technische universität
dortmund

Bachelor Thesis

Multi Energy and Timing Mode Simulation for Non-Volatile Main Memory

Janina Rau

11.12.2023

Supervisors:
Prof. Dr. Jian-Jia Chen
Christian Hakert, M.Sc.

TU Dortmund University
Department of Computer Science
Design Automation for Embedded Systems Group
`https://daes.cs.tu-dortmund.de`

Abstract

Emerging non-volatile memory (NVM) technologies show potential as a possible replacement for DRAM by having comparable characteristics and advantages such as low leakage power, high density and fast read speed. To tackle the major disadvantage of higher energy consumption, many optimization approaches, such as using different write modes, are proposed. As building the required hardware can be hard, optimization methods are often tested in simulators capable of simulating NVM features, one example of which is NVMain 2.0. When focusing on simulating different NVM write modes, most simulation environments lack the necessary features of defining multiple separate write modes, as well as using them only for specific areas in the memory.

This thesis therefore extends a simulation environment (i.e., NVMain 2.0 as the NVM simulator with gem5 as a simulator for an ARM CPU) to be able to define different areas in the memory with assigned write modes regarding the energy and timing parameters. The new feature is complemented by the functionality of analyzing the energy and time consumption of the defined memory areas, as well as a write mode interface that enables a change of the write mode during the runtime of the simulation.

Results of the evaluation show the new features are able to reduce the energy and time consumption of NVMs by defining fine-granular areas in the memory with different write modes. The interface enables the write modes to be changed and used effectively during the runtime of the simulation, which is an advantage especially for programs with dynamic access behavior.

Contents

1 Introduction

Non-volatile memory (NVM) technologies have emerged as candidates for future universal memory. The field of non-volatile memories is large, with many technologies already used in practice while other technologies are still in a prototype or research state [8]. A replacement of DRAM is considered possible because NVMs have comparable device characteristics [7].

Advantages of NVM technologies over conventional memory technologies include, for example, low leakage power, high density, and fast read speed. They also have disadvantages such as higher energy consumption and latency in write accesses compared to DRAM [27]. Moreover, NVM technologies like Phase Change Memory (PCM) only have a limited write endurance. It ranges from 10^8 to 10^9 write cycles, while DRAM usually endures more than 10^{15} writes per cell [7]. To effectively use NVM technologies, optimization is utilized to lessen the impact of the described disadvantages. Many approaches exist to extend the lifetime of NVMs through, e.g., wear-leveling or avoiding unnecessary writes to the memory [28]. Also important, especially for systems with limited available energy, is the reduction of the overall energy consumption. This can be achieved by, for example, avoiding unnecessary writes to the memory [28] or using different modes for write accesses. Write modes can be utilized to closely match the retention requirements of the stored data, thus reducing the energy consumption and latency of individual write accesses [16].

Because the designing and producing of NVM systems without previous optimization can be very expensive and time-consuming, simulations of the memory system are used to test promising methods. Different simulators exist for this purpose, with NVMain 2.0 being one example [6, 2, 15, 18, 20]. NVMain 2.0 is a cycle-accurate simulator specifically designed for NVM technologies [20]. Its flexible design allows for a great variety in simulation aspects. Regarding the simulation of energy and time consumption, NVMain 2.0 neglects some characteristics of NVMs worth considering. In the simulation configuration, the energy and timing values for read and write accesses are set to only one value. With the possibility of memory cells in NVM technologies having different requirements from each other for successful memory accesses and the option to utilize different write modes, one value applying to the whole memory excludes such simulations. Another possible characteristic of NVMs is the difference between the set and reset operations of a write access, in terms of energy and time consumption [26]. This option is also not represented in the NVMain 2.0 simulator, as it makes the assumption that set and reset operations

have the same characteristics. Therefore, NVMain 2.0 lacks support for the simulation of these aspects of NVM technologies.
In this thesis, the simulation environment is extended to support the definition of different areas in the memory with corresponding energy and timing values. Consequently, features are added to support the output and analysis of these memory areas. Another goal is to be able to change the area definitions at runtime. The code for this implementation is publicly available[1].
After a discussion of related work in literature, an introduction to the basic aspects of NVM technologies is given. This introduction includes general aspects as well as the operating principles of three already extensively researched NVM technologies: Phase Change Memory, Ferroelectric RAM, and Spin-Transfer Torque Magnetoresistive RAM. The reasons for their different energy and timing characteristics are explained, and then compared to the characteristics of each other and DRAM. Also, the simulation environment extended in this thesis is presented and explained. It consists of a combination of NVMain 2.0 [20] and gem5 [5]. The latter is used to simulate modern computer architecture at the cycle level, enabling NVMain 2.0 to perform a more accurate simulation. The subsequent contributions are presented in Chapter 4 and can be divided into three different aspects:

- Basic adjustments are made in the simulator. They are needed to define different energy and timing areas. The changes are kept as flexible as possible to allow for further adjustments.
- Changes are implemented to support the output and analysis of the new simulation model.
- A configurable CPU interface is written which is able to configure the write mode during runtime.

Chapter 5 shows an experimental evaluation of all applied changes. The results are discussed and compared to the related work. To conclude this thesis future additions to the simulator are proposed.

[1]`www.github.com/janinarau/bachelorthesis`

2 Related Work

As stated before, the optimization of NVM systems regarding e.g. their energy consumption is an important research topic. Many approaches are proposed, to lower the energy consumption and extend the lifetime of NVM technologies. Furthermore, some approaches rather focus on increasing the memory density by storing more than one bit in a memory cell, which requires different write modes. Simultaneously, different systems, simulators and emulators, are developed or modified to simulate these emerging NVM technologies. Because existing DRAM simulators cannot easily be used for NVM technologies, these new simulators are needed to accurately test new NVM systems or optimization methods. Since this work discusses the simulation of energy and timing modes for NVMs, this chapter presents different NVM write modes, as well as a selection of NVM simulators or emulators.

2.1 NVM Write Modes

To tackle the problem of limited write endurance of NVMs, as well as the high write energy consumption, different write modes are proposed. An example for Phase Change Memory based memories is the dual-write mode, where the memory realizes a fast and a slow write mode [16, 21]. The fast mode writes a volatile memory value, by using shorter write operations with less energy. This leaves the cell in a state not stable enough for it to be considered non-volatile. The slow write mode is equivalent to the normal write mode, using the full latency and energy, therefore writing a non-volatile value. If used correctly, this method can lead to a significant improvement in write energy consumption.
Li et al. propose a compiler directed dual-write scheme (CDDW) that selects the best mode for each write operation [16]. The fast write mode is used in all cases where the corresponding value does not have a long lifetime. This results in a shorter latency and lower energy consumption for each write operation done with the fast mode. In cases where a written value has a long lifetime and would be in need of a refresh, the slow write mode is used instead to create a non-volatile value. Li et al. use this principle by analyzing the worst case lifetime of the memory write instructions of a program. Based on whether the worst case lifetime is less than the retention of a fast write, the best write mode is selected. The conducted experiments show an improvement on both performance and energy efficiency compared to a full fast write mode or a full slow write mode [16].

Another example for a Phase Change Memory based dual-write mode is proposed by Qiu et al.. Building on the previously described CDDW, they suggest a loop tiling approach to be able to effectively use different write modes even in large-scale loops [21]. Programs using loops with intensive data array operations have read and write dependencies. Therefore, without altering the approach and using only the CDDW, the slow write mode would almost always be used. To benefit from the CDDW, Qiu et al. utilize a loop tiling technique to reduce the lifetimes of the memory write instructions. Experimental validation shows this approach leading to a lower dynamic energy consumption compared to only slow or fast writes, but also compared to the unaltered CDDW approach [21].

Besides a dual-write mode, removing redundant bit-writes is another approach to reduce the write energy consumption. Zhou et al. introduce the term of redundant writes, meaning writes that do not change the value of a memory cell. The number of these are significant and can average up to 85%, depending on the cell [28]. They describe this method primarily to improve the endurance of PCM. But since the method aims to remove redundant writes, it can also reduce the write energy consumption. This is achieved because unnecessary writes are not executed and therefore do not consume energy. This result is also demonstrated in their experimental evaluation. To implement the removing of redundant bit-writes, Zhou et al. simply precede each write with a read. After this, only the changed bits are written. Because read accesses consume less energy and are much faster than writes, the loss of performance of this implementation is lower than the overall performance improvement [28].

Cho and Lee propose an improvement to just removing the redundant bit-writes with the concept of Flip-N-Write [10]. Similarly, it suppresses unnecessary writes by comparing the old data with the new data. But going further, Flip-N-Write re-encodes the new data to further minimize the needed writes. To do this, a "flip bit" is added to every memory word that indicates if the associated data of the memory word is flipped or not. By determining the hamming distance between the old and new data, the number of bits that needs to be updated is computed. If this number is higher than half the length of the memory word width, the data and the flip bit value is flipped. If the number is not higher, the data is not flipped. An important side-note is that only the bits of the data are written, that are different to the already stored ones. Because Flip-N-Write flips the data, if more than half the bits of the memory word are different, no more bits than half of the memory word are written. Cho and Lee experimentally show that the proposed Flip-N-Write technique reduces the write time in Phase Change Memory by half, more than doubles the write endurance and reduces write energy [10].

Another existing write mode is the iterative write, or program-and-verify technique. Unlike the previously presented write modes, the goal of this mode is not to save energy or extend the cell endurance. By writing more than one bit per cell, this method aims to increase the storage density while reducing the cost-per-bit. Bedeschi et al. use this method to realize a 2 bit per cell storage [3]. The challenge in storing more than one bit per cell is that the cell resistance needs to be programmed with higher accuracy than a normal cell. By maximizing the read margin between the different states, errors are prevented. This accuracy is achieved by the already mentioned program-and-verify technique. The cell is first programmed to its minimum resistance state, to establish a baseline. After that, the cell receives a single reset pulse, initializing the cell to a reset state. Then the Stair-Case-Up algorithm is used to accurately program the cell. It consists of a sequence of program pulses, each followed by a verify step. Bedeschi et al. show in their evaluation that the proposed technique can realize a 2 bit per cell storage. It typically requires less than 8 programming pulses and thus leads to a promising throughput [3].

2.2 NVM Simulator

The two major Non-Volatile Main Memory evaluation environments are software simulators and hardware emulators. Whereas software simulators enable cycle-accurate simulation with flexible parameters and configuration settings, hardware emulators can execute applications at the speed of the base hardware [6, 2, 15, 18]. Because of the large simulation time of simulators, they are not able to evaluate system-wide performance. On the other hand, the model and implementation of emulators are often not completely validated. Thus, both simulators and emulators are needed as evaluation environments. Below, a selection of simulators and emulators used for Non-Volatile Main Memory are presented.

Bock et al. created HMMSim, a trace-driven simulator for hardware-software co-design of hybrid main memory [6]. It is capable of simulating DRAM and NVM in several hybrid memory organizations, including single memory systems, DRAM cache, and software-managed hybrid memory. While other NVM simulators like NVMain [20] require another simulator to simulate software-managed systems, HMMSim already includes this capability. Bock et al. describe the memory hierarchy as composed of three main components: CPUs, caches and memory. The memory is simulated through a set of configurable objects that can model either DRAM or NVM, accessible through a DDR interface. The model includes, among other things, multiple banks, row buffers and the support for different address mappings and row buffer policies. The simulator also provides a hybrid memory controller that redirects requests to either DRAM or NVM based on the physical address. Bock et al. also implement an API to enable the extension of the functionality of the simulator. To demonstrate the flexibility and usefulness of the HMMSim simulator, two case studies are described that utilize the described API. Additionally, the performance of the simulator is evaluated by analyzing the execution time and memory usage. The results show that HMMSim is fast and scales well [6].

Asifuzzaman et al. utilize a combination of already existing simulators to create a reliable main memory simulation for STT-MRAM [2]. For system simulation, they use the ZSim system simulator [23], which they first upgrade to support the simulation of the wanted processor. As the main memory simulator, they use DRAMSim2 [22]. DRAMSim2 is a cycle accurate model of a DRAM main memory. All major components in a modern memory system are modeled as their own respective objects within the source code. Through a simple interface, DRAMSim2 can be integrated with various CPU simulators. However, the authors conclude a simple integration of ZSim and DRAMSim2 may lead to an underestimation of the main memory access latency, because the delay contributed by all the circuitry between the last level cache and main memory device would not be considered. Therefore, they introduce an extra latency between ZSim and DRAMSim2. To simulate STT-MRAM with a DRAM Simulator, Asifuzzaman et al. divide the memory parameters into a part that does not need adjustment, and a part that does. As STT-MRAM memory is DDRx compatible, many timing parameters are the same. The only fundamental difference in STT-MRAM and DRAM main memory is their storage cell technology. Therefore, the timing parameters associated with the STT-MRAM row operations deviate from DRAM and need to be adjusted to the simulation. Analyzing the system performance impact with STT-MRAM main memory in comparison to DRAM main memory, the results show, that STT-MRAM would provide performance comparable to DRAM systems [2]. It should also be mentioned that this work only concentrates on the timing aspects of STT-MRAM. Asifuzzaman et al. are not able to simulate the energy consumption of STT-MRAM due to the lack of publicly available up-to-date resources at the publication time.

Lee and Yoo develop a reliable NVM emulation hardware platform that can be used particularly for research purposes [15]. They choose Xilinx Zynq as the base system for emulation. It consists of an ARM-based processing system and Xilinx FPGA-based programmable logic. All system stacks of their developed NVM emulation board are considered to support the emulation of NVM. The two major emulations they focus on are the latency and non-volatility. To emulate variable NVM latency on a real hardware platform, Lee and Yoo parameterize the memory timing parameters of the underlying DRAM banks. By doing this, the user application can access and change the latency parameters at run-time. The only limitation for setting the latency values are the predefined minimum latency requirements of DRAM. The emulation of non-volatility is done by modifying the board operations, so the used DRAM is refreshed under any circumstances. Typically, the board reset wipes out the FPGA image including the DRAM controller. Therefore, the DRAM is not refreshed and loses its data. Lee and Yoo modify the reset operation, which leads to no data loses, thus emulating non-volatility. They also conduct a case study with several benchmarks, which shows the memory access behavior of software programs by varying memory latency [15].

Omori and Kimura propose an NVMM emulator for embedded systems built upon an ARM multicore-based Zynq SoC board [18]. The emulator employs three behavior models: coarse-grain, fine-grain and DCPMM-based. The DCPMM model corresponds to the existing memory model Intel Optane DC persistent memory. While the coarse-grain and fine-grain model enable DRAM-based NVMM performance evaluations, the DCPMM model enables DCPMM-based NVMM performance evaluations on embedded systems. Omori and Kimura define the behavior models, representing possible NVMM architectures and behaviors [18]. The coarse-grain model extends the traditional DRAM-based main memory by injecting additional delays at the memory bus between the last level cache and the memory controller. This model represents NVMMs with no caches like row-buffer in memory modules, thus all memory requests are delayed. The fine-grain model does the same but injects the delays directly at the memory controller. Therefore, the delays only impact memory requests accessing memory cells. This model represents NVMM having the organization of banks, rows, and columns, similar to the traditional DRAM architecture. Thus, memory requests are delayed only when they miss row-buffers. The DCPMM model represents NVMM architectures that are similar to DCPMM. The detailed characteristics of this model are listed in [18]. Omori and Kimura validate the proposed emulator with another NVMM emulator, a cycle-accurate simulator and a real DCPMM model. The results show the effectiveness of the emulator implementation [18].

3 Background

In this chapter, background information for the simulation extension is presented. At first, a short presentation of DRAM is given. After that, basic aspects of NVM technologies are explained in Section 3.1. Their operating principles are introduced, and their energy and timing properties are compared to each other. Section 3.2 then gives an overview of the simulation environment, including the operating principle and configuration.

The main memory technology conventionally used is DRAM, which stands for Dynamic Random Access Memory. It uses a single transistor-capacitor pair to store each bit, as seen in Figure 3.1. Because of this, the representation of 1s and 0s is achieved by the electrical charge of the capacitor. These DRAM cells are called dynamic, because the capacitors storing electrons are not perfect devices. Each capacitor in the DRAM must be periodically refreshed, i.e. read and rewritten. Otherwise, the information in the capacitors cannot be retained, due to their eventual leakage [12].

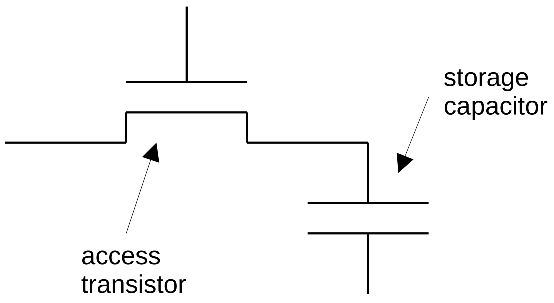


Figure 3.1: Design of a DRAM Cell [13]

A DRAM memory system consists of several components, organized in a hierarchical structure. The smallest addressable unit of memory is a *column*. In most systems, a column of data is the same size as the width of the data bus. A group of storage cells activated in parallel is called a *row*. A memory (sub-) array then consists of a number of memory cells that are organized in rows and columns. Multiple arrays within a DRAM device are grouped together in a so-called *bank*. The next-higher structure is a *rank*, which denotes a set of DRAM devices that operate in lockstep. Lastly, a *channel* refers to the group of all DRAM devices which share the same physical link [11]. Figure 3.2 shows an example memory organization of one channel, with ranks, banks and memory arrays consisting of rows and columns.

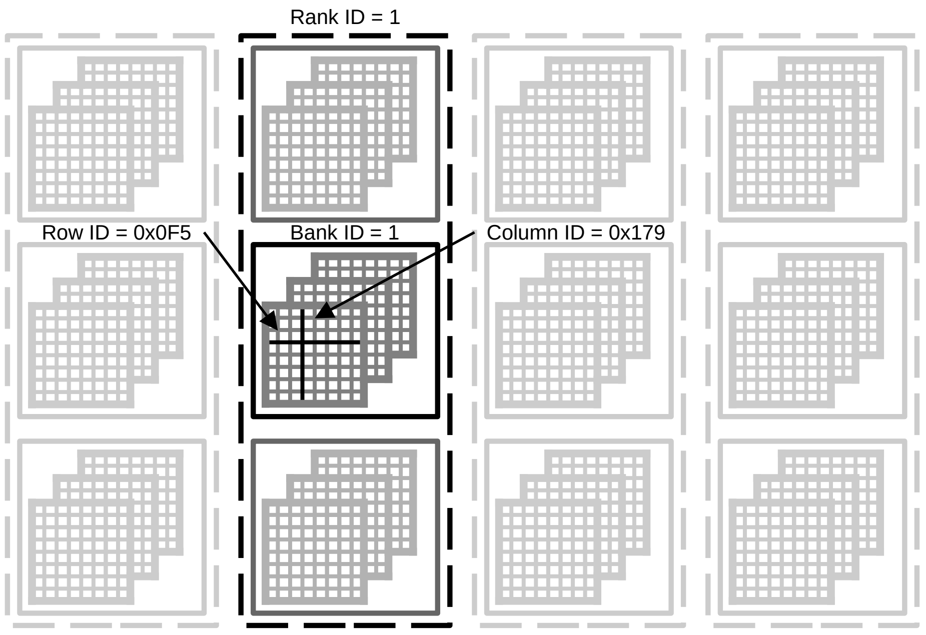


Figure 3.2: Memory System Organization of DRAM [11]

The advantages of using DRAM as the main memory in a system are its low latency in memory accesses, low energy consumption per read and write accesses and an unlimited endurance [27]. On the other hand, one of the greatest disadvantages of DRAM is the need to periodically refresh all memory cells. This not only makes the technology volatile, but also wastes energy and weakens the system performance by interfering with memory accesses. Even though the refresh rates can be optimized from rates such as 64 ms to, for example, 256 ms, they are still relatively high [17]. Therefore, DRAM is overall a suitable memory technology for many cases. But there are also application cases where the disadvantages of DRAM outweigh the advantages.

3.1 Non-Volatile Main Memory

As mentioned in Chapter 1, there are many emerging technologies for non-volatile main memories. While DRAM represents data through electric charge [12], most non-volatile main memories represent the logical 1 and logical 0 through changing physical properties [7]. Therefore, the operating principles of NVMs can be very different from each other. Generally, they have advantages such as low leakage power, high density, and fast read speed. At the same time, they can have disadvantages such as asymmetric read and write speed and energy cost [27].

3.1.1 Technologies

There is a high number of NVM technologies in various stages of production, prototype, and research [8]. Hence, only a short selection of already established technologies is presented in this thesis. Their operating principles and important characteristics are explained in the following section.

Phase Change Memory (PCM)

Phase Change Memory is one NVM technology already extensively researched and put into practice. It relies on the phase change abilities of chalcogenide to store data, by using its two stable states, crystalline and amorphous [27]. The important difference between the states is the resistance of the material, which is used in PCM to store the data. While the crystalline phase has a low resistance and represents the logical 1, the amorphous phase has a high resistance and represents the logical 0. The distinction between the two states is then made through the measured resistance. A PCM cell is usually constructed with a layer of chalcogenide between two electrodes and a heater. As depicted in Figure 3.3, the material in proximity to the heater is the actual programmable region. It changes its state depending on the heating process [26].

