April 2014.

CHI, P. et al. PRIME: A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory. **SIGARCH Comput. Archit. News**, New York, NY, USA, v.44, n.3, p.27–39, June 2016.

CHI, P.; LEE, W.-C.; XIE, Y. Making B^+-tree Efficient in PCM-based Main Memory. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, 2014, La Jolla, California, USA. **Proceedings...** ACM, 2014. p.69–74. (ISLPED '14).

CHOI, Y. et al. A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth. In: IEEE INTERNATIONAL SOLID-STATE CIRCUITS CONFERENCE, 2012, San Francisco, CA, USA. **Proceedings...** IEEE, 2012. p.46–48.

CHUNG, H. et al. A 58nm 1.8V 1Gb PRAM with 6.4MB/s program BW. In: IEEE INTERNATIONAL SOLID-STATE CIRCUITS CONFERENCE, 2011., 2011, San Francisco, CA, USA. **Proceedings...** IEEE, 2011. p.500–502.

COALMON, B. **Theory of Spin Transfer Torque**. 2009. Available at: <https://www.nist.gov/programs-projects/theory-spin-transfer-torque>. Access on: March, 2018.

DASARI, D.; NELIS, V.; MOSSE, D. Timing analysis of PCM main memory in multicore systems. In: IEEE 19TH INTERNATIONAL CONFERENCE ON EMBEDDED AND REAL-TIME COMPUTING SYSTEMS AND APPLICATIONS, 2013, Taipei, Taiwan. **Proceedings...** IEEE, 2013. p.52–61.

DONG, X.; XIE, Y. AdaMS: Adaptive MLC/SLC phase-change memory design for file storage. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE (ASP-DAC 2011), 16., 2011, Yokohama, Japan. **Proceedings...** IEEE, 2011. p.31–36.

DU, Y. et al. Bit Mapping for Balanced PCM Cell Programming. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 2013, Tel-Aviv, Israel. **Proceedings...** ACM, 2013. p.428–439. (ISCA '13).

ENDOH, T. et al. An Overview of Nonvolatile Emerging Memories – Spintronics for Working Memories. **IEEE Journal on Emerging and Selected Topics in Circuits and Systems**, New York, NY, USA, v.6, n.2, p.109–119, June 2016.

EVERSPIN. **Datasheet**: MR2A16A Family. 2015.

EWAIS, M. A.; OMRAN, M. A.; RAAFAT, A.; ALKABANI, Y. A virtual memory architecture to enhance STT-RAM performance as main memory. In: IEEE CANADIAN CONFERENCE ON ELECTRICAL AND COMPUTER ENGINEERING (CCECE), 2016, Vancouver, Canada. **Proceedings...** IEEE, 2016. p.1–6.

FUJITA, M. Highly-Pipelined and Energy-Saved Computing with Arrays of Non-Volatile Memories. In: INTERNATIONAL CONFERENCE ON INTERDISCIPLINARY ADVANCES IN APPLIED COMPUTING, 2014, Amritapuri, India. **Proceedings...** ACM, 2014. p.46:1–46:6. (ICONIAAC '14).

GOOSSENS, S.; CHANDRASEKAR, K.; AKESSON, B.; GOOSSENS, K. **Memory Controllers for Mixed-Time-Criticality Systems**: Architectures, Methodologies and Trade-offs. New York, NY, USA: Springer, 2016.

GUTHAUS, M. R. et al. MiBench: A free, commercially representative embedded benchmark suite. In: FOURTH ANNUAL IEEE INTERNATIONAL WORKSHOP ON WORKLOAD CHARACTERIZATION. WWC-4 (CAT. NO.01EX538), 2001, Austin, TX, USA. **Proceedings...** IEEE, 2001. p.3–14.

HOUDT, J. V. 3D Memories and Ferroelectrics. In: IEEE INTERNATIONAL MEMORY WORKSHOP (IMW), 2017., 2017, Monterey, CA, USA. **Proceedings...** IEEE, 2017. p.1–3.

HU, J. et al. Scheduling to Optimize Cache Utilization for Non-Volatile Main Memories. **IEEE Transactions on Computers**, New York, NY, USA, v.63, n.8, p.2039–2051, Aug 2014.

IELMINI, D. Resistive switching memories based on metal oxides: mechanisms, reliability and scaling. **Semiconductor Science and Technology**, Bristol, UK, v.31, n.6, p.063002, 2016.

KANG, S. H. Embedded stt-mram for mobile applications: Enabling advanced chip architectures. In: NON-VOLATILE MEMORIES WORKSHOP, 2010, San Diego, CA, USA. **Proceedings...** CMRR, 2010.

