Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2012

MSc THESIS

Reordering DRAM Requests for Improved Bandwidth Utilization

A.A.J. Geursen

Abstract

The performance gap between processors and memory has grown larger and larger in the last years. With the emergence of the multicores this problem is additionally accelerated. Stalling row changes in the DRAM increase this gap even more, because the maximum theoretical bandwidth cannot be reached. The solution that is proposed in this thesis is to reorder the requests originated by multiple processing elements inside the memory controller. More data is accessed in burst mode and less stalling row changes are needed when the requests are reordered. The proposed memory controller has a modular setup and does not require any changes of the CPU or the executed software. A few reordering policies have been considered, but one has been implemented as it showed the best expected results in the preliminary analysis.
The speed-up that is obtained by reordering requests varies, but is up to 1.58 for a 3D-FFT and 1.40 for a Conjugate Gradient (CG) workloads with just 64 entries in the sorting array. Power analysis on the DRAM and memory controller shows that the activation power of the DRAM is lowered up to 40% because of the reordering. The total energy that is necessary for each application is reduced with a maximum energy saving of 26.6% for the 3D-FFT and 13.2% for the CG application. This shows that the bandwidth utilization of the DRAM interface is increased while saving energy by using reordering of requests.

CE-MS-2012-01

Reordering DRAM Requests for Improved Bandwidth Utilization

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

A.A.J. Geursen
born in Amsterdam, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Reordering DRAM Requests for Improved Bandwidth Utilization

by A.A.J. Geursen

Abstract

The performance gap between processors and memory has grown larger and larger in the last years. With the emergence of the multicores this problem is additionally accelerated. Stalling row changes in the DRAM increase this gap even more, because the maximum theoretical bandwidth cannot be reached. The solution that is proposed in this thesis is to reorder the requests originated by multiple processing elements inside the memory controller. More data is accessed in burst mode and less stalling row changes are needed when the requests are reordered. The proposed memory controller has a modular setup and does not require any changes of the CPU or the executed software. A few reordering policies have been considered, but one has been implemented as it showed the best expected results in the preliminary analysis.

The speed-up that is obtained by reordering requests varies, but is up to 1.58 for a 3D-FFT and 1.40 for a Conjugate Gradient (CG) workloads with just 64 entries in the sorting array. Power analysis on the DRAM and memory controller shows that the activation power of the DRAM is lowered up to 40% because of the reordering. The total energy that is necessary for each application is reduced with a maximum energy saving of 26.6% for the 3D-FFT and 13.2% for the CG application. This shows that the bandwidth utilization of the DRAM interface is increased while saving energy by using reordering of requests.

Laboratory : Computer Engineering
Codenumber : CE-MS-2012-01

Committee Members :

Advisor:	Dr. ir. Georgi N. Gaydadjiev, CE, TU Delft
Advisor:	Ir. Bogdan Spinean, CE, TU Delft
Chairperson:	Dr. Koen Bertels, CE, TU Delft
Member:	Prof. dr. ir. Henk Sips, PDS, TU Delft
Member:	Dr. ir. Arjan J. van Genderen, CE, TU Delft

Dedicated to my parents and friends

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Georgi Gaydadjiev for coming up with this thesis assignment, the valuable input on the subject, and constructive criticism. I would also like to thank him for making it possible to visit the Barcelona supercomputer as part of a short city-trip. I am grateful to Bogdan Spinean for the supervision, the regular meetings, brainstorm sessions, useful input and talks during my work. It was a pleasant time to work with him. I would also like to thank the administrators and secretary of Computer Engineering, Evert and Eef for the technical support and assistance during my thesis work, and Lidwina for the assistance in the paperwork. I am thankful to the guys sitting in the master thesis room, Robert, Johan, and Aad for the coffee and tea, talks, absurd discussions and fun during the work.

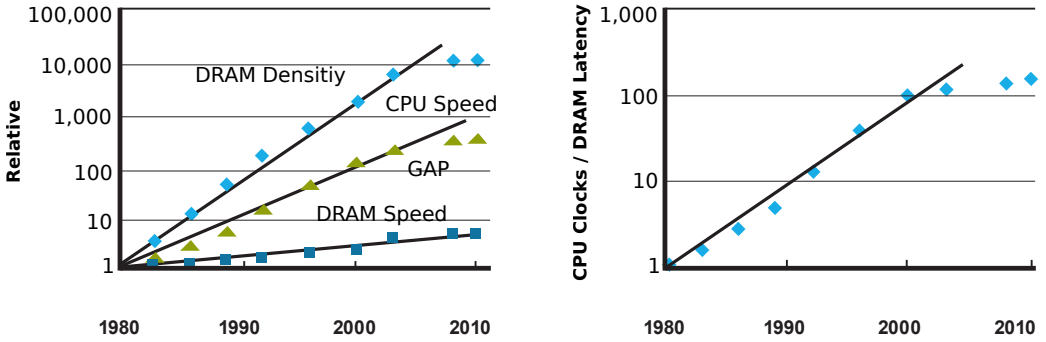
Those who cannot be thanked enough are my parents, they supported me throughout my whole study in all possible ways. Trying to explain my work in a way they could understand also helped me to counter the problems I was facing. Last, but certainly not last, I would like to thank my friends for the lunches, diners, parties, and other support that makes studying a lot more enjoyable.

A.A.J. Geursen
Delft, The Netherlands
February 3, 2012

Introduction and Problem Statement

1

The frequency scaling that was one of the main drivers behind processor performance increases has stopped. The device features sizes keeps scaling, as Moore predicted [20], hence multiple cores are now placed on the same die to improve the computational power. These multiple cores all share the same memory interface and have to compete for the available main memory bandwidth. GPU's have similar problems with up to thousands of cores sharing the same last level memory. The rate of improvement in computational power for the processors is a lot higher then the improvement of the DRAM memory speed [29], this is also made visible in figure 1.0.1.



(a) The gap between the speed of a DRAM and CPU

(b) Amount of clock cycles of a CPU per DRAM latency

Figure 1.0.1: DRAM density and performance, 1980-2010.

Image source: The Future of Microprocessors [6]

The bottleneck for memory intensive applications is clearly the DRAM. Caches in the processor are used to cover up the delays if data is accessed more then once, but for certain applications these caches cannot be used efficiently. An example of this is when the amount of accessed data exceeds the cache size and the cache-lines are renewed every access.

1.1 Problem Statement

The performance gap between the computational power and memory gets large when multiple processors are used to calculate on the same data set. This is the case for scientific problems where multiple processors are collaborating while solving the same problem. The memory that holds the data set should be able to provide the required data at maximum bandwidth when the memory is the bottleneck of the application. The

maximum (theoretical) bandwidth is however not easily achievable in practice.

The DRAMs are holding a lot of data in a matrix structure with long rows. A DRAM is stalling, and will provide the data with a delay, when it has to change rows during a data transfer. The actual used bandwidth lies therefore lower then the maximum theoretical bandwidth, especially when a DRAM is accessed in a non-sequential way and too many of these stalling row changes are happening. The exact reasons for DRAM stalling are described in more detail in section 2.2.

1.2 Proposed Solution

The solution that is proposed in this thesis is to reorder the requests before issuing them to the DRAM. This should increase the used bandwidth and approach the theoretical maximum bandwidth. More data is accessed sequentially and less stalling row changes are happening when the requests are reordered. Many of the requests-in-flight are independent of each other. So reordering across the streams is thus possible. The requests can be packed together with the knowledge of the DRAM organization. With this approach no changes are necessary in the CPU or software when a memory controller is used to do this reordering. This will make sure that all applications can use this solution. The memory controller will receive the incoming requests and reorder them as long the DRAM is busy. A request is sent from the memory controller to the DRAM as soon as the DRAM can accept them to minimize the extra latency that is added by the controller.

1.3 Thesis Outline

This thesis is organized as follows. Chapter 2 presents background information on the DRAM, sorting algorithms, and other related work. Chapter 3 describes the proposed solution for the memory controller. The research that is done on the reordering policies, allocation of space in the data buffer, and memory consistency is described in chapter 4 while the preliminary analysis with the software simulations on the several reordering policies is described in chapter 5. The implementation of one reordering policy in hardware is described in chapter 6. The results of the speed-ups and scalability of the implementation are given in chapter 7. The conclusions of these results are given in chapter 8.

2 Background

In this thesis a memory controller is introduced to approach the maximum bandwidth of a DRAM, in this chapter relevant background information will be described. At first, section 2.1 introduces a memory controller and describes why a memory controller is used. A small introduction will be given to DRAM in section 2.2. After reading this section it will be clear how DRAMs work, what prevents the DRAM to gain its maximum bandwidth, and how this is fixed. Also it should be clear why some techniques like interleaved bank access are used in this thesis for approaching the maximum bandwidth of the DRAM.

In section 2.3 some background is provided on sorting algorithms. These algorithms can be used for the optimization that is implemented in the memory controller. Section 2.4 will give a brief overview of some work that is related to accessing the memory more efficiently.

2.1 The Memory Controller

In this thesis a memory controller is introduced that will approach the maximum bandwidth of a DRAM. This section will describe what a memory controller actually is, and why a memory controller is used instead of connecting the CPU directly to the memory chips.

A CPU does not directly control the signals of the memory when it wants to access data in a memory. The CPU will use a bus to send a request to a memory controller, where the request will be translated into the appropriate electrical signals. The CPU design is not affected by the memory type as long as the bus from the CPU to the memory controller remains the same. A visualization of the memory controller and the signals can be seen in figure 2.1.1.

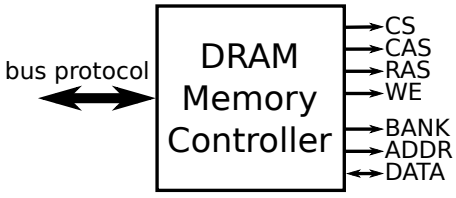


Figure 2.1.1: The memory controller with the bus interface and DRAM control signals.

The signals from the memory controller are used to open a row, read or write data and to refresh the DRAM. These refreshes are necessary when a DRAM is used, as described in 2.2.1.2. A memory controller is thus a piece of hardware that makes sure that data from the DRAM is being refreshed and that this data can be accessed using

simple requests over a general bus. In this thesis a memory controller is introduced that will have some extra functionality to get the DRAM operating at maximum bandwidth, besides the basic functionality of a memory controller.

2.2 DRAM

In this section the DRAM is described to get a more close idea how a Dynamic Random-Access Memory (DRAM) is build up and why it is necessary to optimize the accesses. The description of how data is stored and accessed will give a better understanding on why the DRAM can stall when processing multiple requests.

A short description of SRAM and DRAM and the differences between those two are given in section 2.2.1. How the DRAM will store and reads it data is described in section 2.2.2. In that section the disadvantages of the DRAM are presented. The DRAM memory chip uses some methods to get closer to the maximum bandwidth and are described in section 2.2.3.

2.2.1 Comparing SRAM with DRAM

In this subsection a comparison is made between Static Random-Access Memory and Dynamic Random-Access Memory. SRAM is presented in section 2.2.1.1 where its basic properties are given while the DRAM is presented in section 2.2.1.2. In these two sections is described how a data bit is stored and accessed, and what the costs are to store that bit. A comparison is made between SRAM and DRAM regarding the costs and speed in section 2.2.1.3.

2.2.1.1 Static Random-Access Memory

Six transistors per bit are needed for storing a bit in a conventional Static Random-Access Memory, or SRAM. Four transistors are used to form a latch to store the value of the bit and two other transistors will provide the access to the value. See figure 2.2.1. To store a value into the cell the word-line (*WL*) is set to high and the value that is present on the bit-line (*BL*) is latched into the cell. The value is read by driving the word-line and sensing the value present on the bit-line.

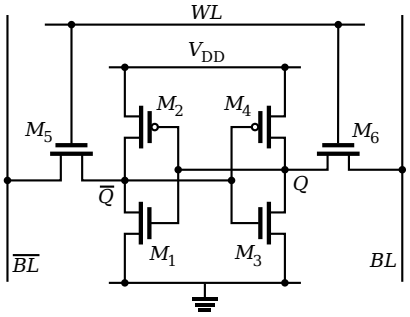


Figure 2.2.1: A SRAM cell build with six transistors, bit-line (*BL*), and word-line (*WL*).