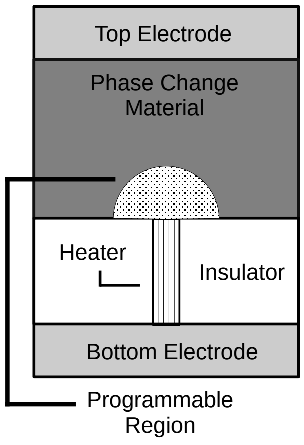
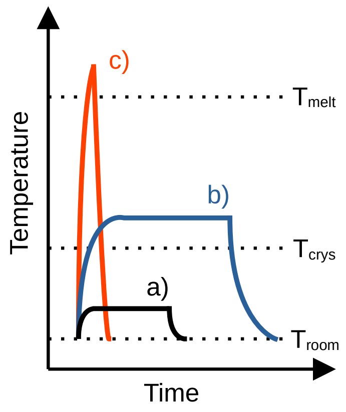


Figure 3.3: Structure of a PCM Cell [26]

As mentioned in Section 3.1, NVM technologies can have the disadvantage of asymmetric read and write speed and energy cost. This applies to PCM because of how memory accesses work. In general, memory accesses are done by applying electrical current pulses. The intensity and duration of the pulse then determine the type of the memory access. While a read access only needs a small electrical current to measure the resistance of the cell (see fig. 3.4 a), the set or reset from a write access require stronger pulses. To set the PCM cell to a crystalline phase, a medium electrical current pulse is applied. The pulse anneals the programming region at a temperature between the crystallization temperature and the melting temperature (fig. 3.4 b). This is done for a duration long enough to crystallize the material. To reset the PCM cell from a crystalline to an amorphous state, the programming region is first melted and then quenched rapidly. This is achieved by applying a large electrical current pulse for a short period of time. At the same time, the melting temperature of the material has to be reached (fig. 3.4 c) [26].



(a) read pulse, b) set pulse, c) reset pulse)

Figure 3.4: Schematic Pulses of PCM Memory Accesses [26]

Because PCM uses a physical medium to store information and not electric charges, achieving an unlimited endurance is not possible. The main source of failure of PCM cells is the repeated heat stress applied to the material, resulting in cells not being able to change their state anymore [27]. Different methods are proposed to increase the endurance and decrease the energy consumption of PCM. Some of them are presented in Section 2.1. While the retention of data in PCM cells can be up to 10 years under ideal conditions [26], the retention of a memory cell is always depending on the used write energy and time. If less than the required energy is used in a shorter pulse, the written value will be lost sooner [16]. As already presented in Section 2.1, this characteristic can also be used beneficially.

Ferroelectric RAM (FeRAM)

Ferroelectric RAM is a type of non-volatile main memory that is already in the stage of mass production. The structure of a FeRAM cell closely resembles that of a DRAM cell [7]. It also consists of a transistor-capacitor pair, but instead of using a dielectric material like DRAM, it uses a ferroelectric capacitor. This capacitor uses the ferroelectric effect of its material, to return to an electrical polarization in the absence of an applied electric field. As a result, it can store data without a high leakage as in DRAM. The structure of a FeRAM memory cell can be seen in Figure 3.5, showing the most significant difference to a DRAM cell being the capacitor. Because of this similarity, the memory architecture of FeRAM is also very similar to DRAM [19].

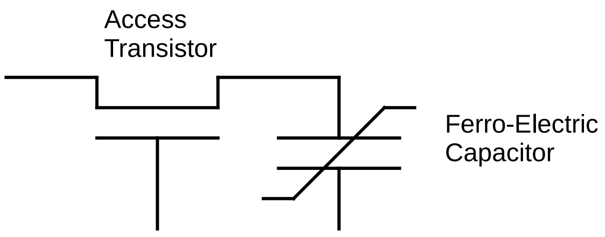


Figure 3.5: Structure of a FeRAM Cell [19]

The process of a memory access in FeRAM is dependent on the properties of the ferroelectric capacitor. Its material exhibits hysteresis, meaning the behavior of the capacitor depends on previous interactions. In Figure 3.6, a hysteresis loop for a ferroelectric capacitor is shown. It displays the total charge of the capacitor as a function of the applied voltage [24], with Q_s being the saturation charge. Based on the direction of an externally applied electric field, the capacitor will move into one of the two stable positions, 0 or 1. Once the external field is removed, the capacitor remains in a stable position [19], also shown in Figure 3.6. The two states can then be switched by applying a voltage pulse across the capacitor. To switch from 0 to 1, a negative pulse is applied to the capacitor and to switch from 1 to 0, a positive pulse is applied [24].

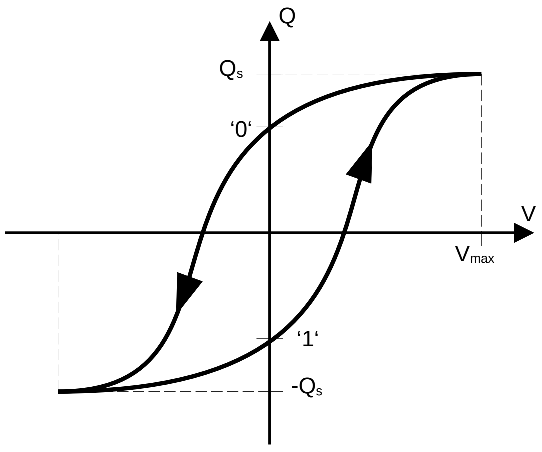


Figure 3.6: Hysteresis Loop Characteristic of a Ferroelectric Capacitor [24]

To read the stored data of a ferroelectric memory cell, an electric field is applied to the capacitor. If the applied field is in the direction to switch the internal dipoles, more charge will be moved than if the dipoles are not reversed. This charge is measured by the FeRAM chip and used to produce either a 0 or a 1. After the read takes place, the chip automatically restores the correct data to the cell [19].

The disadvantage of using a ferroelectric capacitor is the degradation of its switching ability over time. While the endurance of FeRAM is generally sufficient, it is also not unlimited because of this [19].

Spin-Transfer Torque Magnetoresistive RAM (STT-MRAM)

Magnetoresistive RAM uses magnetic properties to store information, by controlling and sensing the magnetic orientation of a special material using electrical signals [7]. An MRAM memory cell, called Magnetic Tunnel Junction (MTJ), consists of two ferromagnetic layers separated by an oxide (tunnel) barrier layer. The *reference layer* has a fixed direction of magnetization, while the *free layer* has a variable direction. In the so-called parallel state, the two layers have the same direction of magnetization, which results in a low resistance of the MTJ. The high resistance state, the antiparallel state, occurs when the two layers have opposite directions. Using the MTJ as a memory cell, the low resistance state represents a logical 0, while the high resistance state represents the logical 1. Both states are shown in Figure 3.7.

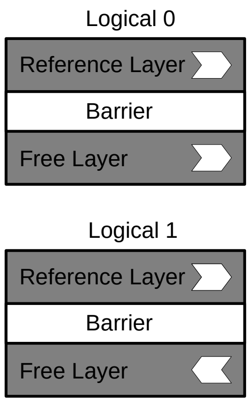


Figure 3.7: MRAM Based Memory Operations [14]

A typical MRAM cell consists of an MTJ and a transistor. The absolute voltage difference between two states is not high enough and requires a transistor to function. Moreover, the transistor also provides the current required for the write operation [4]. An example of an STT-MRAM memory cell, consisting of an MTJ and a transistor, is displayed in Figure 3.8.

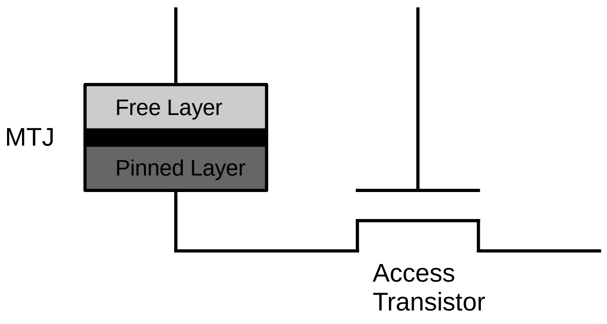


Figure 3.8: Structure of a STT-MRAM Cell [27]

The Spin Transfer Torque variant of MRAM (STT-(M)RAM) uses an applied voltage to change the magnetization direction of the free layer. When a high positive voltage difference is applied, a logical 0 is written. Consequently, a logical 1 is written with a high negative voltage difference. However, the current has to be long and strong enough to ensure a status reversal. This operating principle denotes the difference between STT-MRAM and the original MRAM cell. Initially, the orientation was set using an external magnetic field, while STT-MRAM uses a current of polarized electrons [7]. While the write operation uses a high voltage, the read operation uses a very small voltage, which causes a current to flow through the MTJ. This current is compared to a reference current and depending on the value, relative to the resistance of the MTJ, the value 0 or 1 is read from the cell.

While MRAM in general has the advantage of high density, fast read speed, low leakage and high endurance, there are of course disadvantages. Here, MRAM suffers mainly from two sources of unreliability. Through thermal instability, MTJ cells have the possibility of data loss. Also, the used high write current stresses the memory cells and degrades the junction. This limits the cell integrity over time. Moreover, this degradation can lead to a so-called *read disturb*. The read operation induces a change in magnetization, because of a reduction of the write current [7].

3.1.2 Energy and Timing Properties

As described, NVM technologies have different operating principles and therefore different characteristics. Because of these, they not only differ from DRAM, but they also differ from each other. To give an overview of the possible differences, they are briefly discussed here. In Table 3.1, different device properties of DRAM, PCM, FeRAM and STT-MRAM are listed. These values give an overall idea of the properties and do not have to reflect the exact properties of real devices.

	DRAM	PCM	FeRAM	STT-MRAM
Read Latency[1] (ns)	~10	20-60	20-80	2-35
Read Energy[2] (pJ/bit)	10	12.4	12.4	58.5
Write Latency[1] (ns)	~10	20-150	50-75	3-50
Write Energy[2] (pJ/bit)	10	210.3	210	67.7
Retention Time[3]	64 ms	–	–	–
Endurance[1] (Write Cycle)	$> 10^{15}$	10^{8}-10^{9}	10^{14}-10^{15}	10^{12}-10^{15}

Table 3.1: Comparison of Energy and Timing Properties ([7][1],[25][2], [17][3])

It is noticeable that the latencies as well as the energy consumption in the non-volatile main memories is almost always higher than in DRAM. The reason for this are the previously explained operating principles. The heating of a material to a certain point, for example, is more energy and time consuming than the charge of a capacitor. Despite the values being higher, they are still in reasonable range to be able to use non-volatile main memories as a replacement for DRAM. On the other hand, the retention time of non-volatile main memories is much higher than that of volatile DRAM. Exact values cannot be named, because they are dependent on the write process, but times up to 10 years are reported to have been achieved [26]. As previously mentioned, all non-volatile main memory cells lose, sooner or later, some of their information retaining properties, leading to a worse endurance than DRAM. Despite some of the shown properties being better than others, they are not the only basis considered for choosing a memory type.

3.2 Simulation Environment

As presented in Chapter 2, a variety of simulators and emulators for NVMs exist. In this thesis, NVMain 2.0 [20] in combination with gem5 [5] is used. NVMain 2.0 is a flexible memory simulator for not only DRAM, but also emerging memory technologies such as non-volatile memories and hybrid systems. gem5 models modern computer hardware at the cycle level and is capable of simulating different CPU architectures like x86 and ARM. The source-code from gem5 and NVMain2.0 can be obtained from their official websites[1][2].

3.2.1 NVMain 2.0

Because NVMain is a simulator that is not originally designed for only DRAM, but is explicitly developed for non-volatile main memories, it also supports features that are unique for them. In NVMain, endurance and fault recovery are supported, mainly through data encoding. Furthermore, the simulation of multi-level-cells, meaning the storing of more than one bit per memory cell, as well as the simulation of hybrid systems, containing both NVMs and DRAM, is realized. Because of the flexible design of NVMain 2.0 the simulations can be highly configured, regarding almost every aspect of the memory [20].

The high-level design of NVMain with one memory controller and one channel is shown in Figure 3.9. Each box represents a memory base object. Sub-arrays are defined as the most basic block, therefore supporting the simulation of sub-array-level parallelism [20].

[1]gem5: `www.gem5.org`
[2]NVMain2.0: `www.github.com/SEAL-UCSB/NVmain`

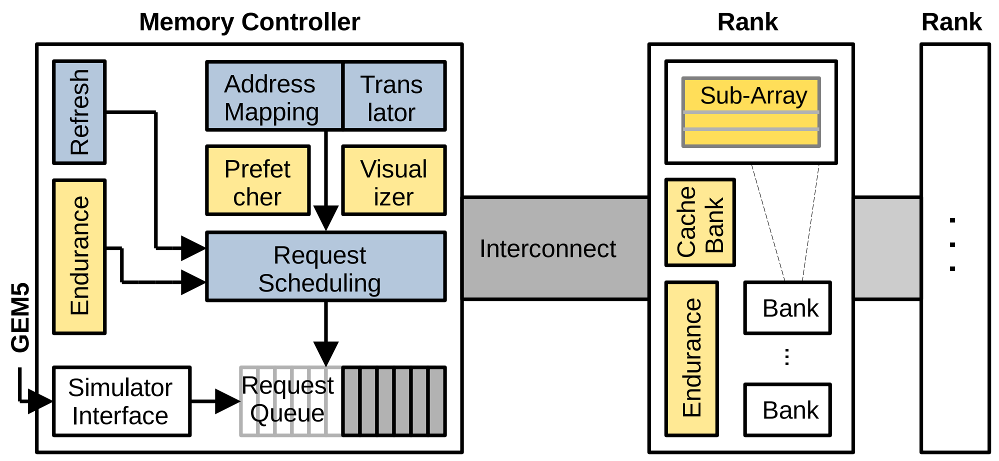


Figure 3.9: Overview of NVMain Architecture [20]

Operating Principle

To get into further details, the operating principles of NVMain needed in this thesis will be now explained in more detail. As previously explained, NVMain realizes all components of a memory system as memory base objects and therefore in separate classes. One of the most important classes is the `NVMainRequest` class. An NVMainRequest object describes a single simulated memory access of any type. It includes the address of the access, the operation type, the status of the request and the new data that is written or read, as well as the old data that is possibly overwritten. Additionally, detailed scheduling information is stored. For example, the cycle in which the request arrived at the memory controller or how often it was canceled. The address is realized in a separate class named `NVMAdress`. It not only stores the address itself, but also the corresponding physical location. It consists of channel, rank, bank, sub-array, row and column. Through an `AddressTranslator` it is possible to convert a physical location to an address and vice versa. Memory requests are created in the `nvmain_mem` class, which is part of the interface to gem5. A packet from gem5 that is received by the `MemoryPort` is used to create a request object with all information from the packet.

At the start of a simulation, the `nvmain` class initializes all important classes depending on the configuration file. This configuration file is used to set all parameters needed for the memory size, its timing and energy parameters as well as for example scheduling and encoding methods. The class that is responsible for all scheduling decisions is the `MemoryController`. It receives each memory request and, based on the scheduling method, places them in a transaction queue to be executed. In almost every class of the memory system, the `IssueCommand` function is implemented. This function receives each request and processes it further, depending on the specific class. For example, the `SubArray` class simulates the column write and read functions and calculates all needed parameter on this level. For every request, the `nvmain` class creates a so-called `TraceLine`, used to store all output-related information. The `TraceLine` then gets handed over to the specified `TraceWriter`. Depending on the implementation of the specific trace writer, details of the `TraceLine` are selected to be used for the output. The output of the `TraceWriter` is then written into a trace file specified in the configuration file, or into the console. A graphical representation of the path a memory request takes through the simulator is shown in Figure 3.10.

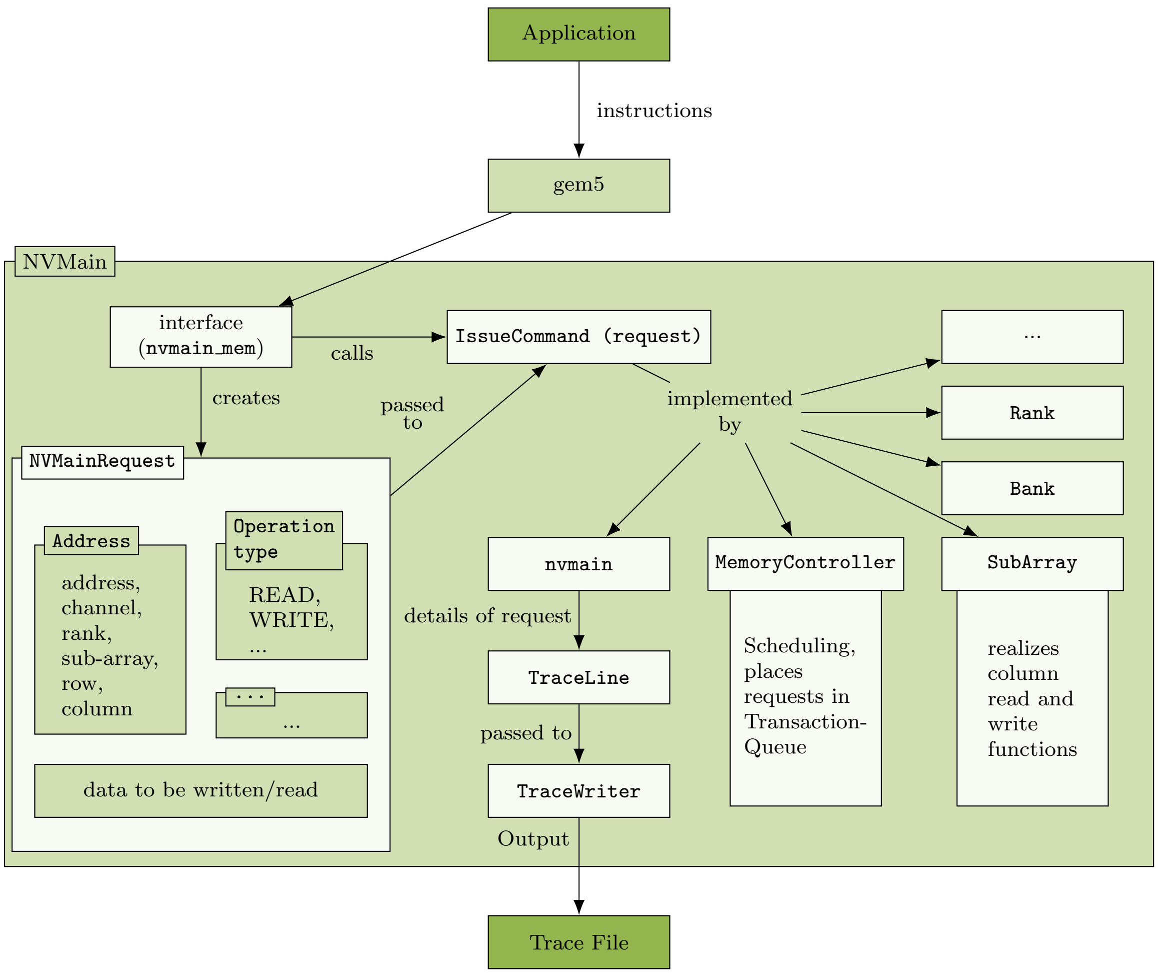


Figure 3.10: Path of a Request

3.2.2 gem5

The gem5 simulator is a cycle-level computer system simulation environment, capable of simulating CPUs with different complex architectures [5]. Containing not only the event-driven simulation engine, but also numerous of models for system models, the simulator is highly customizable. Two different modes can be used in the simulator: a so-called system call emulation (SE) or a full system (FS) mode. The system call emulation mode emulates the operating system. Thus, it ignores the timing of many system-level effects, including system calls, TLB misses and device accesses. The full system mode can boot a full Linux-based operating system. To use it, a concrete CPU model and machine of the provided options have to be configured. The CPU models include, for example, a ”simple” CPU model, which has less fidelity compared to real devices, but has a faster simulation time. Furthermore, gem5 contains high-fidelity models, that have slower simulation time, but provide more realistic results. Examples for this are a detailed in-order CPU model (the ”minor” CPU) and an out-of-order CPU model (the ”O3” CPU) [5].

gem5 supports a number of industry standard architectures (ISA) like ARM, MIPS, RISC-V and x86. Because the CPU model are designed to be ISA-agnostic, all of these ISAs can be used with any of the included CPU models [5]. A RISC instruction set architecture widely used in many different platforms is the ARM architecture [1]. It employs a weakly ordered model of memory, meaning the order of memory accesses is not necessarily required to be the same as the program order for load and store operations. To improve data throughput, the processor can reorder memory operations with respect to each other. The operations can be ordered by address dependencies and half barriers, but have to consider data dependencies and explicit memory barrier instructions. By doing this, the required bandwidth between processor and external memory can be reduced. The number of transfers can also be reduced, by merging together several write operations into one larger operation [1].

Per cycle, the processor can issue and execute multiple instructions. These supported instructions include the load and store instructions with different offsets and privileges. To allow management software to control the view of memory that is presented to the software, virtual addresses are used for the software. Every virtual address has to be first translated to a physical address, used by the memory system, so that a memory access can take place. ARM uses a memory management unit (MMU) for translating the virtual addresses to physical addresses. The MMU consists of the table walk unit, reading the translation tables from memory, and the translation lookaside buffers (TLBs), caching recently used translations. The memory model of ARM also includes the possibility to use caches, to increase the average speed of memory accesses. Multiple levels of cache can be used in a hierarchical system, exploiting the trade-off between size and latency. The hierarchy can consist of one or more levels of separate instruction and data caches, having one or more unified caches located closest to the main memory. When using caches, the memory coherency has to be taken into account, defined by using conceptual points such as Point of Unification or Point of Coherency. Because the caches use physical addresses, every access to a cache has to be translated first by the MMU [1].

Different file formats and networking protocols specify different endianness, meaning the storing of the most significant byte of a memory object at either the least significant or the most significant address. Because of this, ARM supports both of these modes, big endian and little endian, with little endian being the default mode [1].