KANG, W. et al. Yield and Reliability Improvement Techniques for Emerging Nonvolatile STT-MRAM. **IEEE Journal on Emerging and Selected Topics in Circuits and Systems**, New York, NY, USA, v.5, n.1, p.28–39, March 2015.

KAWAHARA, A. et al. An 8Mb multi-layered cross-point ReRAM macro with 443MB/s write throughput. In: IEEE INTERNATIONAL SOLID-STATE CIRCUITS CONFERENCE, 2012, San Francisco, CA, USA. **Proceedings...** IEEE, 2012. p.432–434.

KHWA, W. S. et al. A Procedure to Reduce Cell Variation in Phase Change Memory for Improving Multi-Level-Cell Performances. In: IEEE INTERNATIONAL MEMORY WORKSHOP (IMW), 2015, San Francisco, CA, USA. **Proceedings...** IEEE, 2015. p.1–4.

KIM, H.; KIM, S.; LEE, J. Write-Amount-Aware Management Policies for STT-RAM Caches. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, New York, NY, USA, v.25, n.4, p.1588–1592, April 2017.

KIM, Y.; HAN, D.; MUTLU, O.; HARCHOL-BALTER, M. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In: HPCA - 16 2010

THE SIXTEENTH INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE, 2010, Bangalore, India. **Proceedings...** IEEE, 2010. p.1–12.

KOTRA, J. B. et al. Re-NUCA: A Practical NUCA Architecture for ReRAM based last-level caches. In: PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, 2016 IEEE INTERNATIONAL, 2016, Chicago, IL, USA. **Proceedings...** IEEE, 2016. p.576–585.

KÜLTÜRSAY, E.; KANDEMIR, M.; SIVASUBRAMANIAM, A.; MUTLU, O. Evaluating STT-RAM as an energy-efficient main memory alternative. In: PERFORMANCE ANALYSIS OF SYSTEMS AND SOFTWARE (ISPASS), 2013 IEEE INTERNATIONAL SYMPOSIUM ON, 2013, Austin, TX, USA. **Proceedings...** IEEE, 2013. p.256–267.

KUMAR, P. S.; KARSAI, G. Integrated Analysis of Temporal Behavior of Component-Based Distributed Real-Time Embedded Systems. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT/COMPONENT/SERVICE-ORIENTED REAL-TIME DISTRIBUTED COMPUTING WORKSHOPS, 2015., 2015. **Proceedings...** IEEE, 2015. p.50–57.

LAI, C. H.; YU, S. C.; YANG, C. L.; LI, H. P. Fine-grained write scheduling for PCM performance improvement under write power budget. In: IEEE/ACM INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN (ISLPED), 2015, Rome, Italy. **Proceedings...** IEEE, 2015. p.19–24.

LEE, C.; POTKONJAK, M.; MANGIONE-SMITH, W. H. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 30., 1997, Research Triangle Park, NC, USA. **Proceedings...** IEEE, 1997. p.330–335.

LEE, S. H. Technology scaling challenges and opportunities of memory devices. In: IEEE INTERNATIONAL ELECTRON DEVICES MEETING (IEDM), 2016, Monterey, CA, USA. **Proceedings...** IEEE, 2016. p.1.1.1–1.1.8.

LI, B.; SHAN, S.; HU, Y.; LI, X. Partial-SET: Write speedup of PCM main memory. In: DESIGN, AUTOMATION TEST IN EUROPE CONFERENCE EXHIBITION (DATE), 2014., 2014, Dresden, Germany. **Proceedings...** IEEE, 2014. p.1–4.

LI, H. H. et al. Looking Ahead for Resistive Memory Technology: A broad perspective on ReRAM technology for future storage and computing. **IEEE Consumer Electronics Magazine**, New York, NY, USA, v.6, n.1, p.94–103, Jan 2017.

LI, Q. et al. Compiler-Assisted Refresh Minimization for Volatile STT-RAM Cache. **IEEE Transactions on Computers**, New York, NY, USA, v.64, n.8, p.2169–2181, Aug 2015.

MARTINEZ, J. F.; IPEK, E. Dynamic Multicore Resource Management: A Machine Learning Approach. **IEEE Micro**, New York, NY, USA, v.29, n.5, p.8–17, Sep 2009.

MEENA, J. S.; SZE, S. M.; CHAND, U.; TSENG, T.-Y. Overview of emerging nonvolatile memory technologies. **Nanoscale research letters**, New York, NY, USA, v.9, n.1, p.1, 2014.