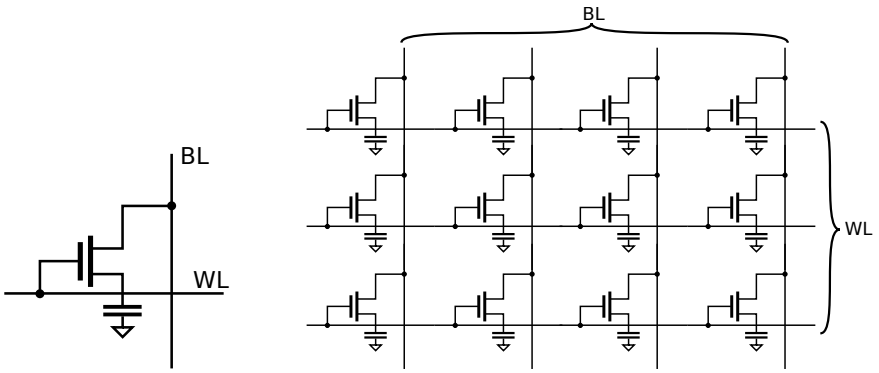
Image source: http://en.wikipedia.org/wiki/File:SRAM_Cell_(6_Transistors).svg

It is also possible to use a setup of four transistors, but this setup is more sensible to noise [24]. Other kind of SRAM chips have eight, nine or even more transistors for several reasons. These setups can provide more ports or higher write stability [3]. Due to the latch structure of the SRAM cell it is not necessary to refresh the data that is stored, the value will be stored in the cell as long as there is power.

2.2.1.2 Dynamic Random-Access Memory

There is just one transistor and a capacitor needed to store a bit in case of a Dynamic Random-Access Memory (DRAM), as shown in figure 2.2.2a. The capacitor is charged to hold the needed value, when this is necessary. This is done by first biasing the transistor to $V_{cc}/2$ and then putting $+V_{cc}/2$ for a logic '1' or $-V_{cc}/2$ for a logic '0' across the capacitor. Reading is done in a similar way, the transistor is biased and the value of the capacitor is sensed. How the DRAM is read and written is described in more detail in section 2.2.2.

Due to leakages in integrated circuits it is not possible to keep the charge in the capacitor for a long time. The charge is slowly leaking away and with that also the stored value. To keep the value stored in the cell it is thus necessary to read the value of each cell and write them back after a certain amount of time. This is called a refresh.



(a) Single DRAM Cell with the bit-line (*BL*) and word-line (*WL*)

(b) An array of DRAM Cells with the bit-lines (*BL*) and word-lines (*WL*)

Figure 2.2.2: The DRAM cell: a single transistor with a capacitor.

To form a memory array, multiple cells are put together in multiple rows and columns as can been seen in figure 2.2.2b. Driving a word line will select the whole row, making it possible to access the capacitors of that row. In this way it is possible to form an array of any size, this is however not without consequences. A long word-line will introduce longer delays because of the increased resistance and capacitance of that word line. A larger parasitic capacitance to the bit-lines is introduced when more rows are used [22].

2.2.1.3 Differences

The maximum size that a single SRAM memory chip can contain is smaller then a DRAM, this due to the large amount of transistors that is needed to form a single SRAM cell. Because of this and the fact that the SRAM cell constantly needs a current flowing through the transistors, the power that is used by the SRAM is also higher then the DRAM. The advantage of SRAM over DRAM however, is that the speed of SRAM lies higher than DRAM. Random read and write accesses are faster in SRAM because there is also no need for precharging and activating rows before data can be accessed. This makes SRAM well suitable for caches, registers and small memories on a micro controller. For the main memory only DRAM is used, it is cheap and larger memory blocks per die are possible. The disadvantage of this is that it is necessary to come up with ideas and techniques to cover up the delays DRAM is introducing as described in 2.2.2. An overview of the differences is given in table 2.1.

	SRAM	DRAM
Area	large	small
Power usage	large	small
Speed (latency)	small	large
Random access times	small	large
Cost per bit (€)	large	small

Table 2.1: The differences between SRAM and DRAM put in one table.

2.2.2 Accessing the DRAM

This subsection is devoted to data access in DRAMs. How the accesses internally are handled by the DRAM is described in the following sections, with reading data in section 2.2.2.1 and writing data in section 2.2.2.2. This will present the stalls that a DRAM is suffering from. Section 2.2.2.3 contains a short description of the DRAM refresh mechanism.

2.2.2.1 Reading a DRAM

As in the previous sub section was described, data in a DRAM is stored in capacitors. The capacitor is discharged over the bit-line when a bit is read from a data cell, the read operation is thus destructive. This small charge is then being sensed with a circuit like the one in figure 2.2.3. The value of the capacitor is being sensed by precharging the data-lines, accessing the bits by switching on the correct word-line, and then firing the sense amplifier latch ($NLAT^*$) and active pull-up (ACT). The row is now activated and the data bits are stored in the sense circuit. The bit-line is brought to V_{cc} or V_{ss} while sensing the data bit and thus refreshing the stored value in the capacitor [14]. So after a read is performed, there is no extra operation needed to restore the values on the capacitor before closing a row. This closing is done by switching of the word-line and precharging the bit-lines again.

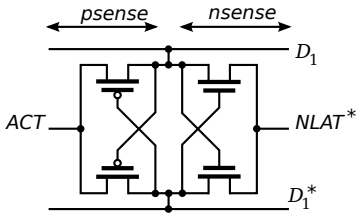


Figure 2.2.3: A DRAM sense circuit used for reading the data from the DRAM cells.

When a read is performed, not just one single bit is sensed, but an entire row (page) of 256 bits for example. That data is still available at the sense amplifiers when something is read in the same row, but in a different column. The time to select a new column is given by the column access select (CAS) latency. By just adjusting the column select, and thus driving different I/O-transistors, the correct data lines are routed to the output or input. So performing multiple reads in the same row has the benefit that it is not necessary to precharge again for each read. As expected, for a read in a different row it is necessary to close the current row, precharge, and activate that new row before the new read can take place.

2.2.2.2 Writing a DRAM

The row that is written must first be precharged and activated in case of a write. The activated sense amplifiers are then driven to the desired value by the write drivers. The bit-lines are then brought to full V_{cc} and V_{ss} values according to the value of the write drivers. It is also possible to write multiple times in the same row without extra delay, like the reading. For reading after writing in the same row however, it is necessary to wait at least the write recovery time t_{WR}. For the other way around, writing in the same row after reading, there is no extra stall and the write can be done immediately after the read.

2.2.2.3 Refreshing the DRAM

The DRAM cells slowly lose their charge over time, because there are leakage currents in integrated circuits. It is necessary to refresh the DRAM cells at regular time intervals to make sure that the stored value is maintained. As described in section 2.2.2.1 a capacitor is refreshed when the row is being activated, the sense amplifiers are then pulling the data line to V_{cc} or V_{ss}. DRAMs have an internal counter to make sure that each row is refreshed automatically every 64 ms. It should be clear that during this refresh the sense amplifiers cannot be used for sensing rows and the DRAM is thus stalling.

2.2.3 Methods to get the DRAM to Maximum Bandwidth

In the previous subsection is described that the DRAM stalls when a row has to be precharged or activated. To cover up some or even all of this delay, DRAMs are using some techniques to get the DRAM operate closer to the maximum bandwidth theoretically achievable. A few of these techniques are described in this subsection.

2.2.3.1 Multiple Banks

During a precharge or activation of a row in an array there is no transfer of data possible in that array. There is a way to cover up these stalls, when multiple arrays or so called banks are used. A setup of this is show in figure 2.2.4. While reading or writing in the first bank, it is possible to refresh, precharge or activate a different bank. When timed right it is thus possible to keep on reading from different locations in the memory as long as these locations are in a different bank or in the same row [22].

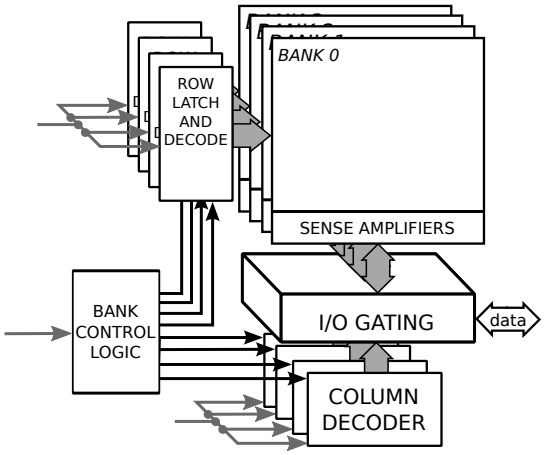


Figure 2.2.4: Four banks of a DRAM and the needed logic for controlling the individual banks

An extra benefit of the bank structure is that the amount of sensed data is higher and thus more words can be accessed immediately. The consequence of this multiple bank setup is that more power is used when multiple banks are active. Also the area is increased a bit due to extra control logic required. However, the benefits out-weigh the costs using banks. As long as there are no changes of a row in the same bank while reading or writing, the stalls to the outside world can be minimized and the maximum bandwidth can be reached. This is achieved with a data mapping across multiple banks.

2.2.3.2 Burst Reads and Writes

For each read or write the appropriate row and column addresses have to be strobed into the DRAM. During this strobing it is not possible to issue another command like precharging an other bank. To have the possibility to get more data out of the DRAM without extra command strobing, the burst mode is available. One command will read or write multiple data at consecutive memory locations. During a burst it is possible to issue new commands to the DRAM, because the command-lines are busy for just one cycle while the data-lines are busy for multiple cycles. This means that during a data transfer the command-lines are available to perform commands in other banks and the bank system as described in section 2.2.3.1 can be used more efficiently.

2.2.3.3 Double Data Rate

For each clock cycle it is possible to fetch new data from the sense amplifiers. The frequency of this has a limitation because of the signal integrity. A workaround to have more data transferred in the same time is used in the Double Data Rate DRAM where data is transferred on both clock-flanks. Instead of just fetching n-bits, the DRAM will also prefetch the next n-bits. This $2n$-prefetch is possible by doubling the width of the internal data bus and adding an extra input/output register with some minimal sized control logic. The internal frequency can be half of the external data transfer rate when DDR is used [19].

2.2.3.4 Multiple Channels

Another way to increase the maximum bandwidth, without relying only on the memory speed is having multiple channels [13]. The memory controller has separate buses or channels to the DRAM modules and is able to read or write data in parallel. In the ideal case each DRAM chip has a separate channel to the memory controller, but this is not practical due to the amount of wires that are needed. The differences in delays of each channel are becoming to large besides the increased area. Two or three channels are used too double or triple the amount of data that can be read or written at each clock-cycle.

2.2.3.5 Reordering Incoming Requests

The maximum theoretical bandwidth can be reached by reordering the incoming requests in the memory controller as proposed in this thesis. Rows will be changed less often compared to the case where no reordering is done. The banks will also be used more efficiently when reordering is done. This is because requests with the same row address will be packed and issued after each other. Banks can also be precharged for the next requests while processing a request in another bank, covering up the stalls for the memory controller.

2.3 Sorting Algorithms

In this section some sorting algorithms are described that can be used as a basis for the reordering of requests. Required for the sorting of the requests is that:

- The sorting is done at runtime, the memory controller should not wait for all requests being received.
- A request must not wait much more than others (on average).

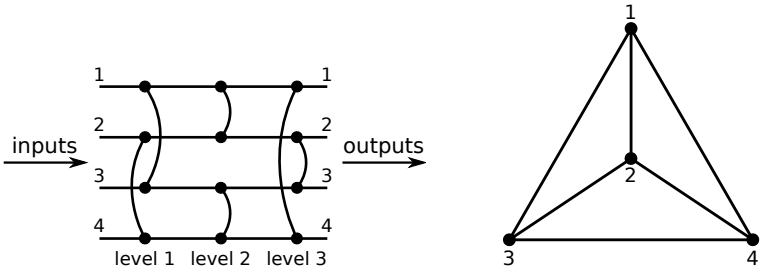
In this section only the algorithms are discussed that are used to sort. How the sorted elements are treated and kept of from starvation as given by the second requirement is discussed in section 4.1.