3.2.3 Configuration

To use NVMain 2.0 in combination with gem5, the sourcetree of NVMain 2.0 has to be included in the building process of gem5. To achieve compatibility with gem5, NVMain 2.0 already includes a patch for this. gem5 is configured to use the full system mode simulation. Additionally, the O3 CPU model is used in combination with the ARMv8 machine architecture. As already mentioned, the main source of configuration for NVMain 2.0 is the configuration file. It defines the memory layout, timing and energy parameters, scheduling and encoding methods, as well as all other important parameters. NVMain 2.0 comes with a number of predefined configuration files. In this thesis, the `printtrace` configuration is used. It is a minimal configuration that sets the minimal required properties and enables the generation of the previously described trace files. Only the memory layout is altered so that more of the individual memory components in the simulation are used. This is beneficial to test the following contributions. Otherwise, for example, only one row per bank would be used in the simulation, due to the higher bank number defined in the default configuration.

4 Multi Energy and Timing Mode Simulation

Extending the simulator to support the features introduced in Chapter 1, the resulting approaches are described in detail in this chapter. They are split in three different parts, each one focusing on one specific feature. First, the simulator has to be able to define and simulate multiple areas in the memory, each having different energy and timing characteristics. Then, to be able to analyze the results of these changes, new trace writers have to be implemented. Furthermore, to allow the selection of different write modes during run-time, a configurable interface is created.

4.1 Energy and Timing Areas

Defining memory areas with individual energy and timing properties requires multiple steps to achieve. To show the development of this task, the general idea of the definition model is first explained. Next, the resulting additions to the simulator are shown, and lastly the testing is displayed, to ensure the accuracy of the changes.

4.1.1 Area Definition Model

The most important point in designing these changes in the simulator is to keep the changes as modular and expandable as possible. Thereby, future changes and additions are enabled. All energy and timing related simulations should therefore be enclosed in their own controllers. They are kept separate to enable the simulator and the user to only simulate one of the two properties. The memory controller then acts as the main interface between the energy and timing controllers and the rest of the simulation.

Both controllers must be able to save the definitions of multiple areas with different characteristics, reflecting those of the NVMs in real-life. For this purpose a suitable data structure has to be selected. In the best case, a data structure is used that is easy to search and can be used for various amounts of data and data types. Because each memory request is unique, the energy consumption and processing time of a request have to be calculated. The functions for this purpose must take the operation type into consideration, as well as the data that is written or read. In addition, the simulator should also be able to read user defined configuration files. They are used to set the energy and timing properties inside the controller.

4.1.2 Energy and Timing Controller

In the following section, the operating principles of the `EnergyController` and the `TimingController` class are explained. They are responsible for all energy and timing related tasks. Their integration into the simulator is described, as well as the area header file that is used to define needed types and functions.

Areas.h

The area header file contains the declaration of the `enValue` and `tiValue` types, which are used to store the energy and timing values. This is done to make a future adjustment of the used data type easier. Currently, the double data type, as seen in Listing 4.1, is used for all energy and timing values. Depending on the size of the used values and the demands to the precision, the need for another data type may arise.

```
typedef double enValue;
typedef double tiValue;
```

Listing 4.1: Data Types for Energy and Timing Values

To store the different areas, it is reasonable to set an order in which they are stored. Because the physical location of the area in the memory consists of six different properties, a comparator function is needed to establish the order. For this, the `cmpLocation` comparator function is defined in the area header file.
Because the physical location consists of six different properties, they have to be combined in some way to be able to be used as a single parameter. This is achieved by writing all six properties, namely channel, rank, bank, sub-array, row and column, together in a string. Three examples of the physical location as a string are presented in Listing 4.2. As shown, the properties are simply written into a predefined order with a blank space between them, to make them distinguishable. All functions needing the details of the properties can later extract the information from the string.
The `cmpLocation` function therefore first extracts the properties of the physical locations. Leaning on the memory layout described in Chapter 3, it then compares the individual properties with each other and returns the greater of the two. The three examples of Listing 4.2, ordered by the `cmpLocation` function, are therefore displayed in the correct order.

```
”0 0 0 0 0 1008” = channel: 0, rank: 0, bank: 0, sub−array: 0, row: 0, column: 1008
”0 0 0 0 1 12”   = channel: 0, rank: 0, bank: 0, sub−array: 0, row: 1, column: 12
”0 0 1 0 3 784”  = channel: 0, rank: 0, bank: 1, sub−array: 0, row: 3, column: 784
```

Listing 4.2: Examples of Physical Location

EnergyController

The `EnergyController` class is added to the simulator for all calculations required for the energy consumption of the simulated memory accesses. Its operating principle and all important aspects are explained in this section. As mentioned, the energy consumption of memory accesses of NVMs varies between read and write accesses, as well as the set and reset operations of cells. Also, the properties of the memory cells can differ from each other or are used with different write modes. Therefore, the `EnergyController` realizes different areas in the simulator with different properties of the energy consumption.

To meet the requirements described in Section 4.1.1 the `map`[1] data structure is chosen. The `map` data structure consists of pairs of a `key` and a `value`. The `key` has to be unique and can be of any other data structure. The `value` can also be any data structure, but does not have to be unique. In this case, the physical location inside the memory is chosen as the unique key. Depending on the translation method of the simulator, the address for a memory cell is not always the same. The address is therefore not suitable as the unique key. Thus, as already mentioned in the description of the area header file, the properties of the location are combined to function as the key. Table 4.1 shows an area is defined up until the point of a new definition. Any location that is not specifically stored in the `map` is matched to the area defined by the highest location that is strictly lower than the searched location. For example, the location ”0 0 0 0 1 256” would be matched to area b as its location is greater than the start location of area b but still lower than the start location of area c.

Channel 0	Column 16	Area a
Rank 0	$\downarrow$	$\downarrow$
Bank 0	$\downarrow$	$\downarrow$
Sub-Array 0	Column 64	Area b
Row 1	$\downarrow$	$\downarrow$
Channel 0	Column 0	Area c
Rank 0	$\downarrow$	$\downarrow$
Bank 1	$\downarrow$	$\downarrow$
Sub-Array 0	Column 128	Area d
Row 2	$\downarrow$	$\downarrow$

Table 4.1: Definition of Memory Areas

[1]reference of the map data structure: https://en.cppreference.com/w/cpp/container/map

The `value` of a key should then be the different energy characteristics. For the energy characteristics, three values have to be saved. These values correspond to the read, set and reset energy that is consumed by accessing a memory cell. To save the three values per location or key, the data structure `tuple`[2] is used. This data structure just groups a defined number of values of previously defined types together. The complete data structure to store the energy can be viewed in Listing 4.3. As described earlier, the location is stored as a string and the energy values are stored in tuples with the type `enValue`, which is declared in the area header file. The described comparator function `cmpLocation` is used to sort the areas in the map by their location from lowest to highest.

```
std::map <std::string , std::tuple<enValue,enValue,enValue>,
        cmpLocation> energyMap;
```

Listing 4.3: Datastructure for Storing Energy Areas

The implementation of the `EnergyController` consists of different components. The one component being not a direct part of the class itself is the configuration file. But because the file is processed by the `EnergyController` its structure is important for the implementation. The other components of the class are its functions realizing the proposed model. All important components are described in the following.

`Configuration file`

To define the different energy areas, a configuration file can be used. In this file, each line represents a different individual area. It starts at the defined location in the memory and ends at the start of another area. An example of a configuration file can be seen in Listing 4.4. The physical location consists of the channel, rank, bank, sub-array, row and column in this specific order. The last three values in a line are the energy consumed through a read, set and reset operation for one single memory cell.

```
0 0 1 0 2 1004 2.79 14.37 19.96
0 0 1 0 2 1008 2.59 14.17 19.85
0 0 1 0 2 1012 2.51 14.07 19.8
0 0 1 0 2 1016 2.43 14.0 19.64
0 0 1 0 2 1020 2.49 14.08 19.79
0 0 1 0 3 0 2.59 14.17 19.85
0 0 1 0 3 4 2.46 14.01 19.7
```

Listing 4.4: Part of a Configuration File for Different Energy Areas

[2] reference of the tuple data structure: https://en.cppreference.com/w/cpp/utility/tuple

`ReadConfig`

In the `ReadConfig` function of the energy controller, each line of the config is parsed to extract the different parameters of the location and the energy values. The physical location parameters are combined in a string, the three energy values in a tuple. The location is stored in the `energyMap` together with the energy tuple. There also exists a variation of this function, namely the `ReadJoinedConfig` function. This function also parses all relevant energy information into the `energyMap` but from a configuration file that contains both energy and timing values.

`GetEnergyValue`

This function is used to return the energy consumption of a specific request. To determine the energy consumption, first the physical location of the request is used to find the corresponding area in the map. This is done by finding the highest location that is still strictly less than the given location, because of the way the areas are defined. Next, a distinction is made by the operation type of the request.

If the request is a simulated read access, the returned energy consumption is calculated as follows. The value for the energy consumption of the read access of a single memory cell is multiplied with the number of bytes the read data block contains. Additionally it is multiplied by a factor of eight, because one byte corresponds to eight bits. In Listing 4.5, an example is displayed with a data block that contains eight bytes. Because eight bytes correspond to 64 Bit, the read energy of the defined area, that is set in the configuration file, is multiplied with 64.

```
Physical location of read access: 0 0 0 0 1 766
Defined area from configuration file: "0 0 0 0 1 764 2.43 14.0 19.64"
data to be read: d06c058000000000 (8 Byte = 64 Bit)
Calculation: 64 * 2.43 = 155.52
```

Listing 4.5: Example for Computing the Energy of a Read Access

If the request is a simulated write access, the calculation is more complex than a multiplication with a single number. The returned energy consumption is therefore calculated in the `DetermineWriteEnergy` function.

`DetermineWriteEnergy`

The function cycles through all bytes of the data block that should be written and counts how much set operations are used. This number is then subtracted from the overall bit number to calculate how many reset operations are used. Consequently, the number of set operations is multiplied with the set energy consumption per memory cell and the number of reset operations is multiplied with the reset energy consumption. Finally, the returned write energy consumption of this data block is the sum of both values. An example of this calculation is displayed in Listing 4.6. Here, the location of the write access (0 0 0 0 1 766) is used to find the corresponding defined area (0 0 0 0 1 764). The ones and zeros of the data that should be written are then counted and multiplied with the set and reset energy. The resulting sum is the energy consumption of this specific memory request.

```
Physical location of write access: 0 0 0 0 1 766
Defined area from configuration file: "0 0 0 0 1 764 2.43 14.0 19.64"
data to be written: d06c058000000000
binary data: 1101 0000 0110 1100 0000 0101 1000 0000 ... 0000
count ones and zeros: 10 ones, 54 zeros
Calculation: 10 * 14.0 + 54 * 19.64 = 1200.56
```

Listing 4.6: Example for Computing the Energy of a Write Access

TimingController

The `TimingController` class is very similar to the `EnergyController` class, therefore only the differences are explained here. This class is added to the simulator to do all calculations required for the access times of the simulated memory accesses. As well as the energy consumption, the access times also vary between read and write accesses. The set and reset times of cells, but also the properties of multiple memory cells can be different from each other. Similarly to the `EnergyController`, the timing controller realizes different areas in the simulator but with different properties of the access times.

The only difference to the data structure for the energy consumption is the typedef `tiValue`, used to store the access times. The rest of the data structure is identical to the one of the `EnergyController`. The complete data structure of the `TimingController` can be seen in Listing 4.7.

```
std::map <std::string, std::tuple<tiValue,tiValue,tiValue>,
        cmpLocation> timingMap;
```

Listing 4.7: Datastructure for Storing Timing Areas

The syntax of the configuration file used for the access times, as well as both `ReadConfig` functions, are almost identical to the ones of the `EnergyController`, with only negligible changes. Only the differing components are therefore described.

GetTimingValue

The `GetTimingValue` function is the corresponding function to the `GetEnergyValue` function of the energy controller. It is used to return the access time of a specific request. Similarly, first the location of the request is used to find the corresponding area in the map. This is also done by finding the highest location that is strictly less than the given location. The distinction between the operation type of the request is also made.

Here, the assumption is made that all accesses to the memory cells needed in this request are made simultaneously. Therefore, for a read access, simply the corresponding read access time from the map is returned. For a write access, the access time is calculated by the `DetermineWritingTime` function.

`DetermineWritingTime`

This function returns either the time needed for a set access or a reset access. To do this, the `DetermineWritingTime` function reads the data of the write request and checks if set or reset operations are needed. If only reset operations occur, the reset access time is returned. Correspondingly, if only set operations occur, the set access time is returned. If both set and reset operations are needed, the greater of the two times is returned.

Integration

Of course, both controllers have to be integrated into the simulator. To achieve this, a number of changes have to be done in already existing classes. The most important modification is to allow requests to store an attribute for its energy consumption as well as its timing characteristics. This is simply done by adding the attributes, get and set functions to the `NVMainRequest` class.

The simulator also needs to know which of the properties should be simulated. For this, the configuration flag `MemoryAreas` is set in the system configuration file. The option `energy` indicates to only simulate the energy consumption, while the option `time` only simulates access times. Additionally, the option `both` enables the simulation of energy and timing properties. For all three options, a flag is set to specify the location of the corresponding configuration file that holds the defined values. These flags are `AreaFile`, `EnergyFile` and `TimingFile`. If a config is needed, but the corresponding file is not set, default values are used. Also, if only energy or only timing values should be used, the other values are marked as invalid by assigning negative values.

To enable a memory controller to use the new `Energy` and `TimingController`, it first has to be able to initialize them. The `SetConfig` function of the memory controller has to realize the extraction of the file names of the configuration files, which should be used for the energy and timing values. After the configuration files are set, the memory controller then has to trigger the reading of the configuration files. Both the `Energy` and `TimingController` realize this with the `ReadConfig` and `ReadJoinedConfig` function. An example of the case distinction of choosing the right configuration file can be seen in Listing 4.8.

```
if( conf->KeyExists( "MemoryAreas" )){
  /* only energy is selected */
  if (conf->GetString( "MemoryAreas" ) == "energy"){
    if( conf->KeyExists( "EnergyFile" )){
      if ( conf->GetString( "EnergyFile" ) == ""){
        std::cout << "NVMain: No energy configuration file set" << std::endl;
      } else {
        std::string energyFile  = conf->GetString( "EnergyFile" );
        if (energyFile[0] != '/'){
          energyFile = "/gem/nvmain/Areas/Energy/";
          energyFile += conf->GetString("EnergyFile");
        }
        std::cout << "Using EnergyFile: " << energyFile << std::endl;
        energyController.ReadConfig(energyFile);
      }
    } else {
      energyController.SetDefaultValues();
      std::cout << "Missing energy configuration file, using default values."
        << std::endl;
  }
  /* set invalid time values to show trace writer time is not simulated*/
  timingController.SetInvalidValues();
...
```

Listing 4.8: Selecting the Configuration File for Different Energy Areas

During the simulation, the memory controller can use both controllers to set the energy and timing parameters of requests. Because the `IssueCommand` function controls the scheduling of requests, it is also used for this purpose. If the parameters are assigned at another point in the simulation, e.g. in the actual read and write functions in the `SubArray` class, the calculation would occur after the information of the request is written into the traceline. Therefore, to be able to use the results, the calculation has to be done in the memory controller. To set the energy and timing values, the set functions of the `NVMainRequest` class are used. The values are calculated by the `GetEnergyValue` or `GetTimingValue` functions of the controllers. An example memory controller that uses the `Energy` and `TimingController` is realized in the new class `MemoryAreasController`. Because the scheduling is not important in this thesis, it simply places each incoming memory request into the transaction queue. This corresponds to a simple first-come-first-serve scheduling algorithm.

4.1.3 Testing Calculation and Definition Correctness

To ensure the energy and timing controller are working correctly, they have to be tested accordingly. This is done by writing a test program for each controller, testing the `GetEnergyValue` respectively the `GetTimingValue` function. The testing of the underlying structure for storing and retrieving the right values in the map is done beforehand. As the test cases are very simple and only provide positive results, they are not elaborated further.

Energy Areas

As described in Section 4.1.2, the `GetEnergyValue` functions returns the energy consumption of a specific request. For a read request, only the selection of the right value from the map and the multiplication with the correct bit number of the data has to be tested. For a write request, the energy consumption is calculated through the number of set and reset operations.

```
volatile long value = 0x0;
for (long i = 0; i < 64; i++){
        value = (value << 1) + 0x1;
}
```

Listing 4.9: Test Function for DetermineWriteEnergy

To test this calculation, a simple function is created that writes data at the same memory location. The written data hereby contains one more set operation per access. Whereas the first write access contains zero set operations, the second one contains one set operation, and so on. This is in done in a loop for a number of times to be able to plot the output. To realize it is exactly one more set operation per loop iteration, bit shift is used, as displayed in Listing 4.9. Per iteration, the value is shifted by one bit and another one is added. The resulting value consists of only ones, in the binary representation.

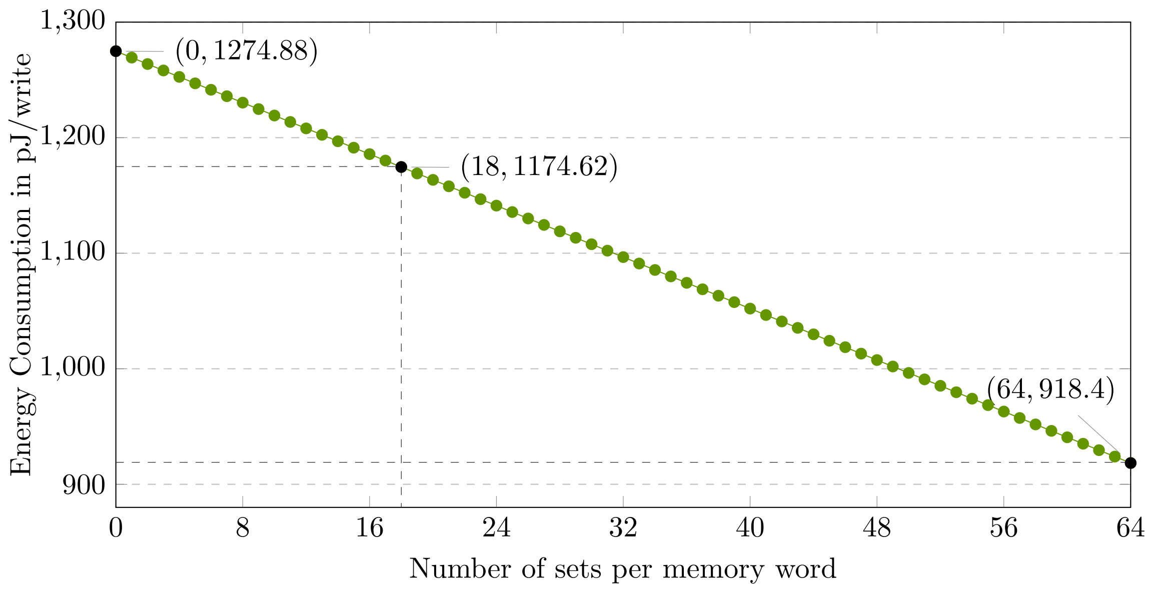


Figure 4.1: Results of Testing DetermineWriteEnergy

In Figure 4.1 the energy consumption of the write accesses is displayed depending on the number of set operations per memory word. Every data point represents a single write access, with its energy consumption set into relation to its number of needed sets to write the word. The plot shows the energy consumption per written word declining linearly. This corresponds to the rising number of set operations, which consume less energy than the reset operations.

To demonstrate the displayed output data making not only sense in itself but also being correctly calculated, the calculations of a few values are displayed in Listing 4.10. The calculated values match the ones returned by the simulator, shown in Figure 4.1. It can be observed that the energy consumption of these words always differ by the value of 5.57. This is the difference between the set and reset energy consumption used in this test ($19.92 - 14.35 = 5.57$).

```
Memory location of "value": 0 0 0 0 3 255
Definition of energy area:
        Physical Location 0 0 0 0 3 2 252
        setEnergy: 14.35 pJ/Bit
        resetEnergy: 19.92 pJ/Bit

written data (8 Byte = 64 Bit) :
        0x0000000000000000: 64 resets , 0 sets
                energy consumption: 64 * 19.92 + 0 * 14.35 = 1274.88
        0xffff300000000000: 46 resets , 18 sets
                energy consumption: 46 * 19.92 + 18 * 14.35 = 1174.62
        0xffffffffffffffff: 0 resets , 64 sets
                energy consumption: 0 * 19.92 + 64 * 14.35 = 918.4
```

Listing 4.10: Calculation of the Energy Consumption of a Write Operation

Timing Areas

As described in Section 4.1.2, the `GetTimingValue` function returns the memory access time of a specific request. For a read request, it has simply to be tested that the right value is selected from the map. For a write request, it has to be tested that the longest time of the executed operations is returned. This corresponds to the test cases that are displayed in Figure 4.2. In the test case (set > reset) a read access lasts 60 ns, a set access 120 ns and a reset access 50ns. In the other test case (set < reset) the times of set and reset are flipped, a read access lasts 60ns, a set access 50ns and a reset access 120ns.

operation	output (set > reset)	output (set < reset)
write 0x0	50	120
read 0x0	60	60
write 0xffffff	120	120
read 0xffffff	60	60
write 0xffffffffffffff	120	50
read 0xffffffffffffff	60	60

Table 4.2: Results of Testing DetermineWritingTime

It can be seen that the read times are always the same and match to the defined time of the timing area. For the value $0x$ffffff saved in the memory as $0x$ffffff000000000, the writing time stays the same. This confirms that the writing time in the case of mixed set and reset operations is calculated correctly, because the maximum value is returned. The noteworthy cases are the writing times of the values $0x0$ and $0x$ffffffffffffffff. These values are the opposite of each other in terms of writing type. While the $0x0$ value only requires reset operations, the $0x$ffffffffffffffff value only requires set operations. Therefore, the output shows correctly that the writing times only depend on the defined set and reset times for this timing area.