MENG, Y. et al. Uniform silicon carbide doped Sb 2 Te nanomaterial for high temperature and high speed PCM applications. **Journal of Alloys and Compounds**, Lausanne, Switzerland, v.664, p.591–594, 2016.

MITTAL, S.; VETTER, J. S. A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems. **IEEE Transactions on Parallel and Distributed Systems**, New York, NY, USA, v.27, n.5, p.1537–1550, May 2016.

MITTAL, S.; VETTER, J. S.; LI, D. A Survey Of Architectural Approaches for Managing Embedded DRAM and Non-Volatile On-Chip Caches. **IEEE Transactions on Parallel and Distributed Systems**, New York, NY, USA, v.26, n.6, p.1524–1537, June 2015.

MUTLU, O.; SUBRAMANIAN, L. Research problems and opportunities in memory systems. **Supercomputing frontiers and innovations**, Chelyabinsk, Russia, v.1, n.3, p.19–55, 2015.

NUMONYX. **The Basics of Phase Change Memory Technology**. 2007. Available at: <http://signallake.com/innovation/PhaseChangeMemory.pdf>. Access on: March, 2018.

OIKE, H. et al. Phase-change memory function of correlated electrons in organic conductors. **Physical Review B**, Ridge, NY, USA, v.91, n.4, p.041101, 2015.

OUKID, I.; KETTLER, R.; WILLHALM, T. Storage class memory and databases: Opportunities and challenges. **it-Information Technology**, Berlin, Germany, v.3, n.59, p.109–115, March 2017.

PAN, F. et al. Recent progress in resistive random access memories: materials, switching mechanisms, and performance. **Materials Science and Engineering: R: Reports**, Lausanne, Switzerland, v.83, p.1–59, 2014.

PEREZ, T.; DE ROSE, C. **Non-Volatile Memory**: Emerging Technologies And Their Impacts on Memory Systems (TR-060). Porto Alegre, Brazil: Technical report, Faculdade de Informática, Pontifıcia Universidade Católica do Rio Grande do Sul (PUCRS), 2010.

PIROVANO, A. An Introduction on Phase-Change Memories. In: **Phase Change Memory**. New York, NY, USA: Springer, 2018. p.1–10.

POREMBA, M. **mrp5060 / nvmain**. 2012, Available at: <https://bitbucket.org/mrp5060/nvmain/overview>. Access on: March, 2018.

POREMBA, M. **NVMain Wiki Main/Home Page**. 2015, Available at: <http://wiki.nvmain.org/>. Access on: March, 2018.

POREMBA, M.; XIE, Y. NVMain: An Architectural-Level Main Memory Simulator for Emerging Non-volatile Memories. In: IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON VLSI, 2012, Amherst, MA, USA. **Proceedings...** IEEE, 2012. p.392–397.

POREMBA, M.; ZHANG, T.; XIE, Y. NVMain 2.0: A User-Friendly Memory Simulator to Model (Non-)Volatile Memory Systems. **IEEE Computer Architecture Letters**, New York, NY, USA, v.14, n.2, p.140–143, July 2015.

POURSHIRAZI, B.; ZHU, Z. Refree: A Refresh-Free Hybrid DRAM/PCM Main Memory System. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS), 2016, Chicago, IL, USA. **Proceedings...** IEEE, 2016. p.566–575.

QURESHI, M. K.; FRANCESCHINI, M. M.; LASTRAS-MONTAÑO, L. A. Improving read performance of Phase Change Memories via Write Cancellation and Write Pausing. In: HPCA - 16 2010 THE SIXTEENTH INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE, 2010, Bangalore, India. **Proceedings...** IEEE, 2010. p.1–11.

RAOUX, S.; XIONG, F.; WUTTIG, M.; POP, E. Phase change materials and phase change memory. **MRS bulletin**, Cambridge, UK, v.39, n.8, p.703–710, 2014.

SALEHI, M.; EJLALI, A. A Hardware Platform for Evaluating Low-Energy Multiprocessor Embedded Systems Based on COTS Devices. **IEEE Transactions on Industrial Electronics**, New York, NY, USA, v.62, n.2, p.1262–1269, Feb 2015.

SONG, B.; LI, Q.; LIU, H.; LIU, H. Exploration of selector characteristic based on electron tunneling for RRAM array application. **IEICE Electronics Express**, Tokyo, Japan, v.14, n.17, p.20170739–20170739, 2017.

SPEC. **Standard Performance Evaluation Corporation**. 2017. Available at: <https://www.spec.org/cpu2017/>. Access on: May, 2018.