The algorithms discussed here are designed for hardware implementation. With a short description of the algorithm, there is also an estimation on how much time the

algorithm would take to complete, if it can operate on the run, and how many processing elements are needed for this algorithm. Each algorithm will need a different kind of processing element with a different size, so the comparison in hardware is not as strict as the time is. Only the amount of processing elements will be given, not the actual number of gates or area size. These complexity evaluations are given in terms of the big O notation.

2.3.1 $O(\log n)$-Level Parallel Sorting

Multiple entries are sorted at once with the parallel sorting described here. This means that the sorting needs multiple requests being present when the sorting is started. At each step comparators are comparing two requests and swap them if needed. In the next steps this will happen again but now with the entries that where possibly swapped in the previous step. At the end of this $\Theta(\log N)$-degree network all numbers are sorted when all comparators are connected to each other in the correct way as seen in figure 2.3.1a for 4 numbers. Each edge represents a compare and swap element. The two inputs of the element are compared and swapped depending on which one is larger. It is easily seen that in this way the output has all sorted elements after $O(\log N)$ steps. With this structure it is possible to use pipelining, but costs $M/2$ processing elements for each stage where M is the number of entries that can be sorted each time.



(a) A 3-level circuit

(b) A 4-node degree-3 network

Figure 2.3.1: Parallel sorting structures for sorting four items.

When this structure is converted to a node network where each node is connected to all other nodes, as shown in figure 2.3.1b, less edges and thus elements are used for the sorting. These $\Theta(N \log N)$ elements will do the exact same as with the previous setup, only there should be an overlooking controller present. This controller assigns each number to the correct processing element at the right time. The sorting of N numbers will still take $O(\log N)$ steps, but no pipe-lining is possible [15]. At the end of this $O(\log N)$ steps, the ith smallest number can be found in the ith node.

2.3.2 Two Step Sorting Algorithm

Instead of using $O(\log N)$ steps to sort N elements as was done in section 2.3.1, the two step sorting algorithm takes just two steps. Also for this algorithm all requests have to be present when the sorting is started. The sorting is done by using $O(N^2)$

processing elements, which are called binary neurons in the algorithm, $O(N^3)$ switches and $O(N^2)$ comparators [27]. The binary neurons use a function that is based on a simplified biological neuron and is given in equation 2.3.1. In this equation, U_i is the input of the ith neuron and V_i the output.

$$V_i = f(U_i) = \begin{cases} 1 & \text{if } U_i > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.3.1}$$

With an $N * N$ array of these neurons, $N * N$ comparators and N^3 switches the algorithm described in [27] can be implemented. Because the amount of hardware that has be used is a lot, the algorithm is just too costly to be implemented and therefore it is not discussed further.

2.3.3 Insertion Sort

It is necessary to have an array filled with request when the sorting can start for the previously discussed algorithms. This is not the ideal case because waiting for requests means also lost time where data could have been processed. Furthermore it is necessary to have a fixed number of request. The algorithm should be able to cope with less requests when there are just not enough requests. In that case a decision has to be made when to start sorting with the requests currently available. In the insertion sort algorithm however, every element is immediately sorted when it is entered. This means that the total sorting time is $O(N)$ for N requests. This is more then the previous algorithms when just looking at the sorting time. The transfer time for all requests is in all cases $O(N)$, only in the case of the insertion sort all elements are already sorted after the transfer.

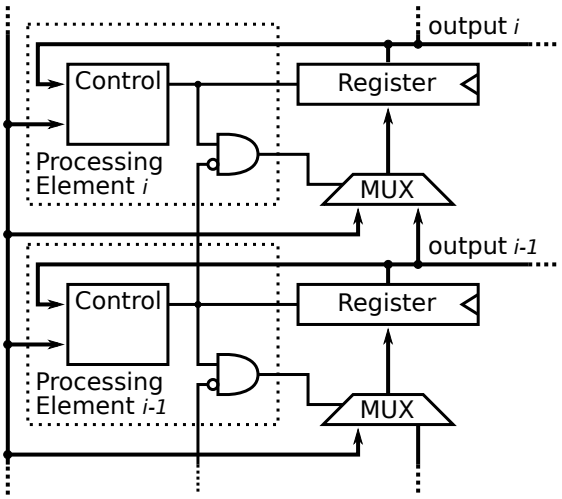


Figure 2.3.2: Basic scheme for the insertion sort algorithm. At each location a processing element will control the register and multiplexer.

Every inserted request has to be compared to each request already present. In software this can be done in $O(\log N)$, but in hardware with a processing element for

each position this could be done in $O(1)$ [5]. The position has be chosen where the request is being entered based on the results of the comparison with each element. When the position is chosen a part of the requests have to shift to make room for the new request. A basic scheme of some sorting elements is given in figure 2.3.2.

Each processing element can easily decide if the current request has to be shifted to the next location or not because each request is already sorted. The processing element at position i inserts the new request when the element at position $i-1$ is not shifting its request. The criteria for this decision is if the request at position $i-1$ is smaller or larger then request being inserted. This decision is only based on position $i-1$, no other positions are required. This means that no extra delay is added when extra locations with processing elements are added.

2.3.4 FIFO-Based Merge Sorting

With a merge sorting algorithm, there are two sorted sets already present and these sets are merged into one single sorted set. So the FIFO-based merge sorting discussed here is also using a sorting algorithm to create the sorted sets. The merge sorting is not entirely a way of sorting on its own. The advantage of a merge sort is that the implementation of the sorting algorithm that is used does not have to be of the size of the whole data set that has to be sorted. By using small sorted sets the implementation of the sorting algorithm is kept small. However, to get the whole set sorted several iterations have to be made which takes $O(M+L)$ time each iteration where M is the size of the inserted sorted set and L the amount of elements already present in the FIFO.

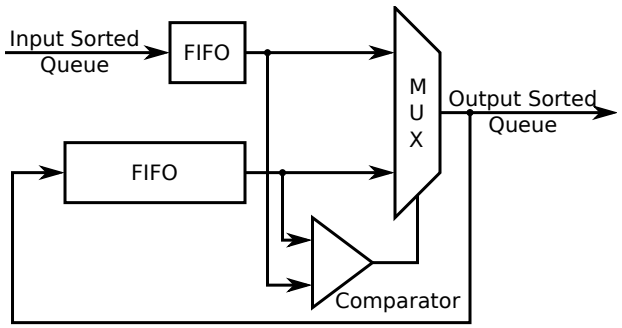


Figure 2.3.3: FIFO-based merge sorting

Two FIFO's are used to store the sorted sets in the FIFO-based merge sort [17]. As can be seen in figure 2.3.3 a single comparator and multiplexer are used to generate the new sorted set. The merging is done by letting the comparator decide which of the outputs of the two FIFO's is being transferred and which one will be compared again in the next round. This happens every clock cycle until all elements are processed and the upper FIFO is empty. The requests can then be transferred in order to the output. New requests cannot be sorted while the transfer to the output is not completed. In the case that the merging is taking place it is not possible to have requests being transferred to the output. This means that when this merge sorting is implemented a full-duplex[1]

[1] In a full-duplex system it is possible to send and receive at the same time.

system is not possible. However, when a system is considered without a full-duplex functionality this merging can be advantageous.

2.4 Related Work

In this section work is described that is related to accessing memory in a more efficient way. The described solutions differ from an application side solution that can run on existing systems to a complete DRAM controller. All propositions try to access the memory more efficiently to cover up the speed differences between DRAM and CPU.

Pichel et al. [21] propose a solution for accessing the memory more efficiently in the case of sparse matrices. A technique for increasing the locality is introduced that consists on reorganizing the data at the application side, no hardware changes have to be made. By increasing the locality, the cache and memory are used more efficiently. For applications that do not use a sparse matrix this solution has no large improvement because the memory it self is still accessed inefficiently.

An other software based solution is proposed by Hu, et al. [12] where a small library is created allowing to optimize an application by adding a few lines of code. The locality of data is improved by the data reordering techniques that relocate the data in the memory. The application where the few extra lines are added in combination with the library do not need further changes. This makes it easy to use the reordering at existing source codes.

The Impulse memory controller focuses more on the improvement in the cache and bus utilization [7] then the accesses of the memory itself. Shadow addresses can be used by using addresses outside the memory space. The impulse controller can remap the shadow addresses to the correct non-continuous data locations in the memory. The Impulse memory controller can be configured by the OS making if possible to use different types of remapping. By using this shadow address concept the cache is used more effectively because only the needed data is sent. This results in a higher cache hit rate. Also less requests have to be sent to the controller, which results in less bandwidth for the bus. The main memory itself is not accessed in a more efficient way.

A solution that focus on the access of the memory itself without influences of software is the memory access scheduling proposed by Rixner et al. [23]. This solution does the scheduling at DRAM controller level by scheduling the commands like precharge and activate. Banks can be precharged or activated efficiently by using multiple memory access requests. The implementation of the proposal of Rixner et al. is for a streaming memory system, where a single request reads a stream of data. The accesses of data for the streams are optimized without the need of changing software applications, cpu, or buses. The streams are generated by address generators and put together in reordering buffers. The single data requests from the generated streams are scheduled for a more efficient access to the memory.

The memory controller that is presented by Lin, et al. [16] is another proposal with a quality-aware scheduler. This memory controller is intended for multimedia platform system-on-chips and features a configurable DRAM memory interface socket besides the scheduler. This memory interface socket is used to translate the accesses granted by the scheduler into the correct commands for the DRAM.

Another solution, where the access optimizer is implemented at the same silicon as the DRAM cells, is proposed by Hitachi [28]. This is a simple access scheduler that can be used as a part of a system-on-chip. The access optimizer consists of units for self prefetching and address alignment to increase the hit rate and decrease the fill latency of the cache. This solution is only useful for small, single die, systems.

3 Proposed Solution

The goal of this thesis is to have a memory controller that will get the DRAM operate as close as possible to the maximum theoretical bandwidth. In this section the proposed solution is given.

The memory controller is located between a bus interface and the DRAM, as described in section 2.1. This requires a part in the design for the communication with the bus interface and one that communicates with the DRAM. For this proposed solution a compiler like structure is chosen with a Front-End, Back-End and a Optimization Block in between to make the proposed solution flexible and easy interchangeable with different kind of bus interfaces and memories. The block in the middle will be the Optimization Block with a number of core elements to make the sorting possible. A diagram of this structure is given in figure 3.0.1.

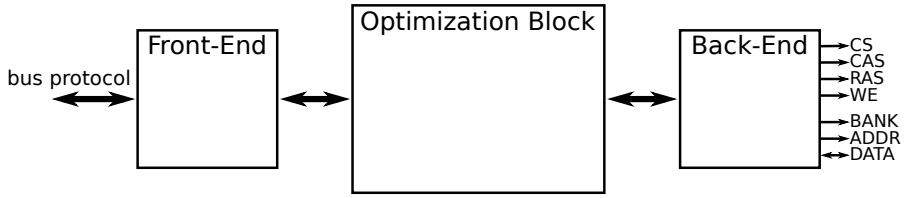


Figure 3.0.1: The proposed structure with the Front-End, Optimization Block, Back-End, and signals between the blocks and the outside world.

In section 3.1 is the portability and flexibility of the proposed solution discussed. The proposed solution has a modular design where easily other buses and policies can be implemented. The Front-End and its flexible bus interface is described in section 3.2 while in section 3.3 the Back-End is described in more detail: how it should operate in the system and how it is connected to the Optimization Block. The Optimization Block with its core elements explained can be found in section 3.4. Section 3.5 presents the limitations of the modular design.

3.1 Portability and Flexibility

With the Front-End, Optimization Block, Back-End structure it is possible to use the controller in different designs and platforms. By just changing the Front-End a different bus interface can be used as described in section 3.2. Changing the Back-End makes it possible to use the design with a different memory connection as described in section 3.3. Small changes have to be made when a Double Data Rate SDRAM is used. It is necessary to increase the data-width inside the system besides having a different Back-End in that case. The DRAM controller will make sure that each part of the data is transferred during

a rising or falling clock edge. The changes are confined to the Back-End, the Optimization Block and the Front-End are not influenced.