4.2 Trace Writer

Trace writers are used in the simulator to generate output for a simulation in a defined way. There are already existing trace writers available in the simulator, but none of them use the changes in the simulator described in Section 4.1. Because the simulator already supports the concept of trace writer, it is possible to simply add more trace writers. The requirements to be fulfilled are described in the following section. After this, the new trace writers are presented and consequently tested.

4.2.1 Requirements

To get the most of the defined energy and timing areas, it is beneficial to generate output based on each individual memory access and its energy and timing properties. Furthermore, these individual accesses should be summarized. Because the column is the smallest addressable unit in NVMain 2.0, the values can be only calculated per column and not per single memory cell. For all used columns the read and write accesses, as well as the energy consumption and the overall latency for memory accesses, can be calculated from the individual accesses. The total read and write accesses done during the simulation and the total energy and time consumption should also be calculated.

4.2.2 AreaTraceWriter

The best way to satisfy all requirements described in Section 4.2.1 is to create a separate trace writer for each of them. Therefore, three new trace writers are created. The first, called `AreaAccessTraceWriter`, displays all memory accesses with their corresponding properties. The operating principle of displaying every access is explained in the next section. All output is written into the trace file specified in the configuration file. The second trace writer is the `AreaStatisticsTraceWriter`. It calculates the number of read and write accesses, energy and time consumption per memory word, as well as the total values of the four parameters. Similarly to the first trace writer, the output is only written into the specified trace file. The last trace writer is the `AreaTraceWriter`. It combines the output of both previously mentioned trace writer. Therefore, it displays not only the details of each memory access, but also displays the statistics. Here, the output is not exclusively written into the specified trace file. To enable the output files to be easily used further, it is important that the output format stays the same. Hence, the memory accesses and statistics cannot be written into the same file, because they have not the same format and information. To solve this problem, only the memory accesses are written into the specified trace file. The statistics are written into an additionally created file, which has the name of the trace file added with an '-statistics'. In the following, the operating principle of the memory accesses and statistics are explained.

Memory Accesses

Some trace writers in the simulator already display information about every memory access. For example, the `NVMainTraceWriter` displays the following properties of every read or write memory access: cycle within the simulation, operation type, address of the access, program counter, data written or read, the old data that is possibly overwritten and the thread ID. For the purpose of simulating different energy and timing modes, not all of these properties are relevant. Namely, the old data and thread ID are of no use in this case and are therefore omitted. Additionally, the energy and time consumption are, of course, important, as well as the physical location of the memory request. Accordingly, the complete list of properties is: cycle, operation type, address, program counter, data written or read, physical location of the access (containing channel, rank, bank, sub-array, row and column) and the energy and time consumption. Only read and write accesses are displayed by the trace write because the energy and timing areas only set values for these accesses.

For every completed memory request, its details are written into a traceline and handed over via the `SetNextAccess` function to the trace writer. To write the properties of the memory access into the trace file, the `WriteTraceLine` function is called. It writes the specified information into the given trace file. All entries for the trace file are written during the simulation.

In Listing 4.11, an example output of the `AreaAccessTraceWriter` is displayed. Here, the first four lines represent the first four read accesses. The last three lines represent the last three simulated read accesses. In between are accesses listed that occur in the middle of the simulation. Each line consists of the previously mentioned parameters, in this exact order, separated by whitespace. The parameters of the physical location are grouped together like for example "0:0:0:0:3:253" to easier identify the individual values.

```
0 R 0x0 0x80000000 c000005800c018d5c00000581f0000919a29009400000000000009800800
     0000000b8bf0280000000000000000000000000000000000000000000000000000
     0:0:0:0:0:0 1264.64 60
83 R 0x40 0x80000040 0000000000000000000000000000000000000000000000000000000000000000
       0000000000000000000000000000000000000000000000000000000000000000
       0:0:0:0:0:1 1264.64 60
89 R 0xa640 0x8000a678 200200d000e845f949dbff97e20300aa000100900120269 1e00302aa
       44dbff97e20300aa200200d001c045f9e00302aafedbff9700000014fd7bb8a9fd030091
       0:0:0:0:0:665 1259.52 59.8
90 R 0x18 0x80000000 0098008000000000 0:0:0:0:0:0 158.08 60
...
23193071 W 0x63f60 0x80001060 80ff00000a000000 0:0:0:0:3:253 1213.61 122.1
23193071 W 0x63f68 0x80001060 d413008000000000 0:0:0:0:3:253 1230.32 122.1
23193071 W 0x63f7f 0x80001068 6f 0:0:0:0:3:253 125.94 122.1
23193071 R 0x63f7f 0x8000106c 6f 0:0:0:0:3:253 21.68 63.4
...
39740956 R 0x2bdb8 0x80001684 d8bd028000000000 0:0:0:0:1:758 178.56 63.6
39740956 R 0x2bdc0 0x80001684 ecae008000000000 0:0:0:0:1:759 178.56 63.6
39740971 R 0xaec0 0x8000aef4 2bd9ff97e20300aa0001009001003491e00302aa26d9ff97e20300
         aa200200d001c045f9e00302aae0d9ff97000080d22d1500941f2003d5fd7bc2a8c0035fd6
         0:0:0:0:0:699 1428.48 63.6
```

Listing 4.11: Example Output of the AreaAccessTraceWriter

Statistics

To create the statistics of a simulation, the four already mentioned parameters are calculated: the number of reads and writes, as well as the energy and time consumption per memory word.

The first step is to define a data structure that is able to store all these parameters. Similarly to Section 4.1.2 the data structure `map` is used, with the memory location acting as the unique key. The four parameters are then stored in a tuple associated with the memory location. With every completed memory request, the `SetNextAccess` function is called with details of the request written into a traceline. In the trace writers, that output the statistics of a simulation, this function then calls the `UpdateMap` function. It is responsible for calculating the individual statistics per memory cell. The function first calculates the memory location of the request based on the properties of the traceline. Then it checks if an entry with the calculated memory location is already saved in the `memoryMap`. If that is the case, the entry gets updated. If not, then a new entry for this memory location is created. To update an entry, the count of either read or write access is increased by 1 and the energy and time consumption of the access is added to the existing sums.

After the simulation is done, the memoryMap is iterated through and all parameters are summed up to calculate the total values. All entries of the memoryMap as well as the calculated total values are written into the file. Additionally, a CSV file with the same output is generated that can be used for further analysis. In Listing 4.12, an example output of the `AreaStatisticsTraceWriter` is displayed. The first line shows the order of the memory layout, to avoid confusion. The following lines each represent the statistics of one specific memory location, i.e. of one column. To make the statistics more accessible, the parameters are all named. At the bottom, the total values of the whole simulation are presented.

```
Memory Layout: Channel, Rank, Bank, Sub-Array, Row, Column
Location: 0 0 0 0 0 0 Reads: 7 Writes: 0 Energy: 6639.36 Time: 420
...
Location: 0 0 0 0 0 79 Reads: 10601 Writes: 0 Energy: 13189340.16 Time: 620158.5
Location: 0 0 0 0 0 80 Reads: 1 Writes: 0 Energy: 1244.16 Time: 58.5
Location: 0 0 0 0 0 81 Reads: 78 Writes: 0 Energy: 97044.48 Time: 4563
Location: 0 0 0 0 0 82 Reads: 276 Writes: 0 Energy: 343388.16 Time: 16146
Location: 0 0 0 0 0 83 Reads: 1 Writes: 0 Energy: 1244.16 Time: 58.5
...
Location: 0 0 1 0 2 737 Reads: 0 Writes: 8 Energy: 10199.04 Time: 424.8
Location: 0 0 1 0 2 738 Reads: 0 Writes: 5 Energy: 6374.4 Time: 265.5
Total Reads: 244852
Total Writes: 72885
Total Energy: 230444567
Total Time: 23169174
```

Listing 4.12: Example Output of the AreaStatisticsTraceWriter

Integration

To integrate the new trace writer into the simulator, the most important adjustment is to add energy and timing properties to the traceline. As mentioned in Section 3.2, the trace writer receives a traceline for every memory access and uses the properties of the traceline to generate or calculate output. Therefore, the defined properties for the energy and timing of a single access are added with the typedefs `enValue` and `tiValue`. Additionally, the energy and timing values of a traceline have to be set with the values of the corresponding memory request. The traceline is generated in the `PrintPreTrace` function of the `nvmain.cpp` class. Therefore, the energy and timing values of the traceline are also set in this function.

To select one of the trace writers in the configuration of the simulator, the keywords `AreaTrace`, `AreaAccessTrace` and `AreaStatisticsTrace` are defined in the `TraceWriterFactory` class.

4.2.3 Testing Memory Accesses and Statistics

The trace files act as one of the main output sources of the simulation. Ensuring the output is correct is therefore very important. The simulator cannot effectively be used to simulate different energy and timing modes if the output is wrong. The two important features of the presented trace writers, in need of testing, are the output of each memory access, as well as the calculation and output of the statistics.

Memory Accesses

Checking the information of every memory request is not possible due to the amount of collected data. Therefore, testing is done by verifying the memory accesses generated by the application are displayed in the trace file. As an example, the `simple_test` function displayed in Listing 4.13 is used for verification. The function cycles through an array ten times and increases the value of the element by one with each access. In this case, the array has the size of 100, therefore after the function is finished, the value ten is stored in 100 memory words.

```
#define TEST_SIZE 100

volatile uint64_t test_array[TEST_SIZE];

void simple_test(){
   for (unsigned int i = 0; i < TEST_SIZE * 10; i++) {
        test_array[i % TEST_SIZE]++;
    }
}
```

Listing 4.13: Test Function

The test function is simulated once and the trace file with the individual memory accesses is generated. To be able to select all memory accesses related to the test function, the program counter of the instructions were extracted. For this, the assembler mnemonics for the machine instructions of the simulator are displayed by using `aarch64-linux-gnu-objdump`. The specific assembler mnemonics of the test function are shown in Listing 4.14. The instructions generating memory accesses are color marked. These include the reading and writing of the function parameters in the memory, like the loop counter `i` or elements of the `test_array`.
A section of the corresponding entries in the trace file is displayed in Listing 4.15. The displayed entries only include the read and write access of the counter `i`, as well as the read and write accesses of old and new values of the `test_array`. At the beginning of the section, the counter has the value "0x63", which corresponds to "99" in decimal representation. After that, the value of the element at index 99 is read and increased by one. Then, the counter is increased to "0x64" or "100" in decimal representation. Because 100 mod 100 equals zero, the next accessed element is at index zero. The last lines in the listing therefore show the value at index zero getting increased for a second time. Hence, the right information of the simulated program is written into the trace file.

Statistics

To ensure the correctness of the calculated statistics, the calculation for one single memory word is done by hand and compared to the result of the simulation. For this, the example from the previous section is used. As the memory word, a part of the array is used. To be

```
00000000800429ac <simple_test()>:
    800429ac:   d10043ff    sub    sp, sp, #0x10
    800429b0:   b9000fff    str    wzr, [sp, #12]
    800429b4:   b9400fe0    ldr    w0, [sp, #12] //load value of i from memory
    800429b8:   710f9c1f    cmp    w0, #0x3e7
    800429bc:   540002e8    b.hi   80042a18 <simple_test()+0x6c>  // b.pmore
    800429c0:   b9400fe1    ldr    w1, [sp, #12] //load value of TEST_SIZE from memory
    800429c4:   5290a3e0    mov    w0, #0x851f                    // #34079
    800429c8:   72aa3d60    movk   w0, #0x51eb, lsl #16
    800429cc:   9ba07c20    umull  x0, w1, w0
    800429d0:   d360fc00    lsr    x0, x0, #32
    800429d4:   53057c00    lsr    w0, w0, #5
    800429d8:   52800c82    mov    w2, #0x64                      // #100
    800429dc:   1b027c00    mul    w0, w0, w2
    800429e0:   4b000020    sub    w0, w1, w0
    800429e4:   b00000a1    adrp   x1, 80057000 <rgnNodes+0x1d0>
    800429e8:   91060021    add    x1, x1, #0x180
    800429ec:   2a0003e2    mov    w2, w0
    800429f0:   f8627821    ldr    x1, [x1, x2, lsl #3] //load value of element from array
    800429f4:   91000422    add    x2, x1, #0x1
    800429f8:   b00000a1    adrp   x1, 80057000 <rgnNodes+0x1d0>
    800429fc:   91060021    add    x1, x1, #0x180
    80042a00:   2a0003e0    mov    w0, w0
    80042a04:   f8207822    str    x2, [x1, x0, lsl #3] //write new value of element in array
    80042a08:   b9400fe0    ldr    w0, [sp, #12] //load value of i from memory
    80042a0c:   11000400    add    w0, w0, #0x1
    80042a10:   b9000fe0    str    w0, [sp, #12] //write new value of i in memory
    80042a14:   17ffffe8    b      800429b4 <simple_test()+0x8>
    80042a18:   d503201f    nop
    80042a1c:   910043ff    add    sp, sp, #0x10
    80042a20:   d65f03c0    ret
```

Listing 4.14: Assembler Code of Test Function

```
19900156 W 0x63fec 0x80042a10 63000000 0:0:0:0:3:255 615.16 122.1
19900156 R 0x63fec 0x800429b4 63000000 0:0:0:0:3:255 86.72 63.4
19900735 R 0x63fec 0x800429c0 63000000 0:0:0:0:3:255 86.72 63.4
19900843 R 0x57498 0x800429f0 0000000000000000 0:0:1:0:2:466 173.44 63.4
19900858 W 0x57498 0x80042a04 0100000000000000 0:0:1:0:2:466 1269.31 122.1
19900864 R 0x63fec 0x80042a08 63000000 0:0:0:0:3:255 86.72 63.4
19900951 W 0x63fec 0x80042a10 64000000 0:0:0:0:3:255 620.73 122.1
19900951 R 0x63fec 0x800429b4 64000000 0:0:0:0:3:255 86.72 63.4
19901530 R 0x63fec 0x800429c0 64000000 0:0:0:0:3:255 86.72 63.4
19901638 R 0x57180 0x800429f0 0100000000000000 0:0:1:0:2:454 178.56 63.6
19901653 W 0x57180 0x80042a04 0200000000000000 0:0:1:0:2:454 1271.85 122.6
```

Listing 4.15: Memory Accesses of Counter

precise the memory word at the physical location ”0 0 1 0 2 454” (see last line of Listing 4.15). Extracting all entries with this location provides a list with all memory accesses done at that location, together with their energy and time consumption. Counting the read and write accesses results in the number of 88 write accesses and 80 read accesses. These numbers are equal to the total reads and writes that the statistics computed for that memory word. The corresponding entry is presented in Listing 4.16.

The 64-bit memory word of the specified location stores eight elements of the array. Every element is read ten times by the test function, resulting in the mentioned 80 read accesses. Furthermore, every element is accessed eleven times to write a value. The first time at

```
...
Location: 0 0 1 0 2 452 Reads: 0 Writes: 8 Energy: 10219.52 Time: 427.2
Location: 0 0 1 0 2 453 Reads: 0 Writes: 8 Energy: 10219.52 Time: 427.2
Location: 0 0 1 0 2 454 Reads: 80 Writes: 88 Energy: 125939.28 Time: 15323.2
Location: 0 0 1 0 2 455 Reads: 80 Writes: 88 Energy: 125939.28 Time: 15323.2
Location: 0 0 1 0 2 456 Reads: 80 Writes: 88 Energy: 123585.52 Time: 14656.8
...
```

Listing 4.16: Section of Statistics Trace

the initialization with the value zero and after that ten times, each to increase the value by one. The different accesses and their respective energy consumption is shown in Table 4.3. Here, the energy consumption per different access is calculated based on the count of the access and the energy costs of writing or reading it. The values are taken from the corresponding trace file.

	count	energy	calculation	total energy consumption
read accesses	80	178.56	$80 * 178.56$	14284.8
write "0"	8	1277.44	$8 * 1277.44$	10218.52
write "1"	8	1271.85	$8 * 1271.85$	10174.8
write "2"	8	1271.85	$8 * 1271.85$	10174.8
write "3"	8	1266.26	$8 * 1266.26$	10130.08
write "4"	8	1271.85	$8 * 1271.85$	10174.8
write "5"	8	1266.26	$8 * 1266.26$	10130.08
write "6"	8	1266.26	$8 * 1266.26$	10130.08
write "7"	8	1260.67	$8 * 1260.67$	10085.36
write "8"	8	1271.85	$8 * 1271.85$	10174.8
write "9"	8	1266.26	$8 * 1266.26$	10130.08
write "a"	8	1266.26	$8 * 1266.26$	10130.08
sum	168	–	–	125939.28

Table 4.3: Calculation of Total Energy Consumption per Memory Word

The final result of all accesses matches the value shown in the statistics file (see List. 4.16). Therefore, the calculation of the total energy consumption in the statistics trace writer correctly considers all relevant data and calculates it correctly.

For the latency calculation, the write accesses can simply be divided into two categories: accesses with only reset operations and accesses with set and reset operations. The only write accesses that consist of only reset operations are the first eight writes, that initialize the array. After that, all accesses consist of both set and reset operations. Therefore, the calculation in Table 4.4 for the total latency is smaller than the calculation of the energy consumption. Similarly, the shown calculation yields the same result as the one computed by the statistics trace writer (see List. 4.16).

	count	latency	calculation	total latency
read accesses	80	63.6	$80 * 63.6$	5088
write accesses (only reset)	8	53.4	$8 * 53.4$	427.2
write accesses (set and reset)	80	122.6	$8 * 122.6$	9808
sum	168	–	–	15323.2

Table 4.4: Calculation of Total Latency per Memory Word

4.3 Write Mode Interface

Selecting different write modes during run-time can be used to improve the performance and energy efficiency of NVMs, as presented in Chapter 2. The general idea of the operating principle, capable of this, is therefore explained first. The subsequent changes in the implementation are presented and tested.

4.3.1 General Idea

To modify the write mode at run-time an interface can be used. A simple version of the interface uses a specific location in the memory for configuration. Depending on the values written in this location, the write mode of a specific memory area is set. Building up on the changes done in Section 4.1, the interface has to be able to specify the physical location of the write mode. Otherwise, the right energy and timing areas cannot be updated. The number of available write modes should be to the power of two to make the storing in hardware easier. The selection of different write modes is then done directly in the code of the simulated program.

4.3.2 Implementation Adjustments

Using a specific location for configuration means the simulated program has to have access to this location. Therefore, a specific 64-bit word is used in the program to modify the write modes. Depending on the values stored in the word, the write modes are adjusted while the program is simulated. The first step to use the interface is to write a previously defined magic number into the bit word to allow the interface to store its physical address. After that, every following write access to this address triggers the interface to interpret the newly written values for the configuration of the write mode. The used bit word has to store seven different parameters in 64 bit, namely the location with its six parameters denoting the area to be updated and the write mode. To simplify the creation of these words, a packed structure is used. The declaration of this struct is shown in Listing 4.17. Every parameter has the size of four bits, except the row and column. With four bits, numbers up to 15 can be represented. This is enough to represent the channel, rank, bank and sub-array, as well as the write mode. Typically, none of these parameters exceed the number of 16. However, the number of rows and columns can be much higher, therefore they need more bits to be able to be represented correctly. The packed attribute means the compiler does not add padding to the struct, which results in it being exactly 64 bit long.

```
struct __attribute__ ((packed)) config_word_interface{
unsigned long long writeMode:4;
unsigned long long column:22;
unsigned long long row:22;
unsigned long long subarray:4;
unsigned long long bank:4;
unsigned long long rank:4;
unsigned long long channel:4;
};
```

Listing 4.17: Interface Struct

The implemented selection of write modes is currently limited to four. As already mentioned, the number of available write modes should be a power of two. If more write modes are needed in the future, the modes and the overall number of them can easily be modified. The write mode has already four bits in the interface at its disposal. The function setting the write modes at the start of the simulation is explained later. To define the values for the write modes, the header file `interface-config.h` is used. A part of an example configuration is shown in Listing 4.18. Here, the standard values for the read access is defined at the top, followed by the values for write mode 1. For both energy and time consumption, the values for set and reset have to be specified. This has to be done for every mode defined in the system, which are currently four modes.

```
#define INTERFACE_STANDARD_READENERGY 2
#define INTERFACE_STANDARD_READTIME 60.2

#define INTERFACE_MODE_1_SETENERGY 14.1
#define INTERFACE_MODE_1_RESETENERGY 19.8
#define INTERFACE_MODE_1_SETTIME 120.5
#define INTERFACE_MODE_1_RESETTIME 50.5
```

Listing 4.18: Write Mode Example

MemoryAreaController

Because the memory controller is responsible for handling the `Energy` and `TimingController`, a part of the functionality for interface has to be implemented there. As mentioned in Section 4.1.2, the `MemoryAreaController` is realized as an example memory controller and therefore also used for the interface. The main function added for the interface is the `UpdateAreas` function. It is responsible for extracting all needed information from the bit word, written in the application. This function gets called in the `IssueCommand` function, together with the setting of energy and timing values. The first purpose of the `UpdateAreas` function is to set the reference address for the interface. This happens as soon as the magic number defined there is written by the program. After this is done, a flag is set to indicate that the address is set and that all write accesses to this address indicate an update of the write mode. If a write access to this specific address is registered, the data of this memory request is extracted. To use this data, a helper function is needed and realized in the `ConvertData` function. Because the data in ARM is stored in a little endian system, it has to be converted to a big endian system to be able to cast the values correctly. This is simply done by reordering the bytes to a big endian system. Without the conversion, for example, the data "3b01" would be cast to "15105" even though the original written value is "315" or "0x013b". This occurs because the little endian version of "013b" is "3b01". From the converted data, the needed information can simply be read by computing the substrings with the predefined lengths. At the end, the obtained information is forwarded to the energy respectively timing controller. An example of this process is presented in Figure 4.2.