THOMAS, L. et al. Perpendicular spin transfer torque magnetic random access memories with high spin torque efficiency and thermal stability for embedded applications. **Journal of Applied Physics**, Melville, NY, USA, v.115, n.17, p.172615, 2014.

WANG, F.; WU, X. Non-volatile Memory Devices Based on Chalcogenide Materials. In: INFORMATION TECHNOLOGY: NEW GENERATIONS, INTERNATIONAL CONFERENCE ON, 2009, Los Alamitos, CA, USA. **Proceedings...** IEEE, 2009. p.5–9.

WANG, K.; ALZATE, J.; AMIRI, P. K. Low-power non-volatile spintronic memory: STT-RAM and beyond. **Journal of Physics D: Applied Physics**, Philadelphia, PA, USA, v.46, n.7, p.074003, 2013.

WANG, Z. et al. Adaptive placement and migration policy for an STT-RAM-based hybrid cache. In: IEEE 20TH INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE (HPCA), 2014, Orlando, FL, USA. **Proceedings...** IEEE, 2014. p.13–24.

WONG, H. S. P. et al. Metal-Oxide RRAM. **Proceedings of the IEEE**, New York, NY, USA, v.100, n.6, p.1951–1970, June 2012.

YAKOPCIC, C.; HASAN, R.; TAHA, T. M. Hybrid crossbar architecture for a memristor based cache. **Microelectronics Journal**, Lausanne, Switzerland, v.46, n.11, p.1020 – 1032, 2015.

YAZDANSHENAS, S.; PIRBASTI, M. R.; FAZELI, M.; PATOOGHY, A. Coding last level STT-RAM cache for high endurance and low power. **IEEE computer architecture letters**, Washington, DC, USA, v.13, n.2, p.73–76, 2014.

YOUNG, V.; NAIR, P. J.; QURESHI, M. K. DEUCE: Write-Efficient Encryption for Non-Volatile Memories. **SIGARCH Comput. Archit. News**, New York, NY, USA, v.43, n.1, p.33–44, Mar. 2015.

ZHAN, J. et al. OSCAR: Orchestrating STT-RAM cache traffic for heterogeneous CPU-GPU architectures. In: ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE (MICRO), 49., 2016, Taipei, Taiwan. **Proceedings...** IEEE, 2016. p.1–13.

ZHANG, H.; XIAO, N.; LIU, F.; CHEN, Z. Leader: Accelerating ReRAM-based main memory by leveraging access latency discrepancy in crossbar arrays. In: DESIGN, AUTOMATION & TEST IN EUROPE CONFERENCE & EXHIBITION (DATE), 2016, Dresden, Germany. **Proceedings...** IEEE, 2016. p.756–761.

ZHANG, Y. et al. ADAMS: asymmetric differential STT-RAM cell structure for reliable and high-performance applications. In: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 2013, San Jose, CA, USA. **Proceedings...** IEEE Press, 2013. p.9–16.

ZHANG, Y. et al. Read Performance: The Newest Barrier in Scaled STT-RAM. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, New York, NY, USA, v.23, n.6, p.1170–1174, June 2015.

ZHANG, Y. et al. Perspectives of Racetrack Memory for Large-Capacity On-Chip Memory: From Device to System. **IEEE Transactions on Circuits and Systems I: Regular Papers**, New York, NY, v.63, n.5, p.629–638, May 2016.

ZHAO, J.; XU, C.; CHI, P.; XIE, Y. Memory and storage system design with non-volatile memory technologies. **IPSJ Transactions on System LSI Design Methodology**, Tokyo, Japan, v.8, n.0, p.2–11, 2015.

ZHOU, J.; KIM, K. H.; LU, W. Crossbar RRAM Arrays: Selector Device Requirements During Read Operation. **IEEE Transactions on Electron Devices**, New York, NY, USA, v.61, n.5, p.1369–1376, May 2014.

ZHOU, M. et al. Real-Time Scheduling for Phase Change Main Memory Systems. In: IEEE 10TH INTERNATIONAL CONFERENCE ON TRUST, SECURITY AND PRIVACY IN COMPUTING AND COMMUNICATIONS, 2011, Changsha, China. **Proceedings...** IEEE, 2011. p.991–998.

ZHOU, P.; ZHAO, B.; YANG, J.; ZHANG, Y. Energy reduction for STT-RAM using early write termination. In: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 2009, San Jose, CA, USA. **Proceedings...** ACM, 2009. p.264–268.

ZOU, Q. et al. Heterogeneous architecture design with emerging 3D and non-volatile memory technologies. In: THE 20TH ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, 2015, Chiba, Japan. **Proceedings...** IEEE, 2015. p.785–790.