The data width increment is made very easy, changing just one generic number in one file gives a new design with a different data width. Other properties, like the maximum request size, the size of the data buffer and the number of entries in the sorting array can be found in one file. This will make sure that there is no need in digging into the code when such a simple property as the bus-width is changed. The same holds for the amount of entries in the FIFO' s for requests entering the system or waiting to be returned. A simple generic number change is enough when a larger FIFO is needed.

The proposed solution described in this chapter is a modular design where it is easily configurable. Besides the Front-End and Back-End as described before, various sorting policies can be implemented with little effort. An implementation of a new policy is easily created, because the sorting array has an interface that is comparable with a FIFO. This means that for a different policy it is only needed to change the sorting array with a version that has the new policy implemented, the rest of the design can be left untouched. The same holds for the data buffer with its allocation policy, the rest of the design is not effected by a different allocation policy of the data buffer. A new allocation policy would mean that just the data buffer has to be changed.

3.2 Flexible Bus Interface

The Optimization Block that is used for the sorting is built for the most flexible bus interface without introducing any limitations. The supported features for the bus interface are:

- Full-duplex, the bus can send and receive data at the same time.
- Non locking operations when a request is being processed, multiple requests can be send before a response is given.
- Any transfer size, the size of a request does not matter.

The Optimization Block will still work when a bus is used that does not support one or more of the above features. There is no reordering if the bus locks for each request, but that request will still being processed. So with a different bus there is no need to change anything functional in the Optimization Block. This makes it easier to use the optimizations in different designs or platforms. To make this possible a separate Front-End is used that converts the bus interface in such a way that the Optimization Block will understand.

When a new request comes in on the bus interface, the Front-End will pass it to the Optimization Block and let the Optimization Block decide if the request can be acknowledged or not. The connection between the Optimization Block and Front-End in the proposed solution is based on the AMBA AXI Protocol, a protocol that is full duplex, does not lock the bus and can be used with variable transfer sizes. This protocol has separate request, data and returning data channels [2]. This is made visible in figure 3.2.1.

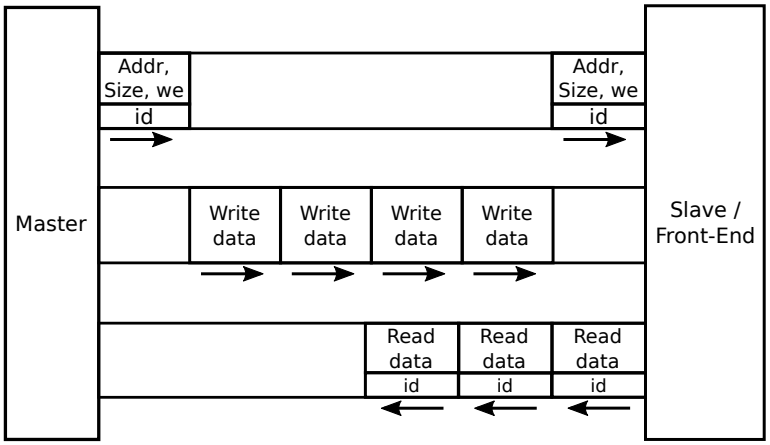


Figure 3.2.1: The Front-End communication channels: The above channel transfers the read and write requests to the slave. Data that has to be written is transferred in the *Write data* channel to the slave. The read data with the request id is returned to the master in the lower channel, there is no need to wait for other channels to be empty.

There is no need for complex logic when the AMBA AXI Protocol is used as the bus interface in the system, this is because the Optimization Block uses a communication with the Front-End that is based on this protocol. Some simple wiring and few logic gates is enough to let the Front-End work as AMBA AXI slave. It is still possible to use different protocols, but a different Front-End should then be designed. The Processor Local Bus, or PLB, is one of the several options one can choose to implement for this design. This bus also supports read and write overlapping and thus can have two data transfers per clock cycle. The separate data and address buses that the AMBA AXI bus has, is also supported with the PLB protocol. For a read the PLB will request a read and waits until this request is acknowledged with the data, something that makes PLB not suitable for a memory controller that uses multiple requests [26].

3.3 Flexible Memory Interface

A Back-End is used to ensure that different types of DRAM can be used with the optimization implemented in the controller. This Back-End is the DRAM controller that issues the correct commands to the DRAM. It will also read data from and write data to the DRAM. The Optimization Block does not need any changes when a different DRAM is used, just a different or modified Back-End. It is however possible that an existing DRAM controller is used where the signals or requests are not compatible with the Optimization Block. In that case a Bridge is needed to convert the requests or signals in a way that the Back-End and the Optimization Block can communicate. The Back-End for the memory controller is taken as is, it is written by someone else and it is not modified.

One or more new requests are read into the Back-End with a simple interface to the Optimization Block. The Back-End will let the Optimization Block know when a request is finished and all data is written from the buffer to the DRAM or the other way around. The further data handling, like returning the read data to the Front-End, is not up to the

Back-End.

There is a possibility that the maximum burst of the DRAM controller does not comply with the one from the requests, in that case a Bridge is needed. In that case the Bridge has to convert the large requests into smaller ones before issuing them to the Back-End. For the Optimization Block there is no need to know if there is a request conversion taking place, so the Bridge should only signal a request as done when the original large request is done. The Bridge will have quite some logic as its own because of the ability to convert requests and controlling of the smaller requests, but the Optimization Block doesn' t need to be changed.

For the Back-End several possibilities are considered, all with their advantages and disadvantages. The one used in the proposed solution is a DRAM controller from OpenCores [25]. This is a fairly simple controller with an easy interface and adaptive bank control. This means that the commands that are issued to the DRAM are depending on the previous accesses. It will also try to limit the stalls by issuing its commands during previous data transfers. The disadvantage of this controller is that it does not work with Double Data Rate DRAM and that it has a maximum burst of 16 words. An other option was a controller from the Xilinx Memory Interface Generator [10]. This memory controller can work with Double Data Rate DRAM and its interface is fairly simple. The OpenCores Back-End is chosen for the proposed solution because there where some problems with integrating the Xilinx Back-End in the design at the time the Bridge was designed.

3.4 Optimization Block Description

The actual optimization is taking place in the Optimization Block by sorting the incoming requests. This is done by one core element, the sorting array. The data that is used for each request is temporarily stored in a data buffer. The data buffer will try to allocate the needed space for each request that is entering the Optimization Block. The requests entering the Optimization Block will be transferred to the sorting array when allocation was possible. A request is read from the sorting array and transferred to the Back-End when the Back-End is able to receive new requests. The Back-End will signal that a request is done when a request is processed, the Optimization Block can than return the read data from the data buffer.

A diagram of the Optimization Block with the Bridge attached is given in figure 3.4.1. In the following subsections more detailed descriptions are given for a few parts of this Optimization Block. At first a description is given of the path a request is following in the Optimization Block in section 3.4.1. A more detailed description for the sorting array is given in section 3.4.2 and in section 3.4.3 the data buffer is explained thoroughly. Section 3.4.4 presents the Bridge that is used in this proposed solution.

3.4.1 The Path of a Request Through the Optimization Block

In this subsection the path of a request through the Optimization Block is described. This will be done by following a request from the Front-End through the Optimization Block,

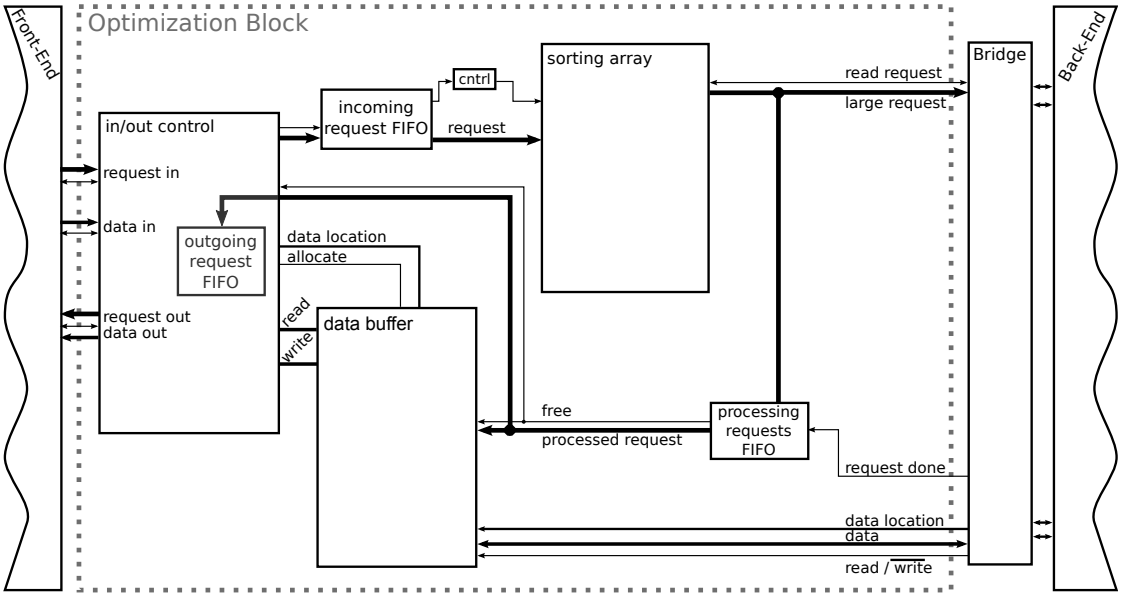


Figure 3.4.1: Scheme of the Optimization Block with the Bridge attached.

passing through all the blocks. Later sections describe in detail how the sorting array and data buffer are operating. In this section the glue logic will have a dominant role.

When a request is coming in at the Front-End, this request will be passed to the input/output controller. This controller will then ask the data buffer to allocate a data block. The request is acknowledged to the Front-End when the data buffer can perform the allocation. The Front-End will receive the appropriate data which is immediately written to the data buffer when the request is a write. The request is written into the incoming request FIFO, with the data location provided by the data buffer. In case the sorting array is not accepting any requests it is still possible to receive requests from the Front-End because of this FIFO. In this way no requests are dropped or simply not acknowledged.

The incoming request FIFO will hold the requests that are already in the data buffer. The requests in this FIFO can be transferred without the need for waiting for allocating when the sorting array is able to accept new requests again. As long as the incoming request FIFO is not full, requests can enter the Optimization Block. A simple and small controller will transfer the requests from the FIFO into the sorting array when possible. The request will be sorted according to the sorting policy when the request enters the sorting array.

The Bridge will read the request from the sorting array to process it. The request is also transferred to the processing request FIFO besides transferring the request to the Bridge. All requests that are being processed are stored in this processing request FIFO. In combination with a request done signal from the Bridge the optimization is able to know when each request is completed. This is possible due to the fact that all requests in the Bridge and Back-End are processed in order. Depending on the request, if it was a read or a write, several things need to happen:

- When the processed request is a **write** the used space in the data buffer can be released.

- In case the processed request is a **read** the read data has to be returned before the used space in the data buffer can be freed. The request is therefore put in the outgoing request FIFO inside the input/output controller. The controller can then start returning the request with the data to the Front-End. The controller signals the data buffer to free the allocated space in the data buffer after all data is returned.

3.4.2 Sorting Array

The goal, to access the DRAM as close as possible at the theoretical bandwidth, is realized in the sorting array. The sorting array is the core element in the optimization process. All other blocks are also needed, but the speed-ups are created by sorting the incoming requests and that is done in the sorting array. This sorting can be done in different ways, or so called policies. How a request should be sorted when it is entering the sorting array and which request is read as first is described in the sorting policy. In section 4.1 there is more described about the different sorting policies. The sorting policy is implemented only in this block, other blocks are not influenced by it. This also means that it is not needed to change other blocks when a different sorting policy is implemented.

When a request enters the sorting array it is sorted immediately or after a while, depending on the chosen policy. The sorting array could take the decision to hold more request and sort them in a more efficient way, but during the that time no requests are processed. The efficiency of the DRAM is then not increased. Instead of waiting for more requests it is better to let the Back-End process already some requests. In either case, the decision of when to sort has to be made by the sorting array, no other block is influencing this. At a moment the Back-End can try to read a request from the sorting array. When to acknowledge this and which request is transferred to the Back-End is up to the sorting array.