The information for the interface is first written into the struct in the application, and the corresponding request is then processed by the memory controller. Because the address of the request matches the previously set interface address, the memory controller extracts the data of the request to use it for the interface. The figure shows the process of extracting the different information parts, by using the parameter sizes of the declared struct. All parameters except the row and column can be used without further computation. Because the encoded row and column are both 22 bit long, they have to be split in the binary representation. After this is done, the correct values can be converted and passed to the energy and timing controller.

Energy and Timing Controller

While the `MemoryAreaController` extracts the information for the interface and triggers the update process, the actual update of the areas takes place in the `EnergyController` and `TimingController`. After the area configurations files are read, the values of the write modes are set by the `SetWriteModi` function. As explained, they are defined in the `interface-config` header file and are used to create the `writeModi` map. Because a read access has minimum energy consumption and latency, the read values are not modified by the write modes. The actual update of the write modes is realized in the `UpdateEnergyMap` or `UpdateTimingMap` function. These functions are called by the `MemoryAreaController` to signal an update. The passed parameters include the location of the area and the write mode that should be used. The function then first locates the defined area, corresponding to the given location, in the map. Then, the values of this area are updated to the values of the specified write mode.

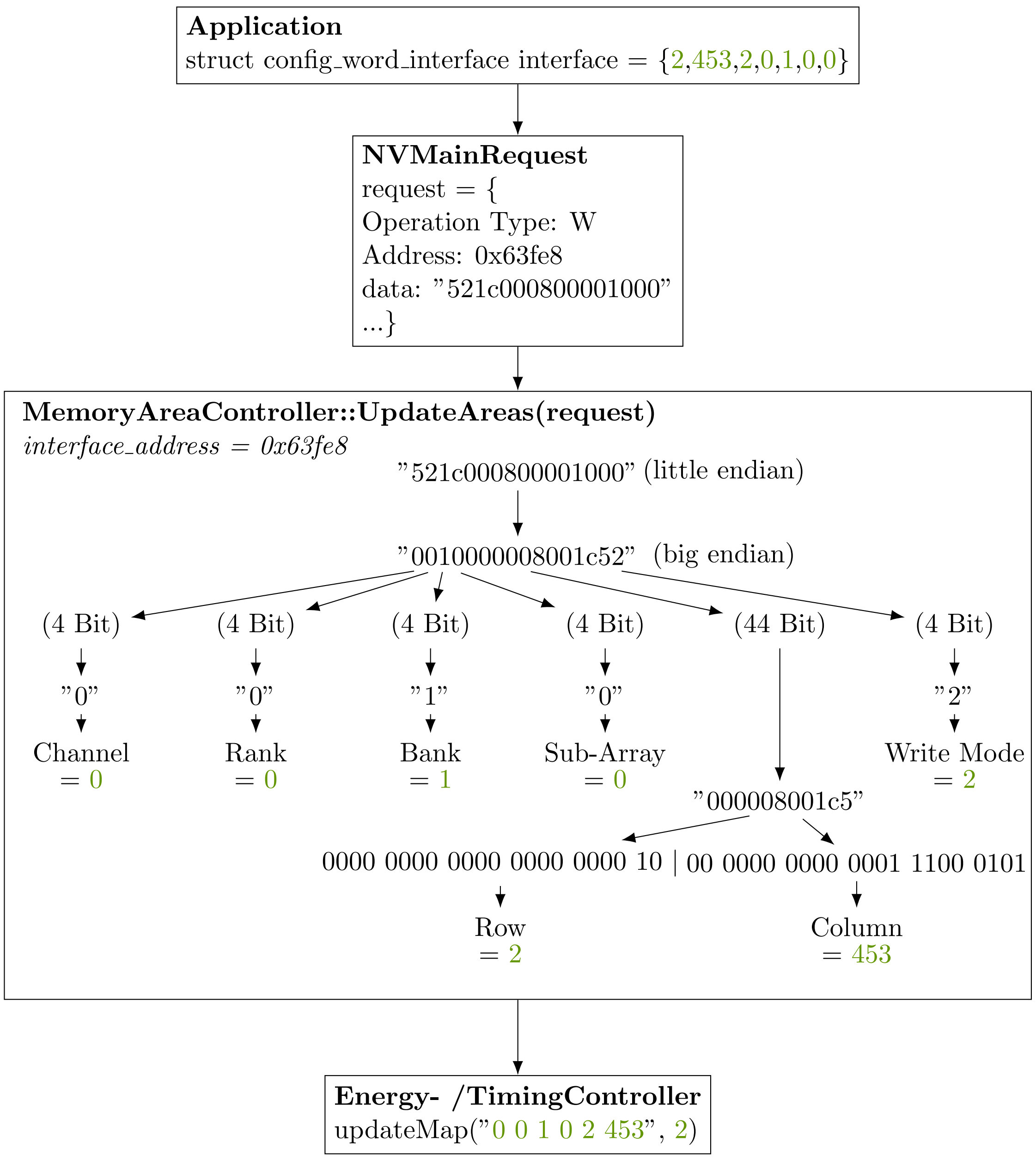


Figure 4.2: Interface Parameters Passing Through the Simulator

4.3.3 Testing Different Write Modes at Run-Time

Calculating the energy and time consumption to verify the simulated values gets increasingly difficult. This is due to the different defined areas, as well as the changing of write modes at run-time. Therefore, to test the implementation of the interface, different points in the trace files are analyzed. The impact of the write modes on the overall energy and time consumption is also discussed.

For both tests, the following example is executed: in the function `use_interface` the write modes are updated, and the test function created in Section 4.1.3 is called a few times to create a more noticeable difference in the results. In Listing 4.19, a section of the trace file is presented. Line 2 and 3 show the reading of the variable in the application code and the subsequent writing of the magic number. Therefore, address "0x63fe8" is used for the configuration of the interface. Line 5 shows a write access from the test function with its written data, energy consumption and latency. In line 7 and 8, the interface gets used for the first time. Here, the area of the test function (0:0:0:0:3:255) is configured to write mode 3, as marked in the listing. After that, the same write access is executed as before with the same written data (line 10). Despite writing the same data, the energy consumption and latency is different to the first execution, because of the write mode modification. The same effect can be seen a second time (line 15) through changing the write mode to mode 2 (line 12 and 13). Therefore, it is shown that the usage of the interface in terms of individual accesses is working correctly.

```
 1  ...
 2  19820635 R 0x50b30 0x800443f8 0300000000003333 0:0:1:0:2:44 159.36 60.5
 3  19820650 W 0x63fe8 0x800443fc 0300000000003333 0:0:0:0:3:255 1213.6 121.5
 4  ...
 5  19859923 W 0x63fc0 0x80044238 fffffffff000000 0:0:0:0:3:255 1043.2 121.5
 6  ...
 7  19881032 R 0x50b38 0x8004440c f30f000c00000000 0:0:1:0:2:44 159.36 60.5
 8  19881048 W 0x63fe8 0x80044410 f30f000c00000000 0:0:0:0:3:255 1202.24 121.5
 9  ...
10  20360801 W 0x63fc0 0x80044238 fffffffff000000 0:0:0:0:3:255 259.8 30.125
11  ...
12  20382209 R 0x50b40 0x800445a4 f20f000c00000000 0:0:1:0:2:45 159.36 60.5
13  20382224 W 0x63fe8 0x800445a8 f20f000c00000000 0:0:0:0:3:255 301.125 30.125
14  ...
15  20861842 W 0x63fc0 0x80044238 fffffffff000000 0:0:0:0:3:255 519.6 60.25
16  ...
```

Listing 4.19: Interface Accesses

Analyzing the impact of the write modes on the overall energy and time consumption, the write modes have to be compared. To compare all write modes and the absence of them, the program simulates the `use_interface` function for all five cases. Because the used test function does not use many memory areas, only the areas containing the columns 251 to 255 of row 3 are modified during this test. This is the memory area used by the simulated program and is therefore the only area that is focused on here. The results of the simulations are presented in Figure 4.3.

The first of the two plots displays the total energy consumption per column in pJ, whereas the second plot displays the total latency per column in ns. Each of the four write modes, as well as the version with no mode, has an own color for its bars. The difference between the write modes is evident. While the "normal" mode, here denoted as mode 1, displays results similar to the results with no used mode, the results of the other write modes are vastly different.

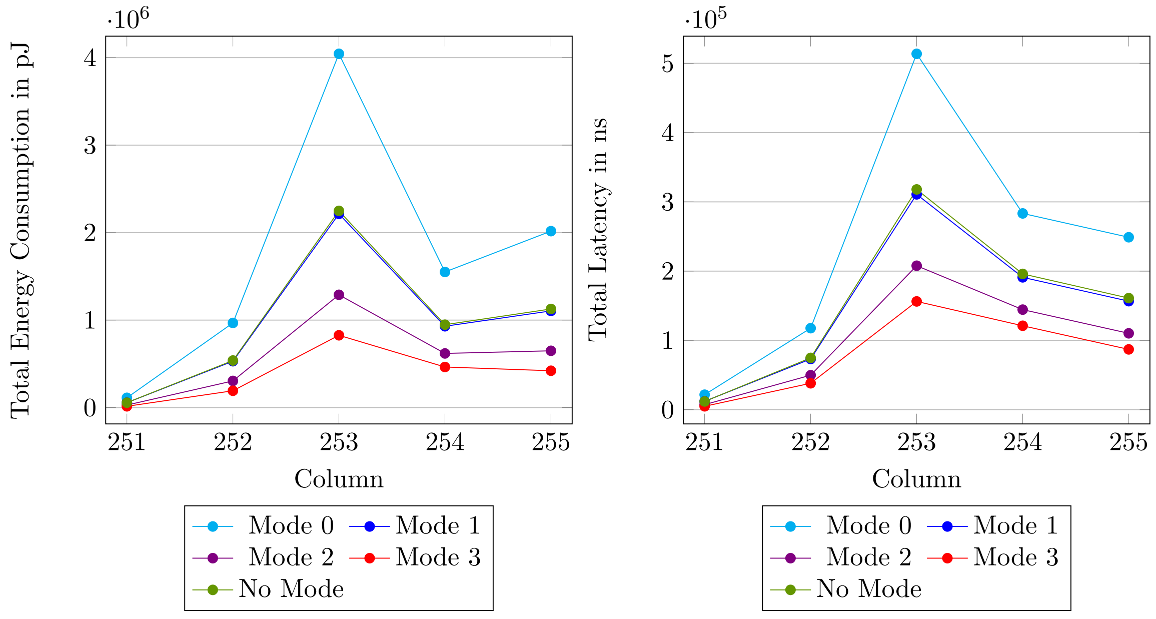


Figure 4.3: Energy and Time Consumption per Column

The similarity between mode 1 and using no mode can be explained by the very similar values in the mode definition and the area configuration file. For example, for column 255, the total energy consumption without a write mode is 1,127,165.26 pJ and with the normal write mode 1,105,027.072 pJ. This is a difference of 22,138.188 pJ which, in proportion, is small. The same results can also be seen in the total latency of the columns.

The "slow" mode, mode 0, uses twice as much write energy and time as mode 1. However, the plots show both total energy consumption and total latency being not twice as high as that of mode 1. There are two main reasons for this. The first reason is that writes occurring in these columns, before the application is called, are unaffected by the changes of write modes. Therefore, a baseline of energy and time consumption is added to the total values. The second reason is that the total energy consumption and latency include the write accesses and also the read accesses. Because the different write modes do not update the values for the read energy and time consumption, these values are the same for all write modes. For example, column 253 gets accessed for 1844 writes, but also for 1498 reads. Although a read access in NVMs usually consumes much less energy and time than a write access, the impact on the results is noticeable nevertheless. To use specific values: the total energy consumption of column 252 with mode 0 is 967455.8834 pJ, with mode 1 it is 530092.0407 pJ. Therefore, the factor between these two total values is 1.8, which is close to the factor of two that is used for values of mode 0.

The same observations can be made for the write mode 2 and 3. While write mode 2 uses half the energy and time of mode 1, mode 3 only uses a quarter of them. The total energy consumption of column 252 with mode 2 is 304781.4018 pJ and with mode 3 192114.8169 pJ. The resulting factors are therefore 0.57 between mode 1 and 2, and 0.36 between mode 1 and 3. As explained before, the deviations from the original factors, used for the calculations of the write modes, can be explained with memory accesses done before the selection of a different write mode, as well as the interference of read accesses. Despite this, the overall impact of different write modes on the results of the total energy consumption and latency can be seen in these two plots.

5 Evaluation

The implementations of simulating different energy and timing modes are explained in the previous chapter. To evaluate the quality of these implementations, a series of benchmarks is used. First, the evaluation setup, as well as the different used benchmarks, are explained in Section 5.1. In the following Sections, the definition of energy and timing areas (Section 5.2), the results and statistics of these areas (Section 5.3) and the performance of the write mode interface (Section 5.4) are evaluated. In Section 5.5, possible options for the implementation in hardware are discussed.

5.1 Benchmark Setup

Generally, the setup explained in Section 3.2 is used for all evaluations, together with the adjustments and additions described in Chapter 4. All evaluations are conducted with both energy and timing values. The following five typical benchmarks are used for the evaluation:

- **bitcount** is a simple algorithm that uses a loop to count the 1 bits of a number of elements in an array. The loop uses the same memory region repeatedly.
- **dijkstra** computes the shortest path between two nodes. In this case, 10 shortest paths are computed, using global arrays for the adjacency matrix and the storing of the interim results.
- **fft** computes the fast fourier transformation for a horizon of values. The calculation is done recursively. Because the data is split into temporary arrays, a large area of the memory is used.
- **lesolve** solves a linear equation system using the gaussian elimination algorithm. The calculation is done iterative in a long running loop, therefore only a small memory region is used.
- **quicksort** sorts an array of values using the quicksort algorithm. It uses a global array in which the elements are swapped recursively.

All data used in the following evaluations is generated with the `AreaTraceWriter`. To have a reference for the evaluations, the access patterns of the described benchmarks are displayed in Figure 5.1. The read and write accesses are shown separately for a more accurate representation of the access behavior of the benchmarks. Furthermore, all data regarding the read and write access counts is displayed in logarithmic representation to be able to display small numbers of accesses alongside the large numbers. While the y-axis displays the number of accesses, the x-axis displays the physical location in ascending order, as used in the area definitions. The access patterns can be connected to the following behaviors of the benchmarks:

- **bitcount**: The main peaks of the access behavior are the counter of the loop, as well as the accesses to the array and the calculation of the number of 1s. The read access count is high, because the data of the array is only read and not written. Only the counter and the result have write accesses, which can be seen by the two write access peaks.

- **dijkstra**: By using global arrays for the adjacency matrix and interim results, most accesses in the dijkstra benchmark are generated through reading and writing array elements.

- **fft**: Because a number of temporary arrays is used by the benchmark, a wide memory region is used extensively.

- **lesolve**: Like for the **bitcount** benchmark, the main peaks for this benchmark are also caused by one long running loop and accessing an array. Additionally, the actions of swapping and subtracting lines of the guassian elimination system create the visible ascending curve of write accesses.

- **quicksort**: This benchmark also creates a visible ascending curve of write accesses and, additionally, read accesses. This curve is generated through the ascending number of comparisons and swap actions used in the quicksort algorithm.

Different area configuration files are used to test the impact of different mappings on the benchmarks. These mappings are determined by the previously described access behavior. As mentioned in Chapter 2, the retention time of values in NVMs is related to the used write energy and time. Therefore, configuring less write energy and time for an area results in less retention time for all values stored in that area. To obtain an ideal mapping of the required retention time of all values, an exact analysis would have to be executed. Although no retention faults are implemented, the assumption is made that less frequently written values should be written with higher energy and time, to simulate their higher retention time. Frequently written values can be written with less energy and time, as they are overwritten after a short time. Based on this, less than ideal mappings are created and used for the evaluation.

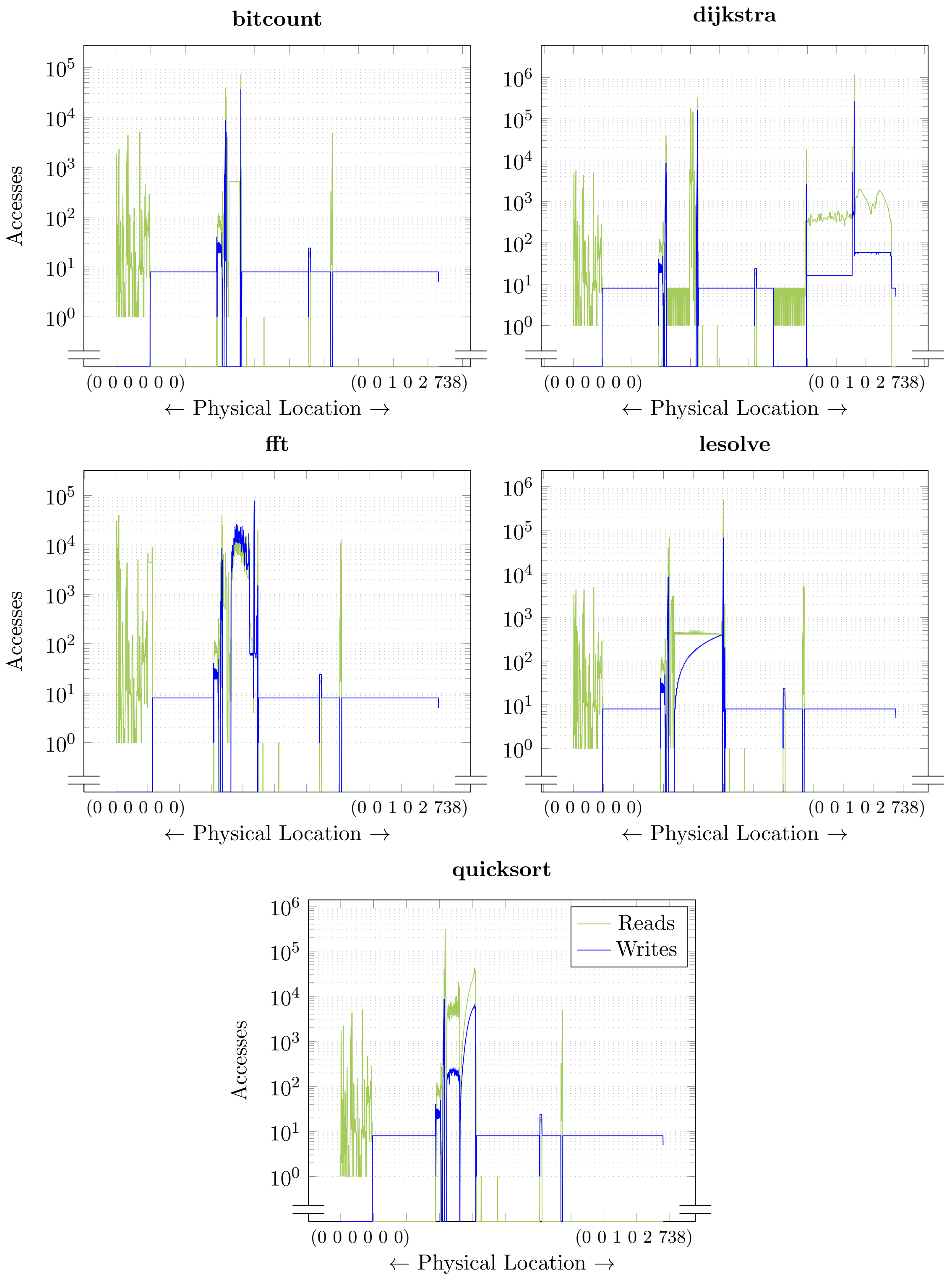


Figure 5.1: Access Patterns of Benchmarks

5.2 Energy and Timing Areas

To evaluate the implementation of different energy and timing areas, two aspects are considered separately. First, the contrast between using one write energy for every access and splitting the write energy in set and reset accesses is examined. After that, the impact of different area definitions regarding the mapping is evaluated.

Write Energy and Latency

Originally, NVMain 2.0 only supports the definition of one value for the write energy and latency. This is done due to the assumption that set and reset values are the same (see Section 1). As explained in Section 3.1, in reality these parameters can be very different from each other, depending on the technology. The difference between these two operating principles is evaluated. The benchmarks are executed once with the same values for set and reset, as well as different ones. For the case of using the same values, the respective maximum values are used. The exact used values [9] are presented in Table 5.1. Only one area for the complete memory is defined with these values, to make the difference clear.