3.4.3 Data Buffer

The data buffer has a few functions to operate in the Optimization Block. When reading data from the DRAM this data has to be stored, this to make sure that all data is available when returning the request. All data should be present before returning a request, depending on the bus protocol that is used. The data should be already in the data buffer before writing to the Back-End in case that the controller should write data to the DRAM. This because the Back-End will assume that all data is available so it can issue bursts to the DRAM without a problem.

In section 3.4.3.1 the inputs and outputs for the data and addresses of the data buffer are described. The allocating of a space in the data buffer for each request is described in section 3.4.3.2.

3.4.3.1 Inputs and Outputs of the Data Buffer

To provide the full duplex functionality of the controller the data buffer has multiple inputs and outputs. One address and data channel is used for reading data from the data buffer to the Front-End. Another address and data channel is used to write data from the Front-End to the data buffer. One address and two data channels are being used for reading and writing data to the Back-End. A visualization of the inputs and outputs is given in figure 3.4.2. With this setup it is possible to read data that is being returned to the Front-End it is possible to write from the Front-End to the data buffer at the same time. At that same time the Back-End is also allowed to read or write data in to the data buffer. The structure with multiple inputs and outputs is only possible when the Back-End and Front-End can assume that the data is not accessed when the request is processed or returned. It is possible to have a structure like this because the Front-End will never read and write data at the same location and the Back-End will never have a request to be processed when the data is not there yet.

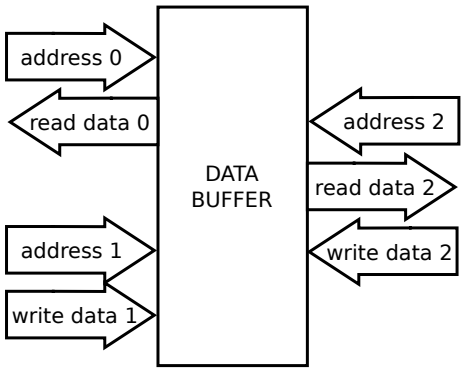


Figure 3.4.2: The data inputs and outputs of the data buffer. The left channels are going to the Front-End and the right channels to the Back-End

The disadvantage is that the amount of extra logic to decode addresses is increased due to extra address ports. It is however, not necessary to have an arbiter that grants accesses to the data buffer or not. Also the effective processing time of a request is lower with multiple ports. The maximum bandwidth of the DRAM can never be achieved with just two ports and an arbiter. If a Back-End is operating at maximum bandwidth, it is reading or writing every clock cycle, meaning that the data buffer is busy serving the Back-End every cycle. This means that the Front-End has to provide the Optimization Block constantly with new requests and data. In that same time the Front-End is not able to return the read data to the bus interface because there is only one input and output port at the Front-End side. To have a full-duplex functionality it is thus necessary to have a total of three address and four data ports.

3.4.3.2 Allocating Data Blocks in the Data Buffer

Requests are processed out of order and thus a simple FIFO for the data buffer is not sufficient. A special allocation policy is needed when requests are processed out of order which causes fragmentation. The data buffer is asked to allocate a space in the data

buffer with a size given by the request when that request is entering the Optimization Block. If there is enough room the data buffer will acknowledge the allocation request and returning the starting position in the data buffer. From that moment on that space in the data buffer is assigned to that particular request. Internally the allocated space in the data buffer is set to used so no other request can allocate space at that location. In case there is not enough space available in the data buffer the allocation request is not acknowledged and the Front-End will not receive an acknowledgment. The allocation policy is described in section 4.2.

When the data buffer receives an allocation request and a signal to de-allocate a space in the data buffer at the same time, the de-allocating has an higher priority than the allocating request. The signal that the used data block can be de-allocated is received after processing the request by the Back-End, and possible the Front-End. The advantage of the higher priority of the de-allocation is that after de-allocating a block the possibilities of allocating a data block is increased. The disadvantage is that a request is being stalled for an extra clock cycle for each de-allocation command.

3.4.4 Bridge

In section 3.3 the Bridge between the Optimization Block and Back-End was mentioned. In this section the Bridge and the blocks that are used will be described. The Bridge used in this proposed solution is for the OpenCores Back-End, a different Back-End may require a different one. A global schematic of the Bridge is given in figure 3.4.3.

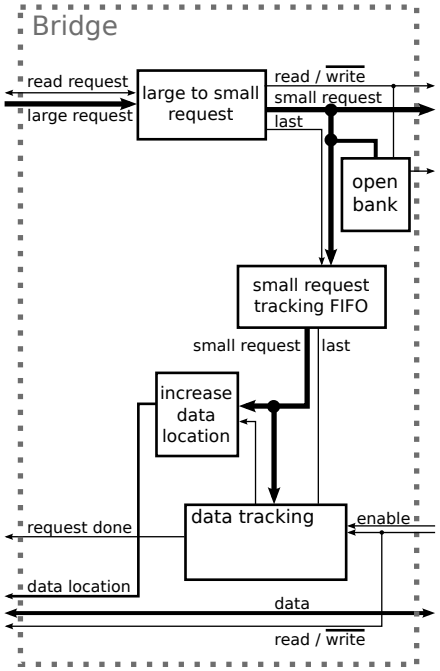


Figure 3.4.3: A scheme of the Bridge with the important blocks and signals.

3.4.4.1 The Large to Small Request Controller Block

The large to small request controller will try to read requests from the Optimization Block when possible. The requests that are handled by the Optimization Block can be of arbitrary size. In the Back-End however, there may be a limit depending on the implementation. The Back-End used in the proposed solution, the OpenCores Back-End, can accept requests up to 16 words. This means that every request from the Optimization Block with a size larger then 16 words has to be split into smaller requests. Each small request is then entering the rest of the Bridge. An extra bit is added to see if the small request is the last request that came from one large request. The reason for this extra bit is explained a bit later in section 3.4.4.2. The Back-End receives the request, besides transferring the small request to the other blocks in the Bridge. The Back-End then knows if it has to read or write data, from what address it has to start reading or writing and how large the burst is. At that moment the Back-End can issue the needed commands to the DRAM.

The request with the extra bit is put in the small request tracking FIFO when the small request is transferred to the rest of the Bridge. In this FIFO the request will be waiting until it is being processed by the Back-End. At the output of this FIFO the request is visible that should be processed at that moment, giving the other blocks of the Bridge the possibility to use its information.

3.4.4.2 Data Tracking Block

One of the blocks that is using the information of the request at the output of the small request tracking FIFO is the data tracking block. The data tracking will track all data interactions from the Back-End with the data buffer in the Optimization Block. For each read and write it will track if a small request is done or not. The data tracking block can determine if a large request, originally read by the Bridge, is processed completely with the extra bit generated by the large to small request controller. The data tracking block will signal the request as done if that is the case. When the last data word is tracked and the request is flagged as done, the Optimization Block knows for sure that all data of that request is processed.

3.4.4.3 Increase Data Location Block

The data tracking will also signal the ïncrease data locationb̈lock to increase the address of the data buffer when needed. This is when a read or write is taken place by the Back-End. No information is skipped and every request is tracked correctly in case the Back-End stalls in the middle of a burst for whatever reason when this signal is used. The increase data location block will load the new data location when a new small request is being processed. This is done in one just clock cycle, so there is no interruption of the data stream. There is no extra time to do any processing when the requests are sorted perfectly and the data stream from or to the Back-End is continuous.

3.4.4.4 Open Bank Check

All blocks described until now can be used for all Back-Ends, with a small variation in the maximum burst used by the large to small request controller. The last block described here however, the open bank check, is probably only needed for the OpenCores Back-End. This DRAM controller will never close a row unless there is a refresh. A positive property when there is enough usage of the bank that will make sure that the bank is not open for to long. When this is not the case and a row in the DRAM is open for too long, there is a chance of loosing data [25]. The open bank check block is to make sure that this does not happen. A counter for each bank is used to see if a row in a bank is open for too long. When there is a change of rows in a bank, the row is closed and the corresponding counter is reset. The open bank check will issue a refresh to the DRAM controller in the case that a counter hits the maximum allowable time. The controller will then do a refresh on all banks and thus closing all rows in the banks.

3.5 Limitations

With the modular design there are also some limitations which are described in this section. These disadvantages arise because of the Front-End, Optimization Block, Back-End structure.

One disadvantage is that the Optimization Block has no direct control over the DRAM. It cannot issue precharge or activate commands to the Back-End for example. On one hand this is a positive thing, this simplifies the Optimization Block. Also when a different DRAM is used which has an other command structure, there is no need to change the Optimization Block. On the other hand, interacting with the DRAM directly would give more possibilities in optimizing the transfers. Removing extra stalls could be possible when the Optimization Block could send the precharge and activate commands in advance before issuing the request to the Back-End. Now the Back-End has to send these commands in advance, if possible.

Another disadvantage is that there is more logic needed, extra FIFO' s are used to keep track of all the requests for example. Because of this there is a limitation of how many requests can be processed in the Back-End at once. This is a trade-off with area and the amount of requests that can be processed at once. There is no need to track a request if there was no separation of the Optimization Block and Back-End. The Optimization Block can issue the needed commands when the requests are still in the sorting array.

Research Questions and Discussions

4

This chapter is considered with the research performed to address the issues with reordering, allocation, and memory state consistency.

Section 4.1 describes the reordering policies that were considered for the memory controller. By reordering the request the maximal theoretical bandwidth of the DRAM is approached, but there are also complications introduced.

The data for each request has to be allocated in the data buffer. This is trivial when requests are processed in order, but for requests that are processed out of order a complex allocation policy is necessary. In section 4.2 the different allocation policies are described.

Another side effect introduced by reordering the requests is data inconsistencies. When for example two requests write at the same location, reordering those requests can cause a memory inconsistency. This memory consistency problem and how this problem can be solved is described in section 4.3.

4.1 Reordering Policy

In this section the different reordering policies are described that can be used in the memory controller. The most important property to base the sorting on in the reordering policy is the address of the request. With only that information the memory controller can make sure that the row in the DRAM is changed as less as possible and that the stalls are minimized.

The reordering policies are used to decide where in the sorting array a request is inserted and which request is read from the sorting array. Just sorting the request based on their address and reading the first request in the array is not enough. Assume that the Back-End reads a request from the sorting array with row address 2 and a two new requests are entering the sorting array with row address 1 and 2. Without a policy the new request with row address 1 will be processed next, instead of the other request with row address 2. This will create a stall that could have been prevented by using a simple reordering policy.

A short description of what sorting algorithm is used for the policies is described in 4.1.1. In sections 4.1.2, 4.1.3, 4.1.4, and 4.1.5 are descriptions given of reordering policies that can be used for a sorting array. Section 4.1.6 describes how even more bandwidth can be gained, independent on what policy is used for the sorting array.

4.1.1 The Request Sorting for the Policies

The sorting that is used for all the policies is based on the row and bank address, by packing the row and bank addresses of a request together where the column address is ignored. Different requests with the same row address are sorted together, but each request is also packed with the requests of the same bank, see figure 4.1.1. The amount of stalls is then minimized when a group of requests with the same row address is processed. The position in the row does not play a role in the stalls, so the column address of the request is ignored for the sorting.

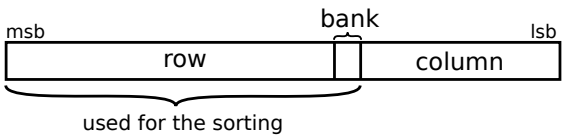


Figure 4.1.1: The address set-up that is used for the sorting. Only the row and bank address are considered.

All policies described in this section are using the insertion sort algorithm, described in section 2.3.3, for the sorting. This algorithm is chosen over the others because the memory controller has to process requests as soon as possible. The requests have to get sorted as they enter the memory controller, without waiting for a minimum amount of requests or time. Each request is inserted at the correct location when the insertion sort algorithm is used.

4.1.2 Smallest Address First, Static Boundary

In this policy the request with the smallest address in the sorting array is sent to the Back-End as first. The requests currently available in the sorting array are protected against new requests with a smaller row and bank address when the first request is read. This is done by setting a boundary around the requests that are sorted in the sorting array. All new requests entering the sorting array after the boundary is set will still be sorted, but not inside the boundary.

The Smalles Address First policy with a static boundary is explained with an example in figure 4.1.2. The boundary is set over the new requests when all requests inside the boundary are processed, the boundary is not changed before. This makes the boundary static.

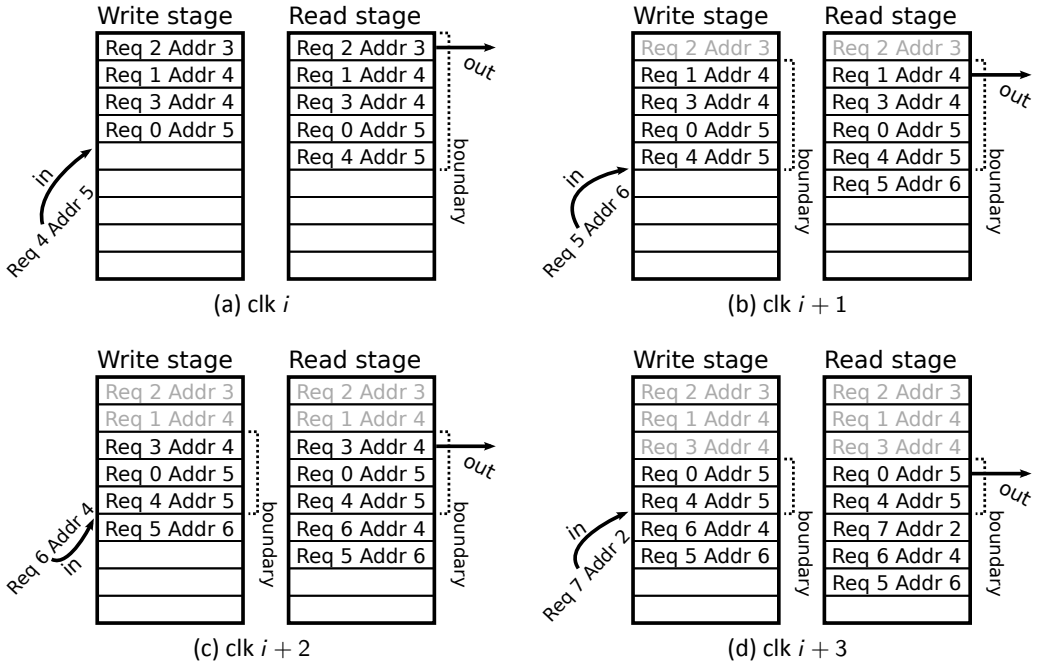


Figure 4.1.2: An example of the Smallest Address First, Static Boundary policy for 4 clock cycles and several requests that are sorted by their address. The grayed out requests are those that have been removed and sent to the Back-End. (a) request 4 is inserted and request 2 is removed and the boundary is set. (b) request 5 is inserted outside the boundary and request 1 is removed. (c) request 6 is inserted outside the boundary and request 3 is removed. (d) request 7 is inserted outside the boundary and request 0 is removed.

4.1.3 Smallest Address First, Dynamic Boundary

With the Smallest Address First, Dynamic Boundary the request with the smallest address in the sorting array is sent to the Back-End as first, just like the previous policy. The difference is that in this case the boundary is not static but dynamic, the boundary can change after it is set. This will happen when a request is entering the sorting array with a row and bank address that is equal or larger then the row and bank address of the last read request. Figure 4.1.3 shows an example.

There is no problem sorting the new requests inside the boundary, the request with the smallest row and bank address inside the sorting array will still be read next and the amount of row changes is not increased. There is even a possibility that extra requests with the same row and bank address can be packed together, minimizing the stalls even more compared to the previous policy.

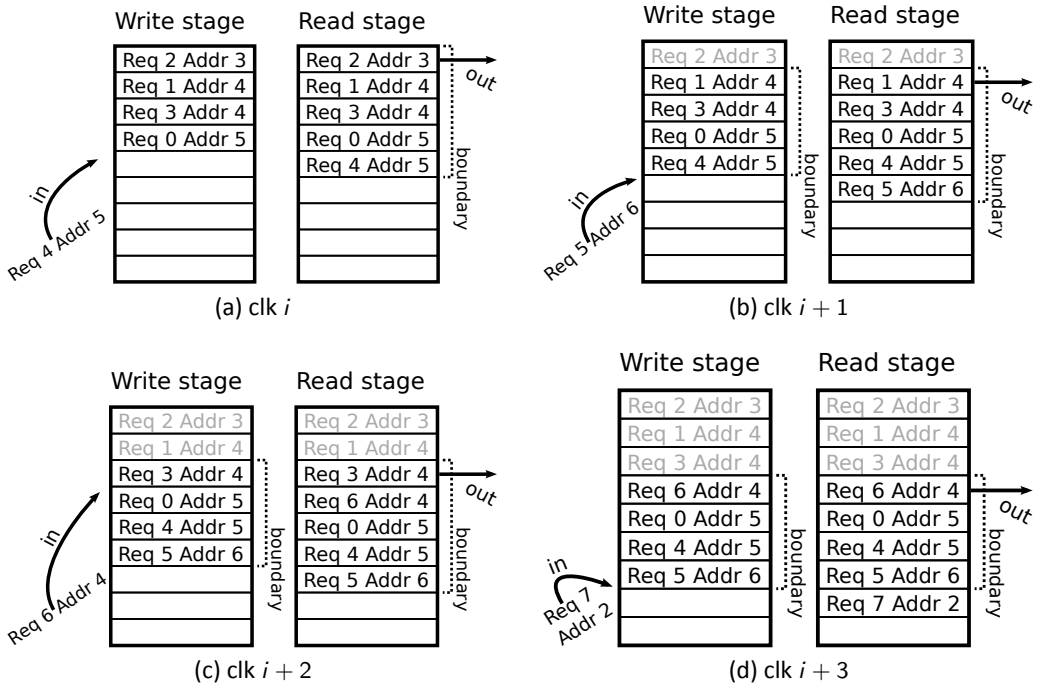


Figure 4.1.3: An example of the Smallest Address First, Dynamic Boundary policy for 4 clock cycles and several requests that are sorted by their address. The grayed out requests are those that have been removed and sent to the Back-End. (a) request 4 is inserted and request 2 is removed and the boundary is set. (b) request 5 is inserted inside the boundary and request 1 is removed. (c) request 6 is inserted inside the boundary and request 3 is removed. (d) request 7 is inserted outside the boundary and request 6 is removed.

4.1.4 Largest Block First, Static Boundary

With this policy the largest block of requests with the same row and bank address is sent to the Back-End as first, instead of the requests with the smallest row and bank address. In this policy the boundary is static, just like the Smallest Address First with Static Boundary in subsection 4.1.2. No new requests are added inside the boundary once a request has been sent to the Back-End, but only outside the boundary. In figure 4.1.4 the policy is explained with an example.

Once a block of requests is chosen to read, this block is processed entirely before a new block is searched inside the boundary. This means that it is not necessary to have a boundary at all, without having extra stalls. However, the chance is that requests that do not belong to a block are never processed when the data buffer is able to allocate all new requests. A boundary is therefore also used for this policy, to make sure that there will be no starvation at the user side.

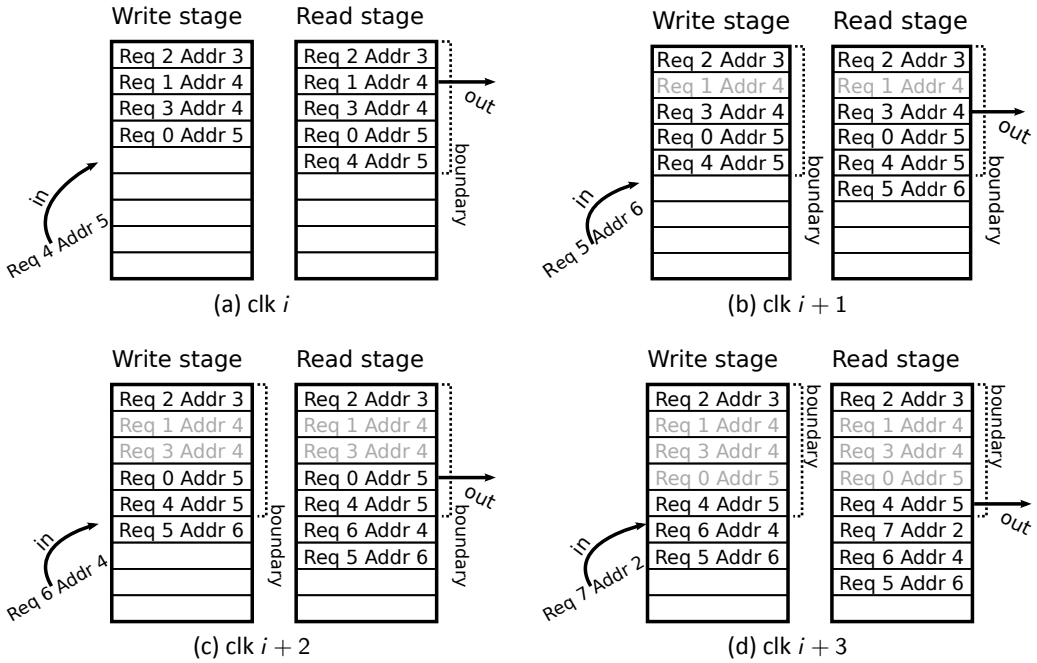


Figure 4.1.4: An example of the Largest Block First, Static Boundary policy for 4 clock cycles and several requests that are sorted by their address. The grayed out requests are those that have been removed and sent to the Back-End. (a) request 4 is inserted and request 1 is removed and the boundary is set. (b) request 5 is inserted outside the boundary and request 3 is removed. (c) request 6 is inserted outside the boundary and request 0 is removed. (d) request 7 is inserted outside the boundary and request 4 is removed.

4.1.5 Largest Block First, Dynamic Boundary

In the Largest Block First, Dynamic Boundary policy the largest block of requests with the same row and bank address is sent to the Back-End as first, just like the previous policy. The boundary is however changed to a dynamic one, where the it can be altered after setting it for the first time. This gives the opportunity to gather more requests for each block of requests to reduce the amount of stalls. A new request is accepted inside the boundary when there is already a block in the boundary with requests that have the same row and bank address. This is visualized with an example in 4.1.5. A request that is not accepted inside the boundary is still sorted, but outside the boundary.

The starvation problem, that one or more requests are not being processed, is remedied by allowing only the requests with a row and bank address that are already present in the boundary. Once a block of requests with a certain row and bank address is processed there are no new requests allowed inside the boundary that have that row and bank address.

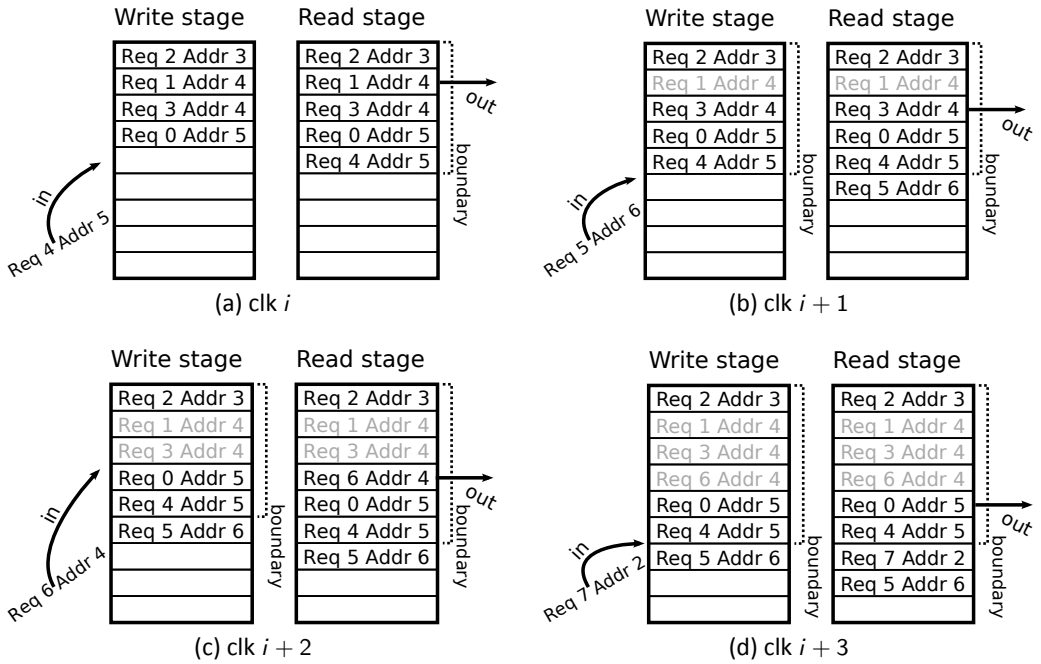


Figure 4.1.5: An example of the Largest Block First, Dynamic Boundary policy for 4 clock cycles and several requests that are sorted by their address. The grayed out requests are those that have been removed and sent to the Back-End. (a) request 4 is inserted and request 1 is removed and the boundary is set. (b) request 5 is inserted outside the boundary and request 3 is removed. (c) request 6 is inserted inside the boundary and request 6 is removed. (d) request 7 is inserted outside the boundary and request 0 is removed.