Figure 5.2 displays the results of the energy consumption, and Figure 5.3 displays the results of the latency. The highlighted areas show that using the same value for set and reset leads to a higher energy consumption and higher latency. Although the noticeable difference is more clear for the energy consumption. By using the highest value in each case, the possible difference per value is greater for the energy consumption than the latency. Because of the assumption that all bits in a write access are written simultaneously, the difference between using the same value or different values for the latency can only be 0 ns or 70 ns. For the energy consumption, the highest possible difference is, depending on the length of the written word, up to 364.8 pJ. This number is calculated by multiplying the difference between the set and reset energy consumption (5.7 pJ) with the maximum number of bits in a column (64). Therefore, the differences between the same and different values for set and reset are more noticeable for the energy consumption than the latency. Additionally, the differences are only visible for areas in the memory that receive a high number of write accesses. Otherwise, the energy consumption and latency is mainly generated through read accesses, which are independent of the set and reset values. These results show there is a difference in the calculated energy consumption and latency between assuming the values for set and reset write accesses are the same and assuming they are different. It is therefore important to take this assumption and its impact on the simulation results into account.

	set (Energy/Time)	**reset**(Energy/Time)
set $\neq$ reset	14.03 / 120	19.73 / 50
set = reset	19.73 / 120	19.73 / 120

Table 5.1: Energy and Time Values for Equal and Unequal Set and Reset

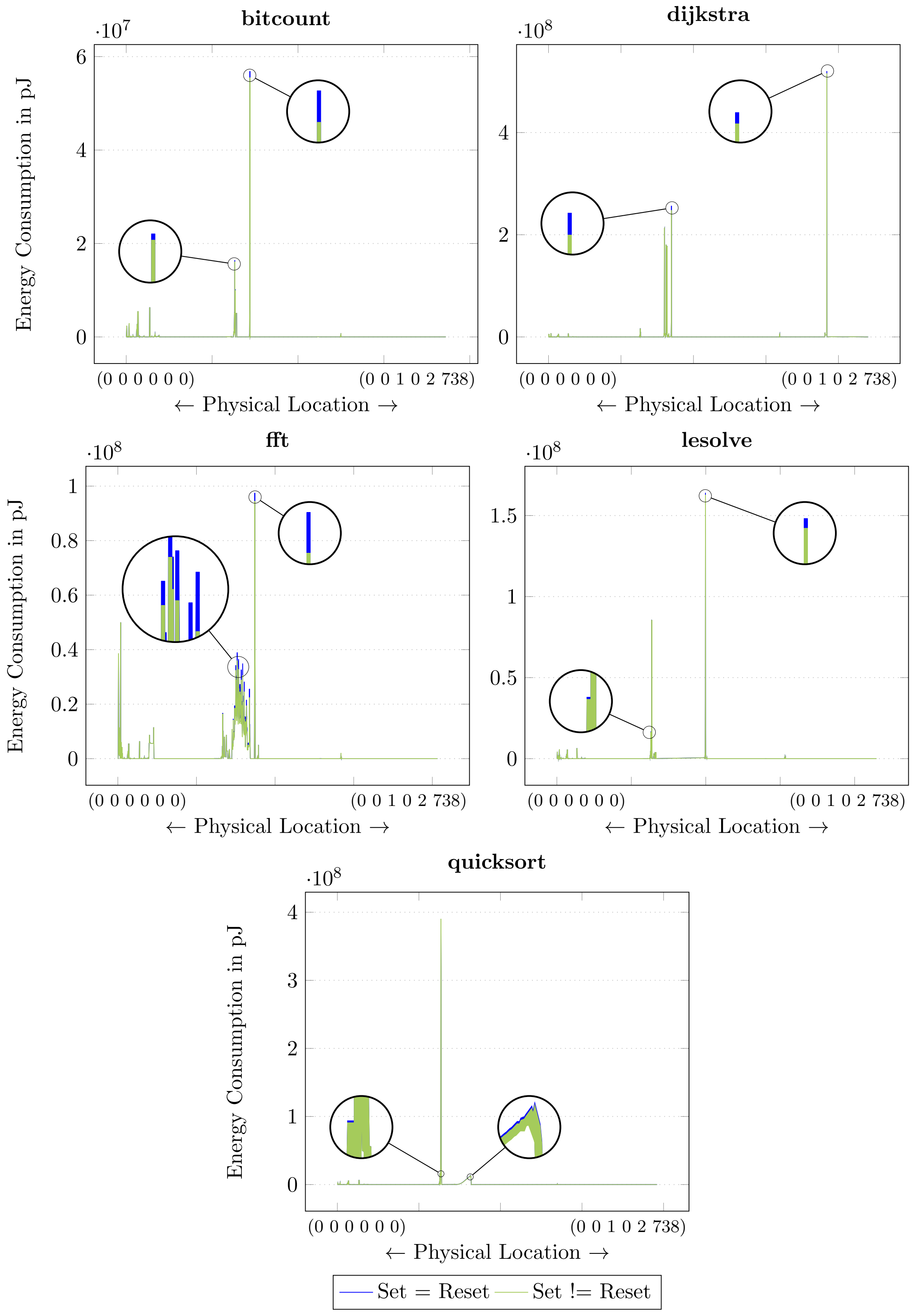


Figure 5.2: Energy Consumption of Benchmarks Depending on Set and Reset Values

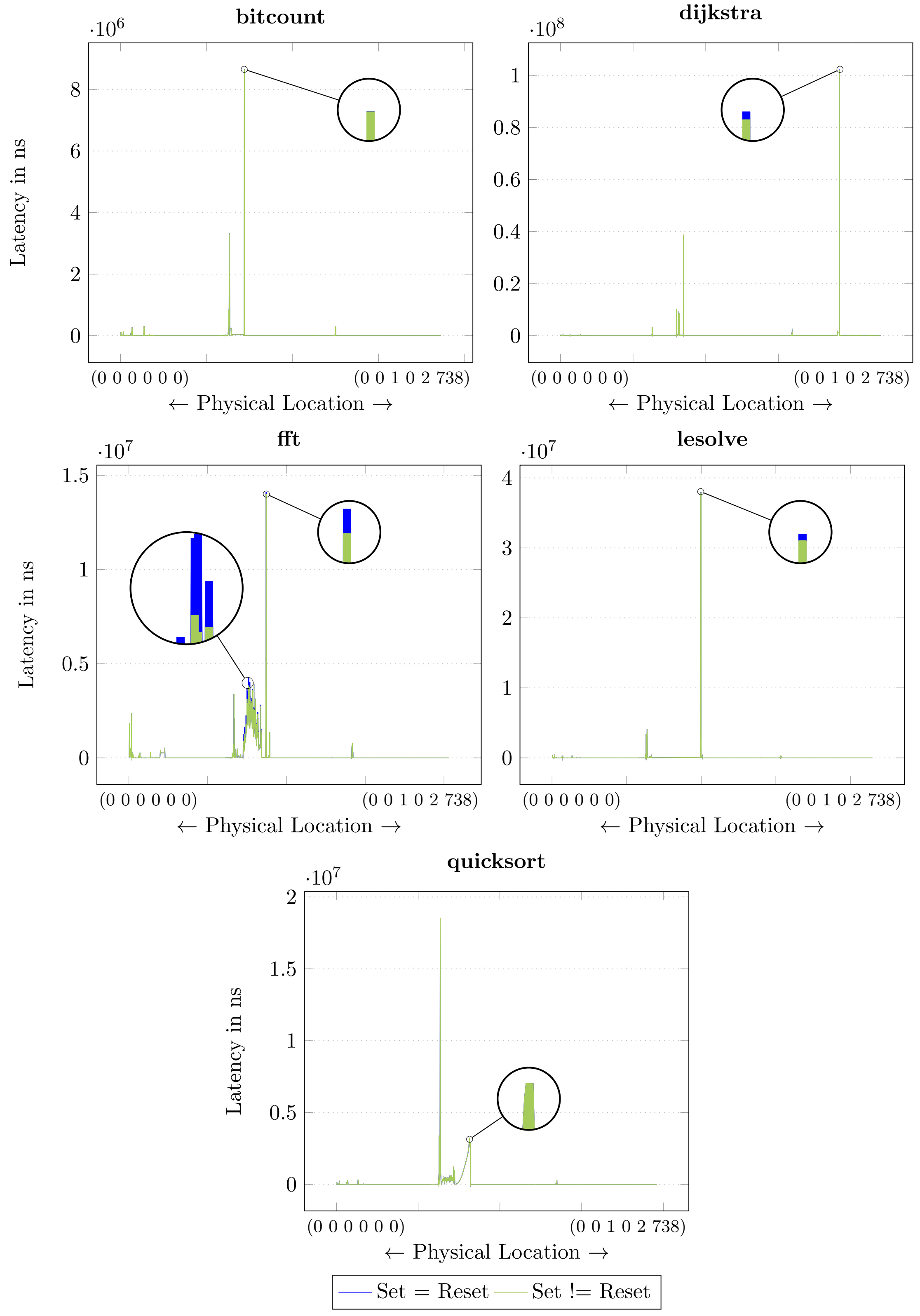


Figure 5.3: Latency of Benchmarks Depending on Set and Reset Values

Mapping

As mentioned in Section 5.1, individual mappings for the benchmarks are used for the evaluation. In Table 5.2, an example of an individual area definition for the **bitcount** benchmark is presented. In it, the two most frequently used areas for write accesses are configured to use only half the write energy and time. This factor is simply used to create a visible difference in the results and is in itself not of importance. The exact factor per area would have to be calculated, like the ideal mapping itself, based on the actual required retention time. Moreover, the read energy and time is configured with the same value for all areas, because it has to meet the minimum requirement of accessing a memory cell. In Listing 5.1, the corresponding area configuration file is shown. In reality, an individual or perfect mapping would be impossible to find out in advance - only at run-time with the right tools.

Location	**Column**	**Read** (Energy/Time)	**Set** (En./Ti.)	**Reset** (En./Ti.)
Channel 0	0	2.47 / 60	14.03 / 120	19.73 / 50
Rank 0	↓	↓	↓	↓
Bank 0	756	2.47 / 60	7.01 / 60	9.87 / 25
Sub-Array 0	↓	↓	↓	↓
Row 1	768	2.47 / 60	14.03 / 120	19.73 / 50
Channel 0	↓	↓	↓	↓
Rank 0	252	2.47 / 60	7.01 / 60	9.87 / 25
Bank 0	↓	↓	↓	↓
Sub-Array 0	256	2.47 / 60	14.03 / 120	19.73 / 50
Row 3	↓	↓	↓	↓

Table 5.2: Definition of Memory Areas for Individual Mapping of Bitcount

```
0 0 0 0 0 0 2.47 60.0 14.03 120.0 19.73 50.0
0 0 0 0 1 756 2.47 60.0 7.01 60.0 9.87 25.0
0 0 0 0 1 768 2.47 60.0 14.03 120.0 19.73 50.0
0 0 0 0 3 252 2.47 60.0 7.01 60.0 9.87 25.0
0 0 0 0 3 256 2.47 60.0 14.03 120.0 19.73 50.0
```

Listing 5.1: Section of the Bitcount Area Configuration File

The individual mappings for the other benchmarks are created similarly and then used to simulate their corresponding benchmarks. In Figure 5.4, the resulting impact of the different configuration files on the energy consumption is compared. Figure 5.5 displays the resulting impact on the latency of the benchmarks. All benchmarks show the same energy consumption and latency in the areas that were not changed with the individual mapping. The only difference occurs in the areas that were configured to smaller values, indicated by the blue color. The highlighted areas show additional smaller differences that would be otherwise hard to see. Noticeable is the size of the difference between using no mapping and using an individual mapping, depending on the write count of the benchmarks in the configured areas. For example, the **fft** benchmark generates a higher write access count in its adjusted areas than the **quicksort** benchmark. Therefore, the impact of the mapping is more noticeable in **fft** than in **quicksort**, which is also transferable to the other benchmarks.

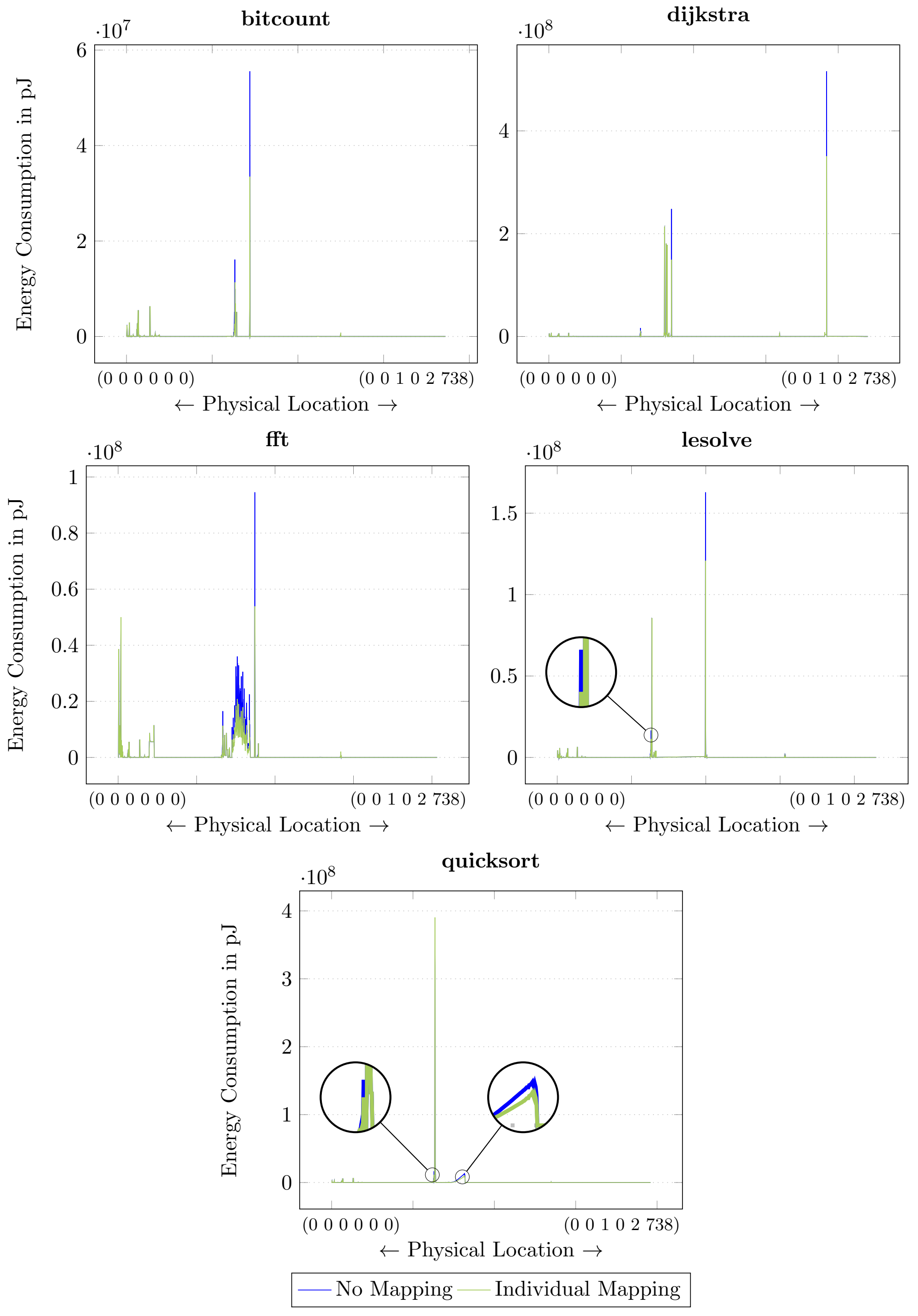


Figure 5.4: Energy Consumption of Benchmarks with Individual Mapping

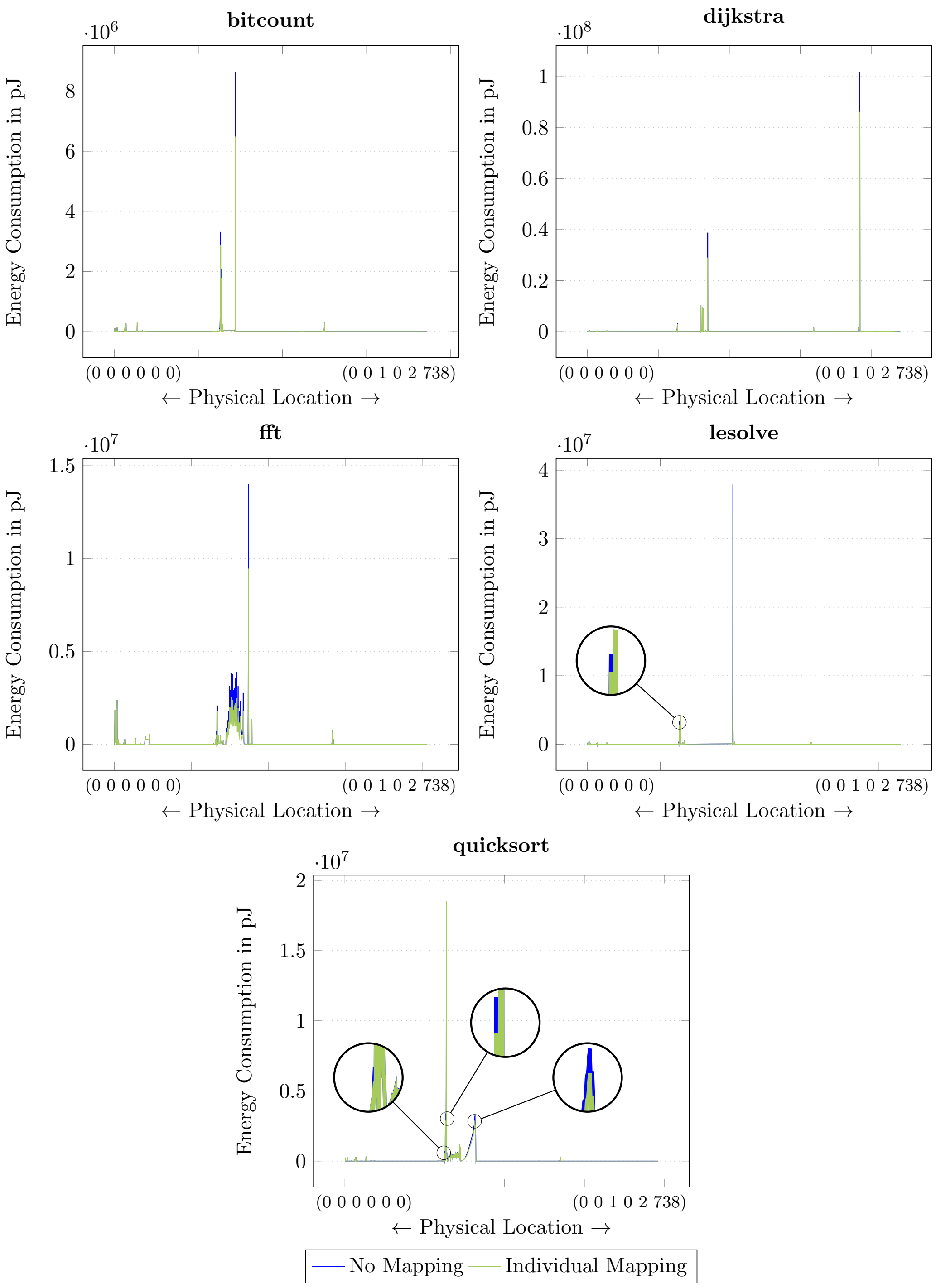


Figure 5.5: Latency of Benchmarks with Individual Mapping

The used individual mappings are, as already explained, not perfect mappings. If perfect mappings are created and used with their corresponding benchmarks, the differences to using no mapping would be even more noticeable. Not only in the already displayed areas, but over the whole memory. As shown, using different area configurations to create mappings can be used to lower the energy consumption and latency in areas based on their usage.

5.3 Trace Writer

To evaluate the new trace writers, the total read and write accesses of the five benchmarks are set in relation to the total energy consumption and latency. Evaluating the individual tracelines is not sensible because the plotted data would be the same as the data of the statistics trace. Furthermore, the statistics of the individual memory locations are already used to generate all data for this chapter. The total values of the five benchmarks without a mapping are displayed in Figure 5.6, alongside the total values of the individual mappings.

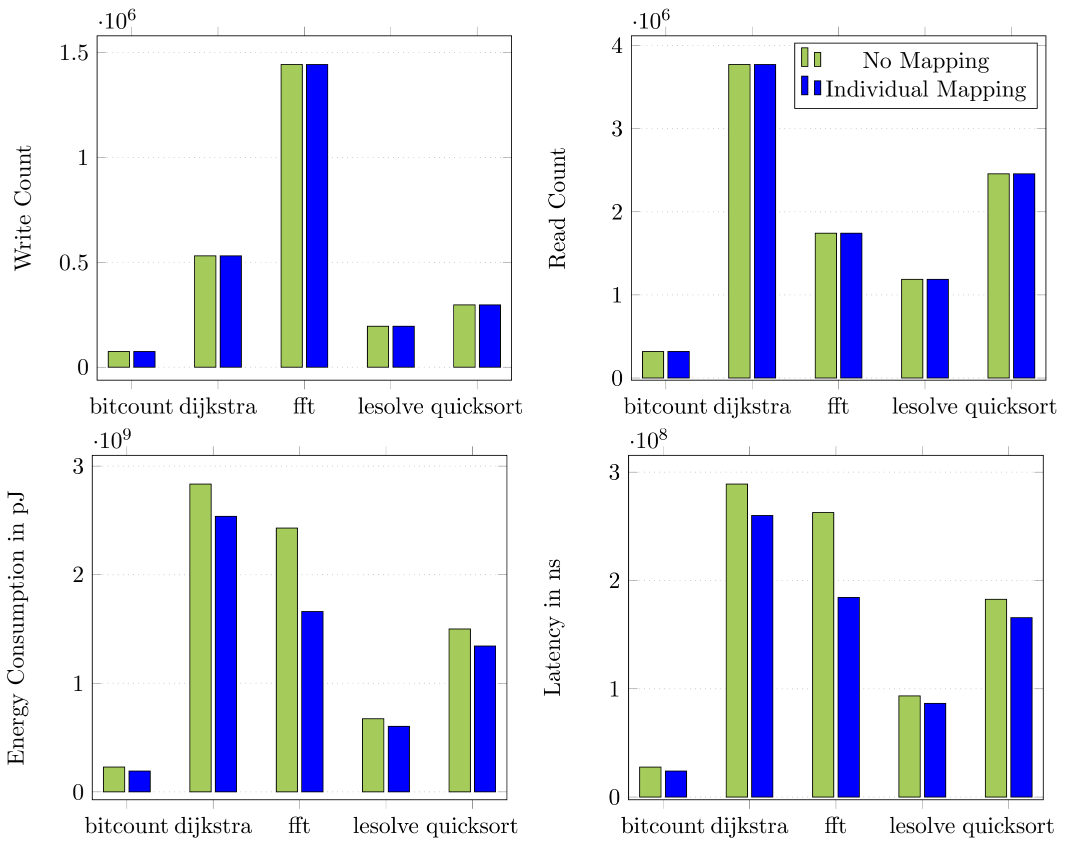


Figure 5.6: Total Energy Values of Benchmarks

The access patterns and program behavior presented in Section 5.1 are used to explain the results of the benchmark, regardless of the used mapping:

- **bitcount**: In comparison to the other benchmarks, **bitcount** has the least memory accesses. Therefore, the total energy consumption and latency are also the lowest of the benchmarks.