4.1.6 Sorting Array per Bank

There is no guarantee that when a block of requests of a certain row is processed, the next block of requests are located in an other bank when one of the policies of the previous subsections is used. The Sorting Array per Bank described in this section is an extension of one of the previous policies by using a sorting array with a reordering policy for each bank. The reordering policy that is used for each bank can be any other policy. Extra bandwidth can be gained when a policy can make sure that a new bank is selected when a block of requests is done and a request with a new row has to be selected.

A sorting array for each bank, an input arbiter for the writing, and an output arbiter for the reading is used to make this possible. A high level block diagram of this is depicted in figure 4.1.6. Each sorting array can have any policy implemented as described in the previous sections, the extra bandwidth is gained by using the output arbiter. This arbiter decides which bank has to be read based on information the sorting arrays are providing. The arbiter will select a new bank when a sorting array would output a request with a different row then its previous request. This will make sure that, as long as there are requests available in all sorting arrays, a different bank is chosen when a request with a new row has to be read.

The writing arbiter has the function to route the incoming request to the correct

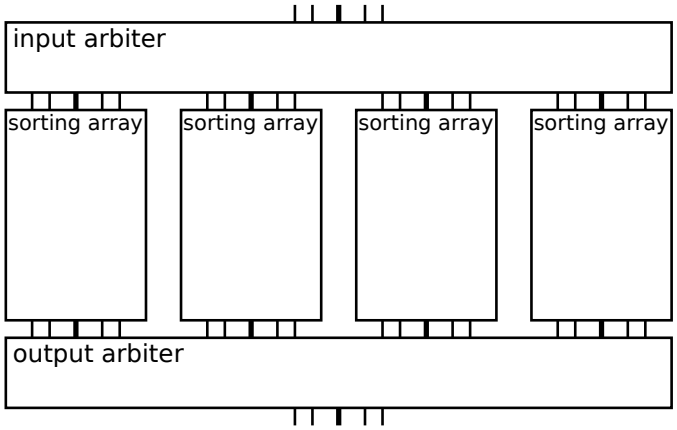


Figure 4.1.6: The maximum theoretical bandwidth can be approached closer by using a separate sorting array with a reordering policy for each bank and an input- and output-arbiter.

sorting array and pass the acknowledgments back when needed. The arbiter will only signal an overflow when all sorting arrays are full and a write is coming in, in other cases the write is just not acknowledged. The arbiter is also responsible for passing on all necessary inconsistency protection signals to all sorting arrays. It is not possible to make sure that requests from different sorting arrays are processed in the correct order when there is an overlap between them. The incoming request that would overlap with a request in a different sorting array has to wait until the overlapping request is processed before it can be acknowledged. More of the inconsistency can be read in section 4.3.

The policy that is implemented is the Smallest Address First with Dynamic Boundary and its implementation is described in section 6.1.

4.2 Data Buffer Allocation

In this section the data buffer allocation is described. There is a possibility that a request is not being processed for a while when requests are sorted in the sorting array. In that time the space in the data buffer that was assigned for that request cannot be used by another request. This means that there is a check needed to make sure that a location in the data buffer is not used when an allocation is requested.

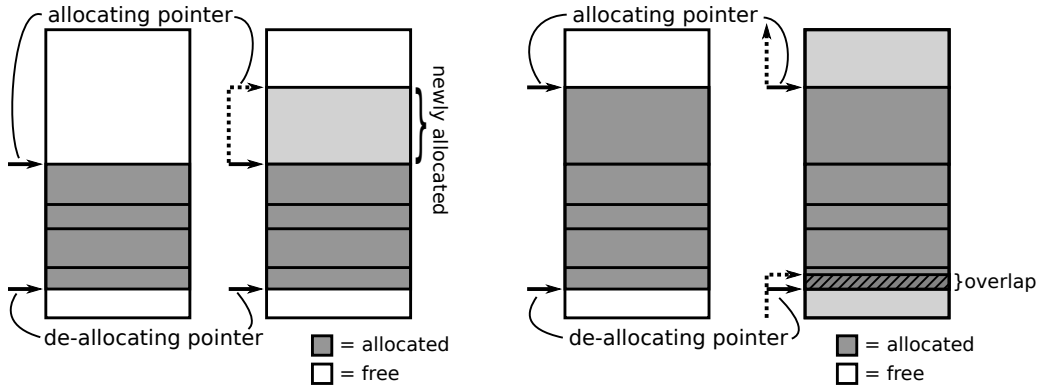
One of the properties of the memory controller is that each request does not have to be of equal size. So the size of the request plays an important role in the allocation and de-allocation process.

A simple way to protect the used space in the data buffer is given in section 4.2.1. This allocating is done by assuming a non-sorted set of requests, something that is not realistic for this memory controller. However, this allocation policy can also be used for ordered sets with some minor edits, as described in section 4.2.2. In section 4.2.3 an allocation policy is described where the allocation is more advanced while the de-allocation is fairly easy. A more advanced allocation and de-allocation structure, with some similarities with the software based allocation, is given in section 4.2.4.

4.2.1 Simple Allocation for a Non-Sorted Set of Requests

Keeping track of the upper and lower bound of the allocated space in the data buffer is enough in the case that the requests are not sorted and thus no optimization is used. There is no fragmentation taking place when de-allocating because every request is processed in the same order as they were allocated. The boundaries of the allocated space in the data buffer are represented by two simple pointers and these pointers have to be updated when allocating or de-allocating.

The upper boundary pointer can be updated by adding the size of the request to its current value for allocating. If the new pointer has not passed the lower boundary pointer, the new value can be accepted and the allocation can be acknowledged. An example of this is given in figure 4.2.1a. The previous value of the upper boundary pointer is returned as the starting data location for the request when the allocation is accepted. The allocation will not be acknowledged when there is not enough free space in the data buffer. The upper boundary pointer passes the lower boundary pointer as seen in figure 4.2.1b and the request has to wait until enough data blocks are de-allocated.



(a) Enough free space (b) Not enough free space

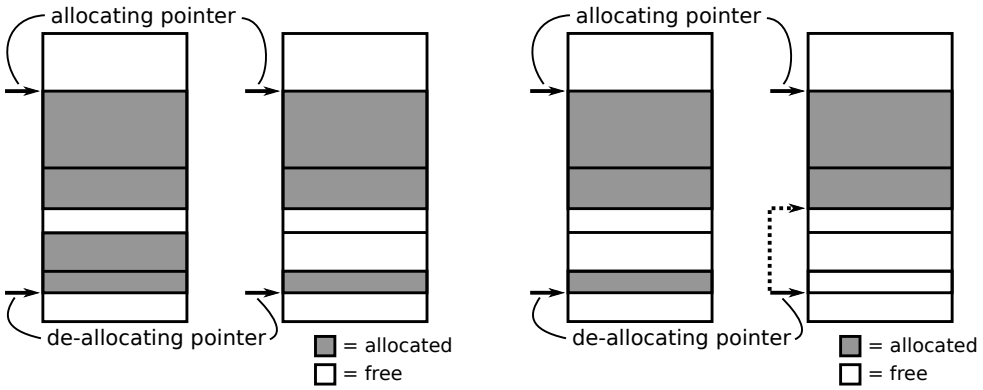
Figure 4.2.1: Simple allocation for non-sorted requests

De-allocating a data block in the data buffer needs just a simple addition of the lower boundary pointer with the size that has to be de-allocated. Allocating and de-allocating is thus very simple, there are just two pointers needed. There is no need to keep track of all used locations in the data buffer. However, request reordering makes this allocation policy not suitable.

4.2.2 Simple Allocation for a Sorted Set of Requests

Allocating a new data block in the data buffer can be done in the same way as in the previous subsection when an ordered set of requests is used. There is enough space for the data block when the upper boundary pointer can be increased by the size of the request without passing the lower boundary pointer. For the de-allocation however it is not that simple anymore, this because the de-allocation is done out of order. This means that the lower boundary pointer cannot simply being updated by subtracting the

size when de-allocating. The lower boundary pointer cannot be updated when a data block is de-allocated and this block is not at the edge, as seen in figure 4.2.2a. As long as there is used space at the location of the lower boundary pointer this pointer cannot be updated.



(a) No update for the de-allocation pointer is possible.

(b) The de-allocation pointer can be updated to a new location in the data buffer.

Figure 4.2.2: De-allocating with the Simple Allocation for Sorted Requests policy.

When a data block at the edge of the lower boundary pointer is de-allocated, there is some updating to do. This updating is not simply done by adding the size of the last request because several other requests could have been de-allocated in advance. The new value of the pointer is therefore depending on the whole de-allocated space connected to the data block that is being de-allocated. This is made visible in figure 4.2.2b.

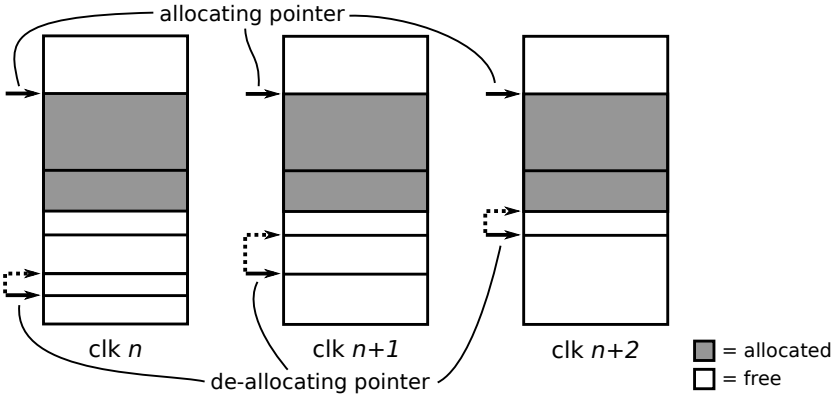


Figure 4.2.3: The de-allocation pointer is updated each clock cycle to the next free space in the data buffer.

One way to update the pointer is by hopping through the data buffer from previous de-allocated data blocks to the next de-allocated data block. As visualized in figure 4.2.3 this will take some clock cycles to complete, especially when there are a lot of free data blocks to hop through. Besides the extra amount of clock cycles there is also a need for extra hardware. Each position in the data buffer has to contain the size of the data block

and a bit if the data block has been de-allocated or not.

Another way is to count how many free locations are ahead of the pointer and add this to the pointer value. This counting can be done in a parallel way and updating of the pointer could be done in one clock cycle. Only the free locations until the first occupied location can be counted however, this makes the parallel counting harder and more hardware costly. There are several ways how this could be implemented, but for just updating a pointer it is not the best way.