- **dijkstra**: Although **dijkstra** does not have the highest write access count and does not access a wide range of memory, it has the highest overall energy consumption and latency. The reason for this is the extremely high read access count. Even if read accesses consume less energy and time than a write access, **dijkstra** has approximately double the read accesses than the **fft** benchmark. And while **fft** has more write accesses than **dijkstra**, the factor is lower.

- **fft**: This benchmark generates the highest write count of all benchmarks through the usage of temporary arrays. Therefore, its energy and time consumption is also high, with write accesses consuming more energy and time than read accesses.

- **lesolve**: With the second-lowest write and read access count, **lesolve** thus has also the second lowest energy and time consumption. Although the benchmark has a lot of array accesses, the total access count is much lower than through using a number of temporary arrays like for **fft**.

- **quicksort**: Through the many comparisons needed in a sorting algorithm, **quicksort** generates a lot of read accesses. Additionally, the swap actions generate write accesses that further influence the energy and time consumption. This results in **quicksort** having the third-highest energy and time consumption.

The impact of the individual mappings, described in Section 5.2, are also reflected in the total values. Comparing the write and read count shows the same results, because the benchmarks were executed under the same circumstances except the mappings. But focusing on the energy consumption and latency, the differences are clear. As already mentioned, the more write accesses a benchmark has, the greater the impact of the individual mapping is.

5.4 Write Mode Interface

Changing the write mode for benchmarks during their run-time only has a noticeable influence on the performance if the right areas are modified. This is already shown by the individual mappings in Section 5.2. In a system used to run multiple applications with different access patterns, it is not suitable to define the area and timing areas at the start of the system for all applications. Instead, the areas have to be adjusted during run-time. To evaluate the write mode interface, the following evaluations are done for all benchmarks. First, it is evaluated if the usage of the interface changes the access pattern of the benchmarks, through generating more read and write accesses. For this, the benchmark is executed once without using the interface and once with using the interface, in both cases defining the same values for all areas. Next, the interface is used to define the same individual mappings that were used in Section 5.2. This evaluates the difference between defining the areas at the start of the simulation or during run-time. Lastly, the impact of four different write modes is evaluated.

Impact of the Write Mode Interface on Memory Accesses

In Figure 5.7 and 5.8 the results of the read and write accesses are displayed for defining the same energy and timing areas with and without an interface. As already stated, the benchmarks are executed once without the usage of the interface and once with the usage of the interface. In the second case, the interface is simply used to update the areas to the same values as they were before. This method is used to detect if the usage of the interface generates a not negligible amount of memory accesses. Because the parameters for the energy and time consumption are configured to the same values, only the read and write accesses are considered. They are again displayed on a logarithmic scale to be able to detect small changes. The plots show there is only a slight difference, despite the added accesses of the interface. The only noticeable difference is on the used memory areas, as the interface partially shifts the areas the application uses. Therefore, the interface does not have a significant impact on the performance of the simulated system.

Changing Area Definitions with the Write Mode Interface

Using an individual mapping as described in Section 5.2 means the areas are defined before the application is called. Therefore, all accesses to the specific areas used by the application have the same values for energy and time consumption. Regardless of originating from the simulated application or not. By using the interface to define the individual mappings during run-time, only the accesses originating from the application use the values from the individual mappings. Therefore, a difference should be seen in areas that are used by both, application and system. To evaluate the differences, two cases are simulated. In one case, the individual mappings are defined statically at the start of the simulation and the application is executed without the use of the interface. In the other case, the individual mapping is defined at run-time through the interface. Figure 5.9 and 5.10 display energy consumption and latency of both cases. The interface partially shifting the areas the application uses can be seen again. Apart from that, a difference in one specific area is noticeable, caused by accesses that occur before the usage of the interface. For the energy consumption of the **bitcount** and **fft** benchmark, the difference can simply be seen by the color difference. For the other benchmarks, the area is highlighted to be able to see the difference more clearly. This memory area is therefore accessed by the system before the calling of the application, but also by the application itself. Hence, to use an individual mapping of an application beneficially, it is sensible to configure it through an interface during run-time. Otherwise, values of the system could be lost due to the individual mapping being only suited for the specific application.

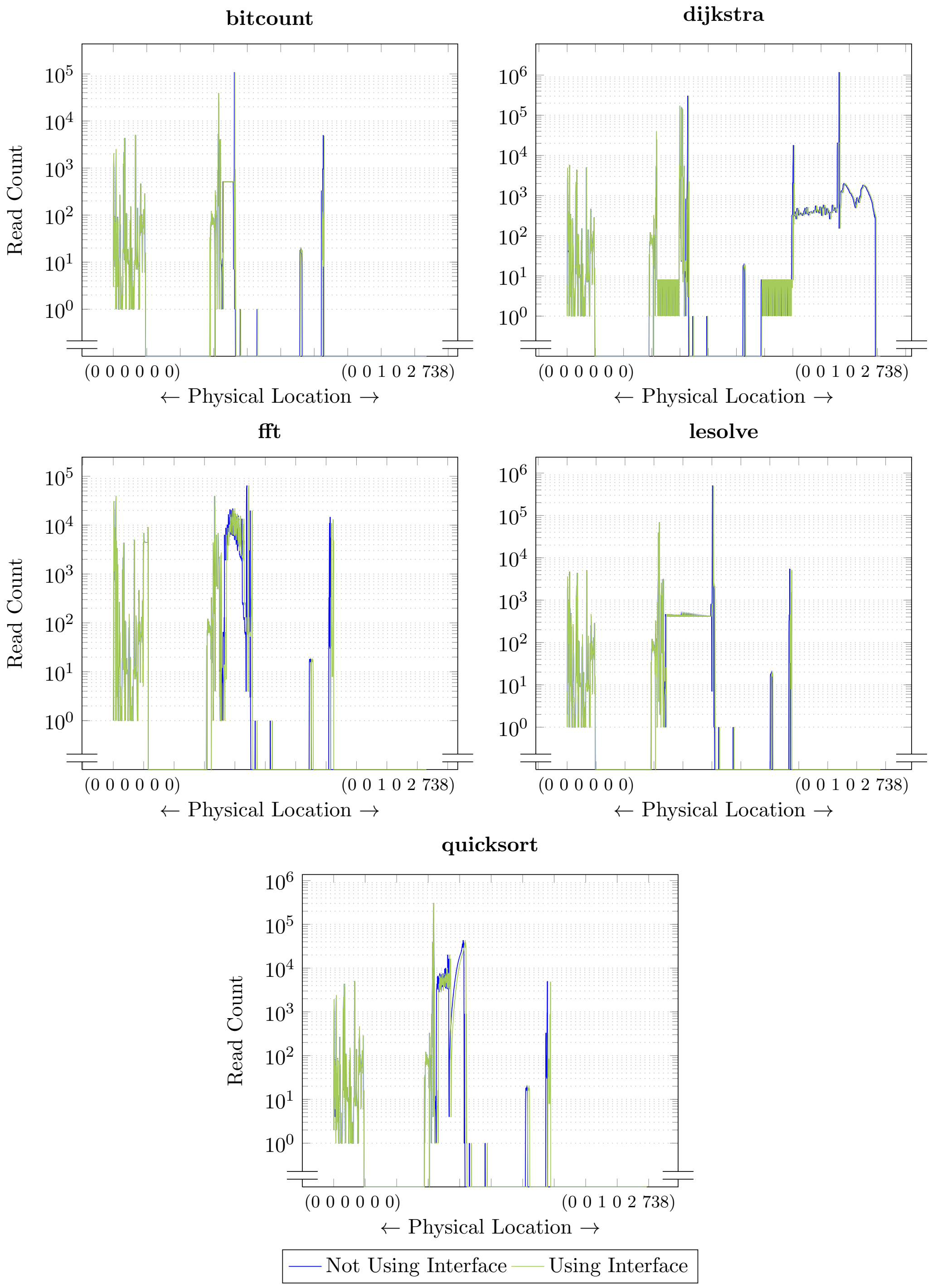


Figure 5.7: Impact of the Write Mode Interface on Read Accesses

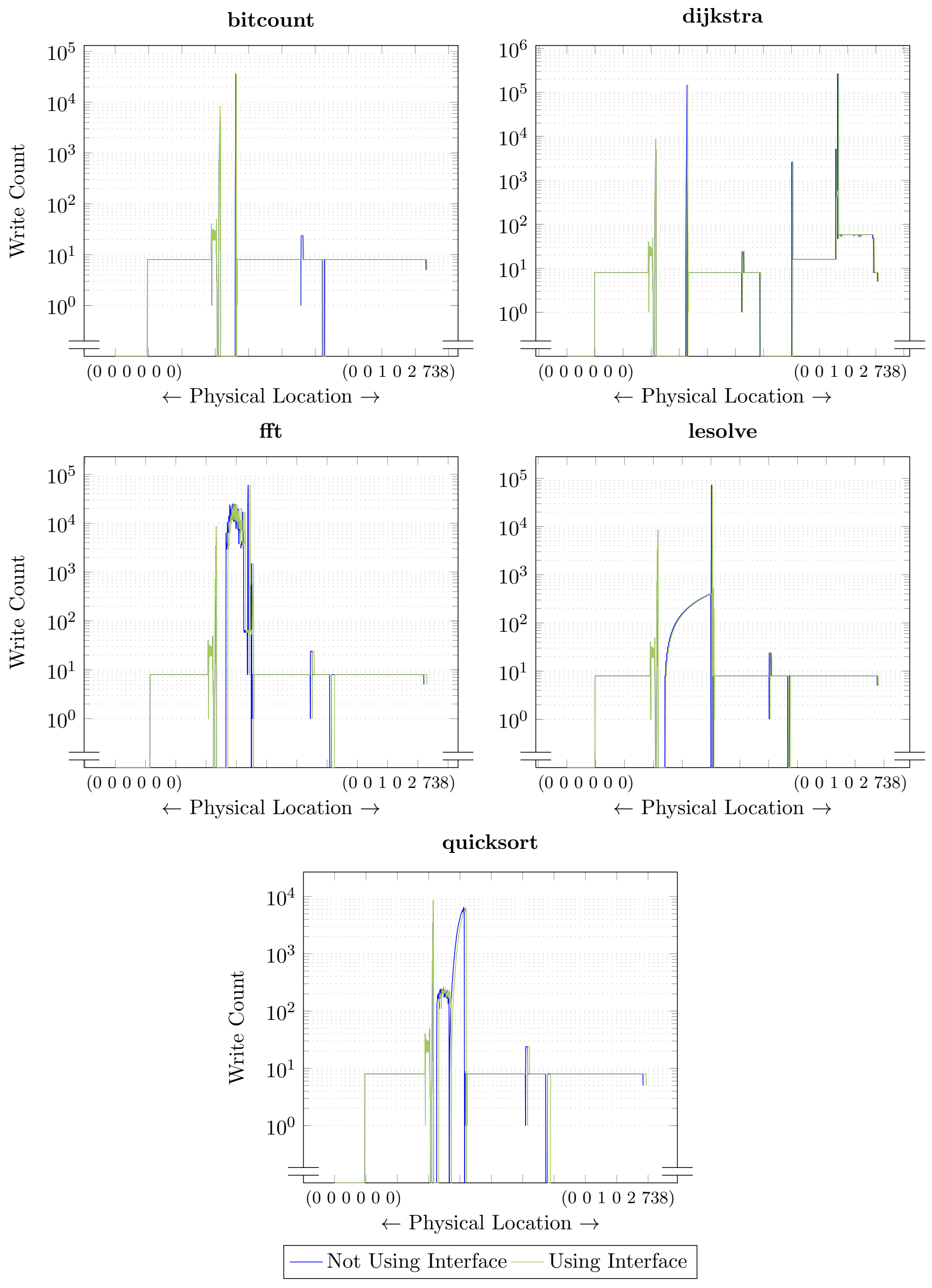


Figure 5.8: Impact of the Write Mode Interface on Write Accesses

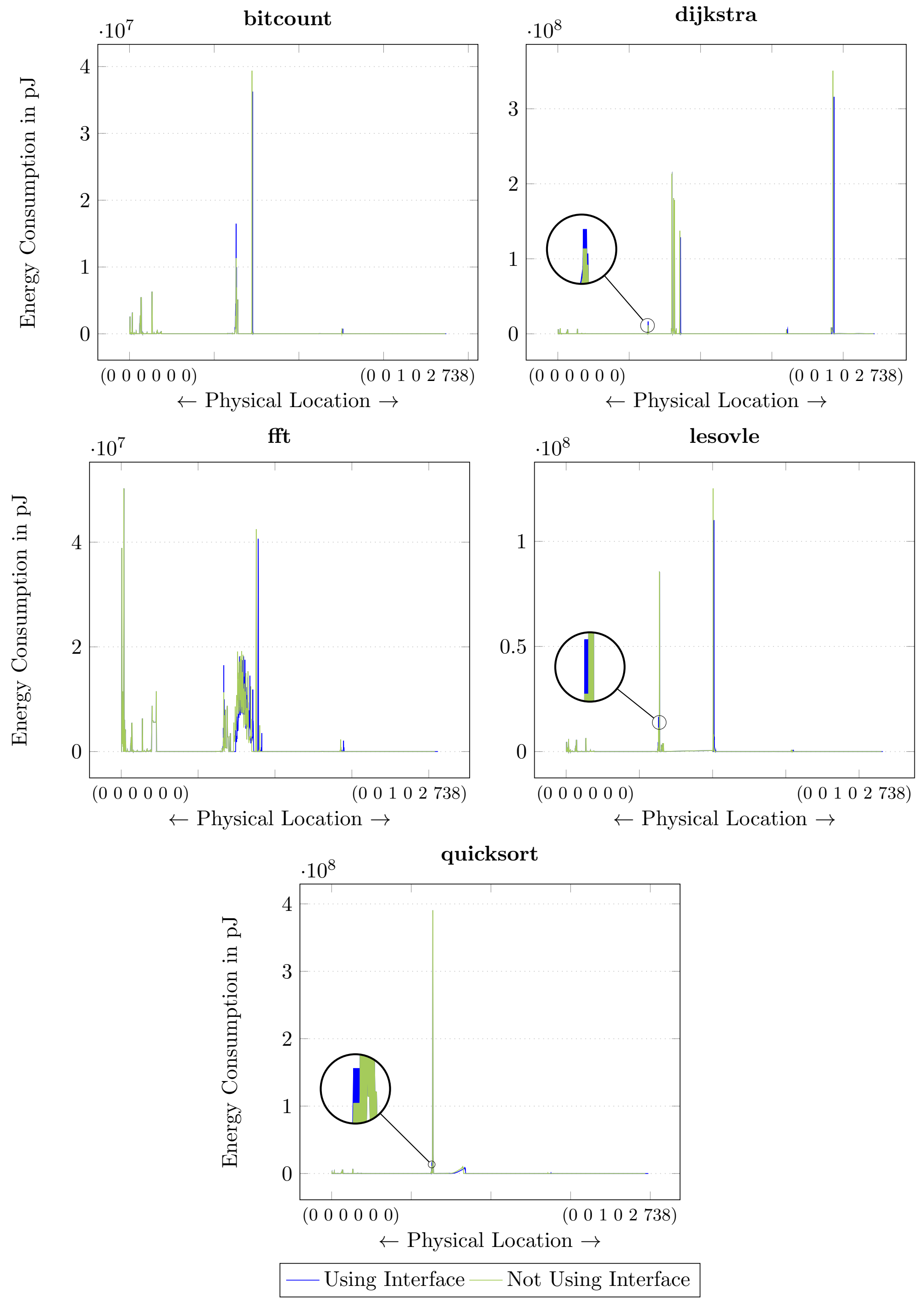


Figure 5.9: Energy Consumption: Creating Individual Mapping with Configuration File Compared to Write Mode Interface

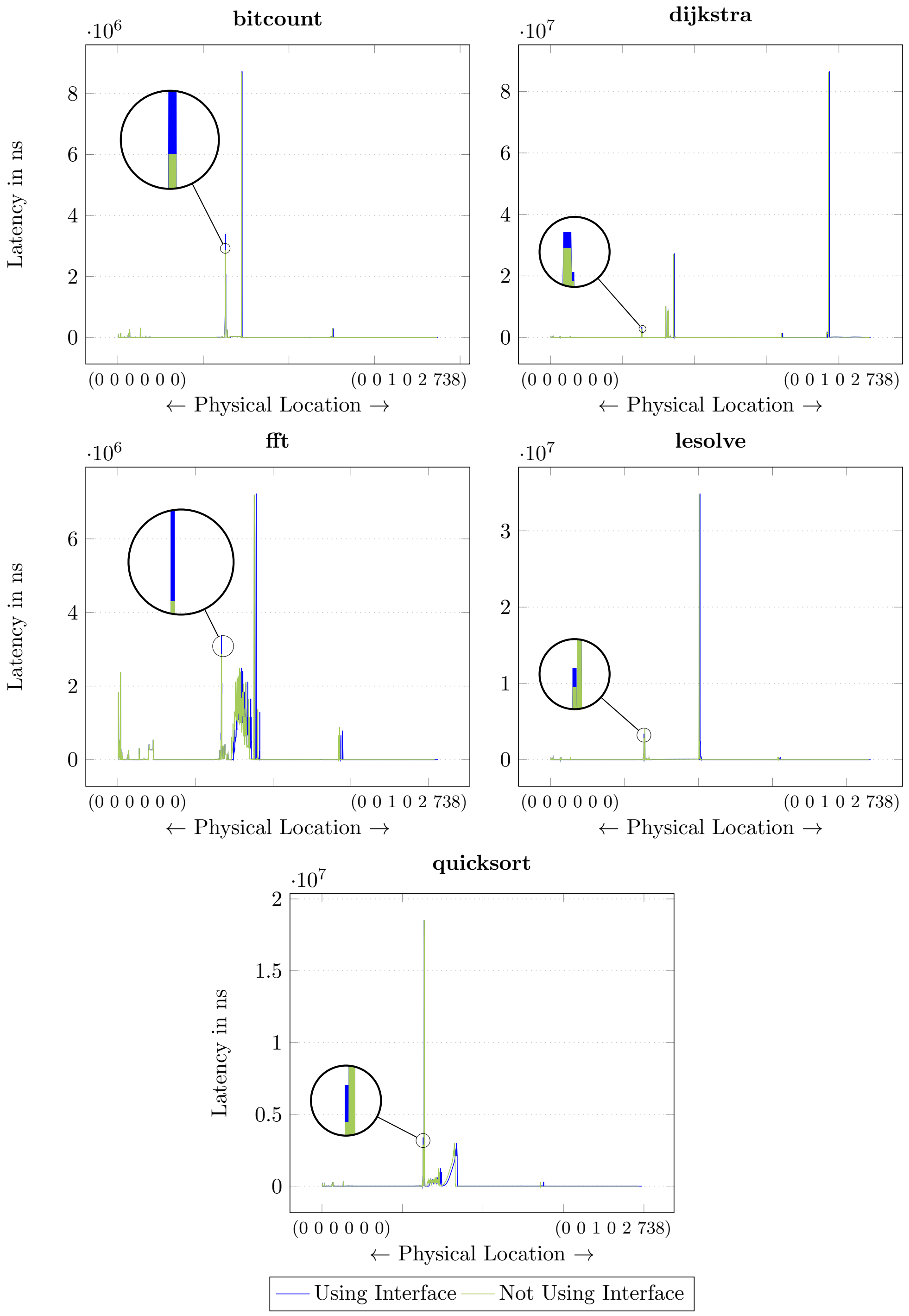


Figure 5.10: Latency: Creating Individual Mapping with Configuration File Compared to Write Mode Interface

Using Different Write Modes

The interface can be used to define and configure multiple write modes. To evaluate this feature, four different write modes are used. The values of the modes are the same as in Section 4.3.3 and are displayed in Table 5.3. Write mode 1 therefore represents a

”standard” write mode, mode 0 a slower and more energy consuming mode and mode 2 and 3 being faster and less energy consuming write modes.

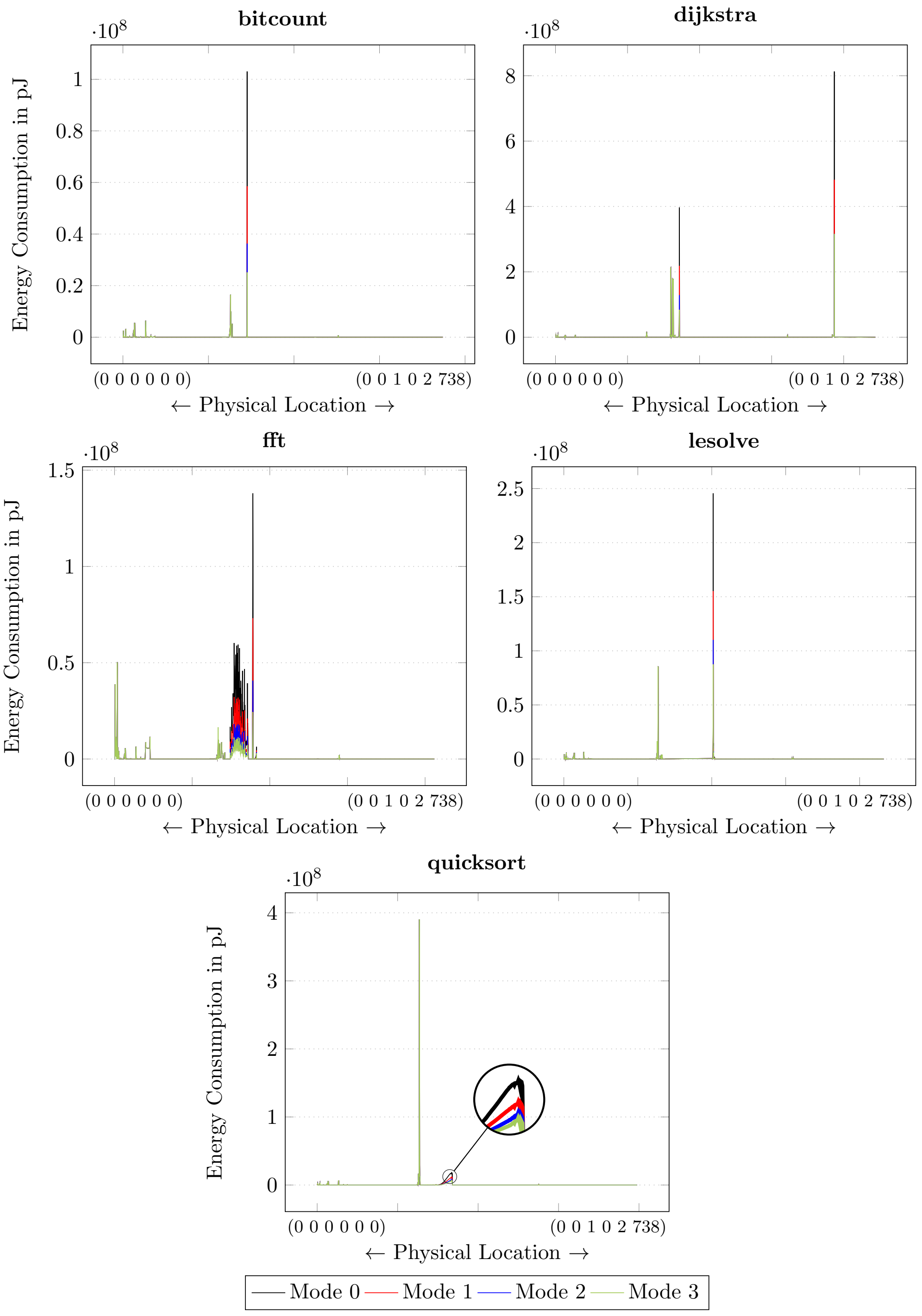


Figure 5.11: Energy Consumption: Comparison of Different Write Modes

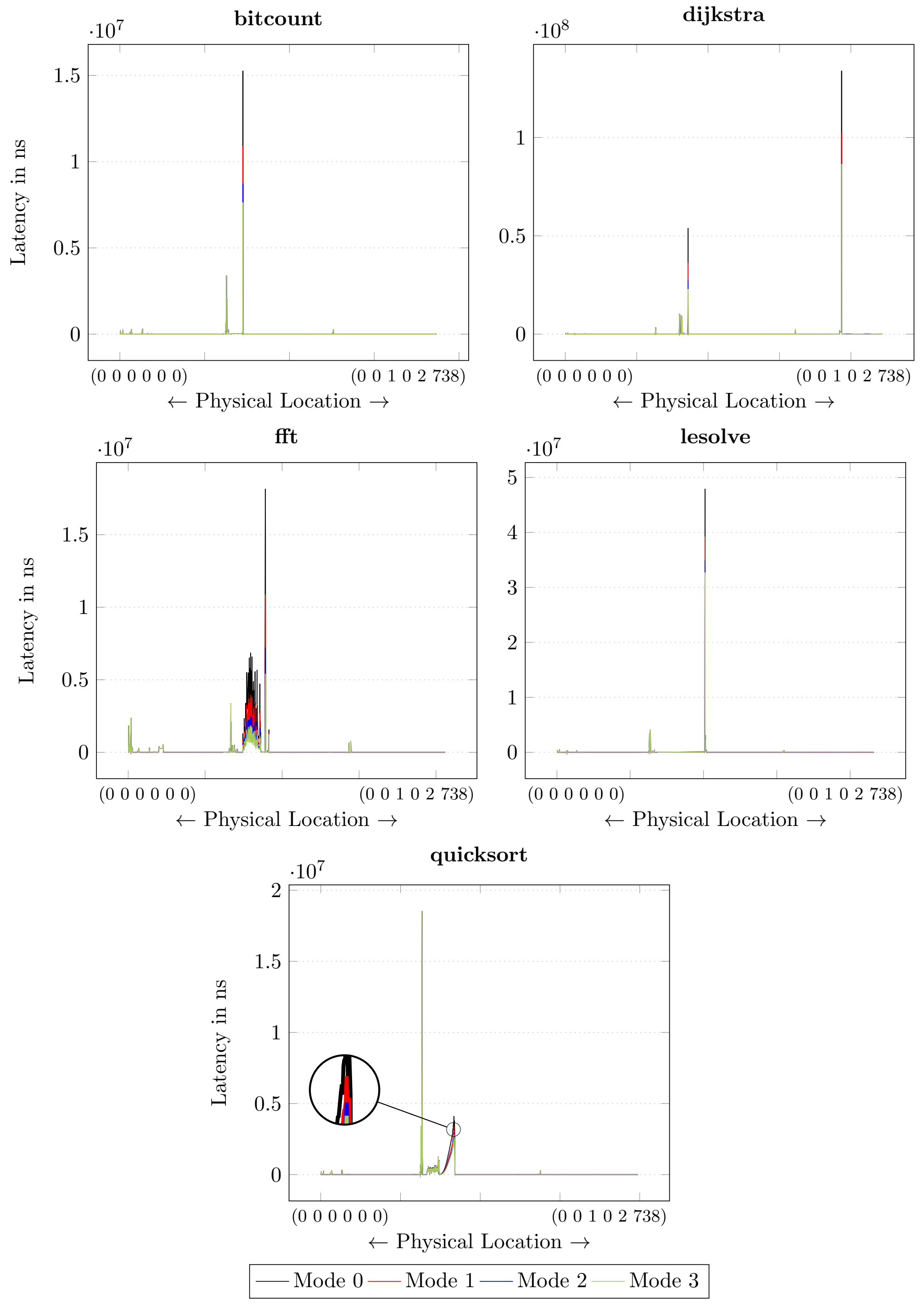


Figure 5.12: Latency: Comparison of Different Write Modes

	Mode 0	**Mode 1**	**Mode 2**	**Mode 3**
Read (Energy/Time)	2 / 60	2 / 60	2 / 60	2 / 60
Set (Energy/Time)	28.2 / 241	14.1 / 120.5	7.05 / 60.25	3.525 / 30.125
Reset (Energy/Time)	39.6 / 101	19.8 / 50.5	9.9 / 25.25	4.95 / 12.625

Table 5.3: Write Modes

All benchmarks are executed with each write mode, which is used for the individual mappings. The resulting energy and time consumption is displayed in Figure 5.11 and 5.12. In all plots except for the **quicksort** benchmark, the differences can be simply seen by the color differences. For the **quicksort** benchmark, the area is highlighted to make the difference visible. Because the write modes are only changed for the specific areas of the individual mappings, the effect of the write modes is only noticeable in these areas. Additionally, the differences between the write modes are only significantly different at the memory locations that receive a large number of write accesses. Because in other memory locations, the differences are very small. Therefore, the benchmarks with a high write count, for example **fft**, show a higher impact than benchmarks with a low write count, for example **quicksort**. Also noticeable is the results not showing the same ratios as the values the simulations were executed with. For example, the defined values of write mode 2 are half the defined values of write mode 1. But in the results, it can be seen that the energy consumption and latency of write mode 2 are not half of the results of write mode 1. As already explained in Section 4.3.3, this is caused by write accesses that are executed before the change of the write mode, as well as by read accesses that are not influenced by the change of write modes.

5.5 Implementation in Hardware

To use multiple area definitions with different configurable write modes outside the simulator, the implementation presented in this thesis has to be converted to real hardware. Possible requirements for this are presented in this section, regarding for example the used storage for the area definitions.

Defining multiple memory areas first requires the storing of the areas with their size or boundaries. In this thesis, the areas are stored by simply using the memory location as the starting point. This would be not reasonable in hardware, because it would require too much storage for the required parameters. Instead, the areas could be stored in a fixed sequence, with only the write mode itself needing to be stored. The area the write mode corresponds to is then encoded by the address it is stored at. The write mode should only be stored in its encoded form to save storage. To support four write modes, each area therefore only has to store two bits.

The amount of areas needing to be stored is composed of multiple parameters. Namely, the number of different components of the memory layout and size of the defined areas. A method for approximating the number of needed areas is shown in equation 5.1.

$$\frac{\#\mathsf{Channels} * \#\mathsf{Ranks} * \#\mathsf{Banks} * \#\mathsf{Array} * \#\mathsf{Rows} * \#\mathsf{Columns} * \#\mathsf{Bits\ in\ Column}}{\#\mathsf{Bits\ per\ Area}} = \#\mathsf{Areas} \tag{5.1}$$

The used number of each memory component in the system, for which areas should be defined, is denoted as "#Component". These numbers are multiplied with each other and the number of bits per column, to calculate the complete memory size in bits. This number is divided by the granularity of the areas, i.e. the number of bits an area should be defined for. The result of this calculation is the number of areas that need to be stored to define energy and timing areas with these constraints. To increase or decrease the number of stored areas, the easiest way of adjusting the number is to raise or reduce the granularity of areas. For example, if every defined area applies to 1 KB each, the number of areas is four times as big, as if every defined area applies to 4 KB each. Therefore, the number of areas to store have to be carefully weighed up against the minimal required size for each area. If the areas are defined too big, there is risk to lose information at run-time in the system, because the configured write mode may not be ideal for the whole area.

5.6 Discussion

Summing up, the key additions to the simulator are the definition of different energy and timing mode areas, multiple ways of analyzing the results and an interface to change the write mode during run-time. The goal of defining different energy and timing areas is not only to separate the set and reset parameters, but also to be able to define areas smaller than the memory components. If the simulation assumes a write access always has the same value, not separated in set and reset accesses, the value would have to be chosen as the higher one of both. This leads to a higher computed energy and time consumption than it would be in reality. The evaluation shows this difference between separated set and reset values and using one value, both for the energy consumption and the latency. The results for separated set and reset values are noticeably lower. The extension of the simulator for this feature is therefore proven sensible.
By defining other write modes for specific, usually highly used areas, the retention times of accesses in this area can be lowered. If configured right, they are still sufficient for the application to work. The definition of areas can therefore be used to define mappings that reduce the energy and time consumption of applications. This is verified by the evaluation. Individual mappings are created to configure highly used areas to a different write mode. The assumption is made that the retention time of these accesses can be lower than in the rest of the areas. The results of the evaluation for the individual memory columns show a lower energy consumption and latency in the areas configured to a different write mode. This demonstrates the effectiveness of the method and the functionality of the definition of areas.
The same impact can be seen when the total values, write and read count, as well as energy and time consumption, are considered. They are evaluated by setting the results of the total values in relation to the access behavior of the benchmarks. All benchmarks show a

reduction in energy and time consumption, if executed with the individual mappings, in comparison to using no individual mapping. This again demonstrates the effectiveness of using individual mappings. The added trace writers are proven to be able to analyze the overall behavior of simulations with different defined memory areas.

Defining different energy and time areas at the start of the simulation is of little use if the simulated applications have a dynamic access behavior. They require different write modes for the same memory locations at different times. The write mode interface is implemented for this use case, being able to adjust the write mode of the areas at run-time. The results of the evaluation show the usage of the interface during a simulation does not have a major impact on the energy and time consumption. This makes the interface feasible to use, because otherwise the advantages of using the interface would have to be weighed up against the added energy and time consumption.

By comparing the results of the individual mappings defined at the start of the simulation to the definition using the interface, it is first shown that the interface is able to define the same individual mappings as the configuration files. Furthermore, the evaluation shows a part of the areas, configured to a different write mode, are also used before the application is executed. If the write mode is defined statically at the start of the simulation, it cannot be guaranteed that the configured write mode for these areas are sufficient to retain all information before the start of the application. Because of this, the usage of the write mode interface is preferable to a static definition of energy and timing areas. Not only is the interface able to change the write mode of areas at run-time, but it also supports the definition of four different write modes. Using this feature enables a more precise adjustment for the individual mappings, as shown in the evaluation.

Considering the implemented changes in the simulator, it is clearly not possible to just copy them to real hardware. Therefore, it is suggested to store only the currently configured write mode in a fixed sequence. This avoids the storing of each individual memory location. To save storage space or fulfill maximum storage space requirements, it is also recommended to weigh up the minimal required size of each defined area against the total number of areas having to be stored.

6 Conclusion

The main purpose of this thesis is to extend the introduced simulation environment. This extension first includes the definition of different energy and timing areas. Also, new trace writers for output and analysis, as well as a write mode interface, are added. All additions and evaluations are done in the presented simulation environment. To conclude this work, the presented additions are first summarized and later compared to some of the related work. At the end, future additions are proposed, which could be realized on the basis of this work.

The feature of defining memory areas with different energy and timing properties is realized in separate controllers to enable further modifications. Defined memory areas are not limited to the boundaries of the individual memory components, but instead are independent of them. Energy and timing properties are separated in different controllers, allowing the simulation of only one of them. The definition of the areas takes place in configuration files. For each area, the starting point is defined by its physical location in the memory as well as its properties regarding read, set and reset energy or time. During the simulation, the controllers calculate the corresponding energy and time consumption for each memory request based on its individual properties. The controllers are integrated into the simulator and are tested and evaluated. The evaluation shows the impact of separating set and reset operations, as well as using area definitions to create individual mappings. Both can be used to optimize the energy consumption of the simulated system.

Trace writers are one of the main output sources for the simulator. New trace writers are therefore created to serve different purposes for the simulation output of memory areas. One trace writer focuses on displaying information for each individual memory request with its energy and timing properties. Another trace writer calculates the total values for memory accesses and their energy and time consumption for each used memory column. Also, the total of these values for the whole simulation is calculated. Additionally, a trace writer is implemented that combines both features of the described trace writers. It outputs the individual memory accesses as well as the calculated statistics. Because the simulator already supports trace writers, the new ones are integrated and tested similarly. The evaluation shows they can be used to effectively analyze simulations in combination with the new feature. Also, the impact of the access behavior on the total energy and time consumption of the simulation is presented.

The interface to configure the write mode during runtime builds upon the feature of defining different memory areas. Through the interface, the application can trigger a change of the write mode during the simulation. For this, a bit word is used in the application to carry all relevant information for the change to the corresponding controllers. Four modifiable write modes are implemented that can be freely used for every defined memory area. During the simulation, the writing of the bit word results in a change in the stored definition of the specific memory area. From this point on, the new definition gets used until further changes happen. The interface is integrated into the simulator by adjusting the already described implementation of memory areas. Testing the interface shows it working correctly in combination with the other added features. Evaluating the interface shows it has a negligible impact on the overall access behavior of the simulation. Additionally, the interface can be used successfully to not only create individual mappings, but also to use the different configured write modes for them.

Comparing the extended simulator from this thesis to the simulators [6, 2, 15, 18] presented in Chapter 2, the differences are clear. The presented simulators mostly focus on simulating hybrid systems or the general different features of NVMs. None of them focus in particular on the energy properties or simulating different write modes. Of the presented write modes [16, 21, 28, 10, 3], the CDDW approach of Li et al. is the one most similar to this simulator extension. By using the added interface, the write modes of areas can be adjusted to the proposed slow and fast modes. A dual-write scheme can therefore be supported. Only the worst case lifetime analysis has to be done separately. The proposed Flip-N-Write mode, as well as storing more than one bit per memory cell, is to some extent already implemented in the simulator but is currently not supported by the new extensions.

For future work, the resulting simulation environment from this thesis can be further extended. On the one hand, this can be done by adding more features to support a larger variety of NVM properties. This can include the integration of more of the presented write modes. Another possible addition is the accurate simulation of retention times coupled with error injection. Meaning if the write mode of an area is badly chosen, the stored values can be lost due to an insufficient retention time. On the other hand, added features can be improved. A possible example of this is realizing the interface and memory area definitions more closely as they would be implemented in real hardware. Aspects to take into consideration are already discussed in this thesis.

Bibliography

[1] Arm architecture reference manual armv8, for armv8-a architecture profile. https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile.

[2] K. Asifuzzaman, R. S. Verdejo, and P. Radojković. Enabling a Reliable STT-MRAM Main Memory Simulation. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '17, page 283–292, New York, NY, USA, 2017. Association for Computing Machinery.

[3] F. Bedeschi, R. Fackenthal, C. Resta, E. M. Donze, M. Jagasivamani, E. C. Buda, F. Pellizzer, D. W. Chow, A. Cabrini, G. M. A. Calvi, R. Faravelli, A. Fantini, G. Torelli, D. Mills, R. Gastaldi, and G. Casagrande. A bipolar-selected phase change memory featuring multi-level cell storage. *IEEE Journal of Solid-State Circuits*, 44(1):217–227, Jan 2009.

[4] S. Bhatti, R. Sbiaa, A. Hirohata, H. Ohno, S. Fukami, and S. Piramanayagam. Spintronics based random access memory: a review. *Materials Today*, 20(9):530–548, 2017.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011.

[6] S. Bock, B. R. Childers, R. Melhem, and D. Mosse. Hmmsim: a simulator for hardware-software co-design of hybrid main memory. In *2015 IEEE Non-Volatile Memory System and Applications Symposium (NVMSA)*, pages 1–6, Aug 2015.

[7] J. Boukhobza, S. Rubini, R. Chen, and Z. Shao. Emerging nvm: A survey on architectural integration and research challenges. *ACM Trans. Des. Autom. Electron. Syst.*, 23(2), nov 2017.

[8] A. Chen. Emerging nonvolatile memory (nvm) technologies. In *2015 45th European Solid State Device Research Conference (ESSDERC)*, pages 109–113, Sep. 2015.

[9] J. Chen, R. C. Chiang, H. H. Huang, and G. Venkataramani. Energy-aware writes to non-volatile main memory. *SIGOPS Oper. Syst. Rev.*, 45(3):48–52, jan 2012.

[10] S. Cho and H. Lee. Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 347–357, Dec 2009.

[11] B. Jacob, S. W. Ng, and D. T. Wang. Chapter 10 - dram memory system organization. In B. Jacob, S. W. Ng, and D. T. Wang, editors, *Memory Systems*, pages 409–424. Morgan Kaufmann, San Francisco, 2008.

[12] B. Jacob, S. W. Ng, and D. T. Wang. Chapter 7 - overview of drams. In B. Jacob, S. W. Ng, and D. T. Wang, editors, *Memory Systems*, pages 315–351. Morgan Kaufmann, San Francisco, 2008.

[13] B. Jacob, S. W. Ng, and D. T. Wang. Chapter 8 - dram device organization: Basic circuits and architecture. In B. Jacob, S. W. Ng, and D. T. Wang, editors, *Memory Systems*, pages 353–376. Morgan Kaufmann, San Francisco, 2008.

[14] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267, April 2013.

[15] T. Lee and S. Yoo. An fpga-based platform for non volatile memory emulation. In *2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–4, 2017.

[16] Q. Li, L. Jiang, Y. Zhang, Y. He, and C. J. Xue. Compiler directed write-mode selection for high performance low power volatile pcm. In *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '13, page 101–110, New York, NY, USA, 2018. Association for Computing Machinery.

[17] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu. Raidr: Retention-aware intelligent dram refresh. *SIGARCH Comput. Archit. News*, 40(3):1–12, jun 2012.

[18] Y. Omori and K. Kimura. Non-volatile main memory emulator for embedded systems employing three NVMM behaviour models. *IEICE Trans. Inf. Syst.*, 104-D(5):697–708, 2021.

[19] E. Philofsky. Fram-the ultimate memory. In *Proceedings of Nonvolatile Memory Technology Conference*, pages 99–104, 1996.

[20] M. Poremba, T. Zhang, and Y. Xie. Nvmain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems. *IEEE Computer Architecture Letters*, 14(2):140–143, July 2015.

[21] K. Qiu, Q. Li, J. Hu, W. Zhang, and C. J. Xue. Write mode aware loop tiling for high performance low power volatile pcm in embedded systems. *IEEE Transactions on Computers*, 65(7):2313–2324, July 2016.

[22] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[23] D. Sanchez and C. Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 475–486, New York, NY, USA, 2013. Association for Computing Machinery.

[24] A. Sheikholeslami. Operation principle and circuit design issues. In Y. A. Hiroshi Ishiwara, Masanori Okuyama, editor, *Ferroelectric Random Access Memories - Fundamentals and Applications*, pages 149–164. Springer Berlin, Heidelberg, 2008.

[25] A. Suresh, P. Cicotti, and L. Carrington. Evaluation of emerging memory technologies for hpc, data intensive applications. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 239–247, 2014.

[26] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, Dec 2010.

[27] C. J. Xue, Y. Zhang, Y. Chen, G. Sun, J. J. Yang, and H. Li. Emerging non-volatile memories: Opportunities and challenges. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '11, page 325–334, New York, NY, USA, 2011. Association for Computing Machinery.

[28] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 14–23, New York, NY, USA, 2009. Association for Computing Machinery.