4.2.3 More Advance Allocating and Simple De-Allocating

It is also possible to have a more advanced allocating and a simple de-allocating. All allocation policies described until now used an inspection of the next free space in the data buffer for allocating. The same holds for the policy described in this subsection. A request can be allocated if the free space located at the allocating pointer is large enough. The request is not being allocated when this is not the case, even when there is another block of free locations at a different location in the data buffer that actually is large enough. This is because that block of free locations is not being considered. This means that there is no need to mark every location in the data block as used, but only the first location. There is no location in the data block in the data buffer that can be allocated twice, this because the allocating pointer is never allowed to pass a location that is already allocated. An illustration of the allocation can be seen in figure 4.2.4.

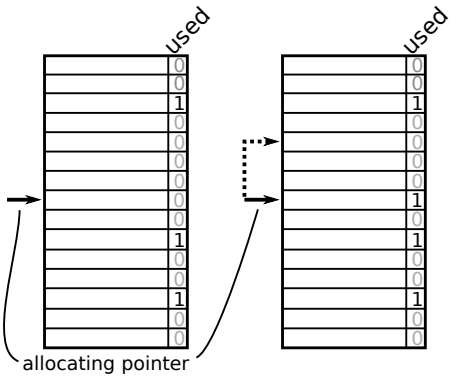


Figure 4.2.4: The location where the allocation pointer is pointing to is set to used when allocating. The pointer is then moved to the new location in the data buffer. In this example a block of three locations is allocated, but only the first location is set to used.

Just one bit has to be cleared when a whole data block has to be de-allocated, the one in the beginning of the block that was set when the data block was allocated. Every location of the data block can be allocated again when this bit is cleared, there is no need to take other blocks into consideration for the de-allocation process. The advantage of this, is that there is no pointer to be updated.

There are some challenges in the checking if a data block can fit in the next free space in the data buffer. The size of just the first free space has to be counted, not every free location in the data buffer. This can be done by first truncating all free locations after the first occupied one. A parallel counter can then be used to count the number of free

locations. Both the truncating and counting will cost a lot of hardware when the whole data buffer is considered. By dividing the data buffer into segments this can be made easier. Each segment will signal if the whole segment is free or not, as can be seen in figure 4.2.5. In that way only the next free segments have to be counted and not the whole data buffer. An or-tree structure can be used in each segment for determining if there is an allocated location in the segment.

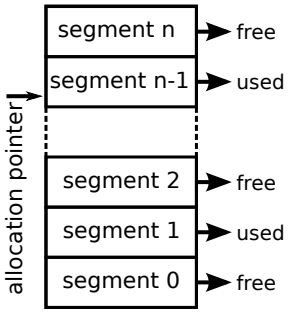


Figure 4.2.5: The data buffer divided in segments where each segment signals if the whole segment is free or used.

Instead of only using the segments to determine if there is enough space available, the segment where the allocation pointer is located is also used. It is known that there are no allocated locations from the allocating pointer until the last location of the segment where the allocating pointer is currently in. This because it is not possible to allocate in a new segment if there is a location already allocated. The amount of free locations in a segment can easily be determined by using this property. The last n-bits of the allocation pointer address have to be inverted when the segments of the data buffer are of size 2^n. The sum of all free segments shifted n times can then be added, which will give the maximum size that is free in the data buffer for the allocation, as can be seen in figure 4.2.6.

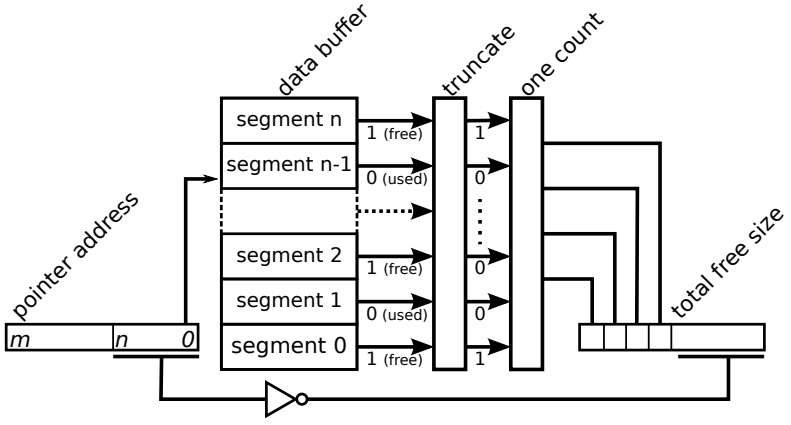


Figure 4.2.6: Determining the size of the next free space by using the position of the allocation pointer in the segment and the upcoming free segments.

As stated before, the disadvantage of this approach is that only the next free space in

the data buffer is taken into account. An allocation can not be acknowledged when there is a request with a size that is larger than the next free space, even when there is a block of free locations in the data buffer that would have been large enough. However, there is no need to update every location of a data block when it is allocated or de-allocated.

4.2.4 Advanced Way of Allocating

A more advanced allocating policy is considering all free spots in the whole data buffer. Every location of the data buffer has to be marked as allocated instead of just the first one, as done in section 4.2.3. As long as there is a space available that is large enough allocation can taken place, no matter where the space is located in the data buffer.

With a software based allocator like malloc, a linked-list is used for keeping track of all free blocks in the memory or data buffer. There is no need for marking every single location when a data block is allocated. The linked list has to be searched when a new data block has to be allocated. Every item of the linked list points to a location in the data buffer where a free block is starting. This list is build up while allocating, an header is added before the allocatable data block stating the size of the data block and a pointer to the next free block. After a free space is found that is large enough the location of the space is returned to the user and the linked list is updated. A reference to a free data block is added to the linked list or merged with other free data blocks in the linked list when a data block is de-allocated [4]. This way of allocating and de-allocating is not the fastest and efficient way for an hardware implementation.

To find out if there is a block that is large enough in hardware an or-tree can be used as done in [8]. For each level l of the or-tree it can be read out if there is a space available of size 2^l, as illustrated in figure 4.2.7. One can choose not to mark each location in the data buffer, but a small segment of locations. When using segments of size m there is a change that a maximum size of $m-1$ is allocated and not used, but the amount of or-gates is reduced.

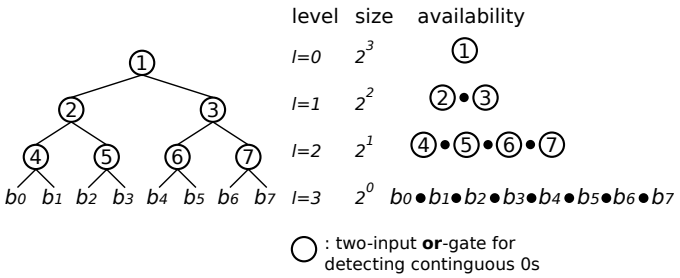


Figure 4.2.7: The or-tree that is used for determining a free block of sufficient size. At level l the availability will say if a size of 2^l is available for allocating.

With a buffer size of 2^N, the availability of level l of the or-tree has to be checked to read out if there is a block available of size $2^N - l$. The availability can be checked by using a 1 for an allocated and a 0 for a free location in the data buffer. Using a big and-gate for each level, the availability will be zero when at that level there is at least one sub-tree stating that there is enough space available.

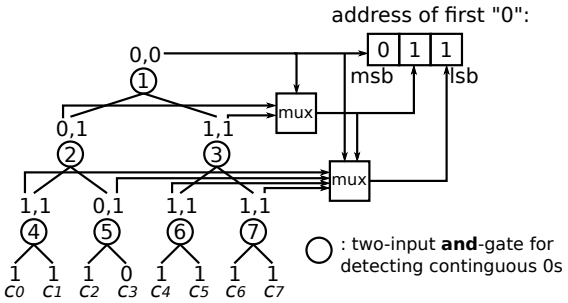


Figure 4.2.8: Mapping the and-gate tree in hardware

The address of the location where the free block is starting can be found using a ’non-backtracking binary search algorithm’ implemented with an and-tree and some multiplexers [8]. An illustration of this can be seen in figure 4.2.8. Every location in the allocated data block has to be marked after detecting if there is enough space and from which address this data block is starting from. A bit-flipper is used to flip the correct bits according to the data block address and size. How this bit flipper is implemented is left for the interested reader in [8].

The allocation policy that is implemented is the More Advance Allocating and Simple De-Allocating and its implementation is described in section 6.2.

4.3 Memory Consistency

The memory inconsistencies introduced by sorting the requests are described in this section. Correct data is being read and written when the requests are not sorted in the memory controller, in that case there is no need for a consistency check. All data that is written to the same location in the memory has to be written to that location in that order. A possible memory inconsistency is then taken care of by the programmer, compiler or processor [1, 11]. There is a possibility that this is not the case anymore when the requests are sorted, the memory state can become inconsistent. The memory controller can create inconsistencies because the sorting is not exposed to the programmer.

The inconsistencies are created when the requests are sorted that will write or read entirely or partially at the same location in the memory, the requests are overlapping. In section 4.3.1 the types of overlaps are described that the requests can have. The different situations where those overlaps are creating inconsistencies are described in section 4.3.2 and in section 4.3.3 the solution to the inconsistency problem is described. In section 4.3.4 it is described if it is possible to use the inconsistency check for extra optimizations.

4.3.1 Types of Overlaps

One of the properties of the memory controller is that it can process requests of different sizes and different locations. The memory controller does not work on word level, entire requests with just a start address and size are being sorted. These requests are however aligned at word level, meaning that several types of overlaps of the requests are possible.

- **No overlap.** The locations where the requests have to write or read to are further apart from each other then the size of the request. There is no common location and the requests are thus not overlapping. This can be seen in figure 4.3.1.

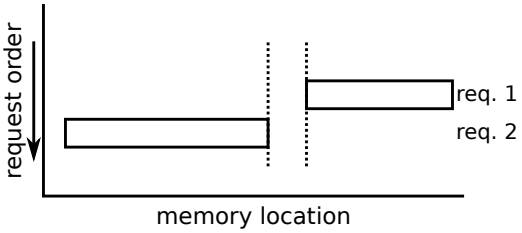
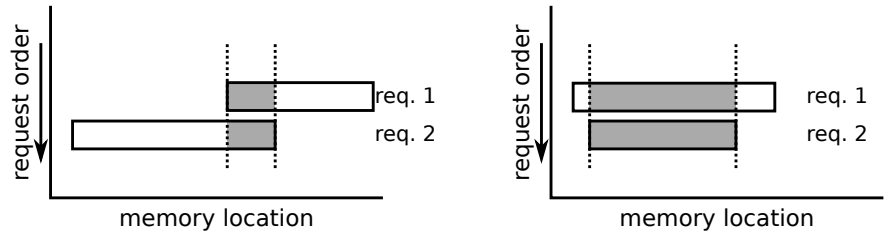


Figure 4.3.1: Requests are not overlapping

- **Partial overlap.** Some locations of the requests are read written by both requests. An example of such an overlap is visualized in figure 4.3.2a and figure 4.3.2b. A part of the large request is being overlapped because the small request of figure 4.3.2b is issued as last. There was a full overlap when it was the other way around, when the small request was issued before the large request.



(a) Edges of both requests are overlapping

(b) The middle part of request 1 is overlapped by request 2

Figure 4.3.2: Requests are partially overlapping

- **Full overlap.** Every location of a request has to read or write is also read or written by another request. As visualized in figure 4.3.3 one of the requests is fully overlapped by the other request.

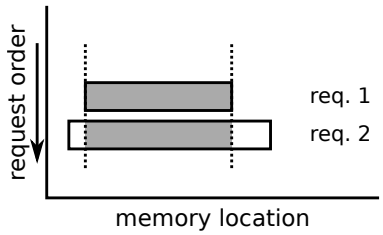


Figure 4.3.3: Request 1 is fully overlapped by request 2

These overlaps can be detected using the address and size of both requests. To see if there is an overlap just two situations have to be considered. The first is when the

value of the address of request 1 is less or equal as the value of the address of request 2 added with the size of request 2, like equation 4.3.1a. The second situation is similar as the first, when the value of the address of request 2 is less or equal as the value of the address of request 2 added with the size of request 2, like equation 4.3.1b. At least a part of both requests are overlapping when both situations are true, so when *overlap* in equation 4.3.1c is set to 1.

$$s_0 = address_{req1} \leq (address_{req2} + size_{req2}) \tag{4.3.1a}$$
$$s_1 = address_{req2} \leq (address_{req1} + size_{req1}) \tag{4.3.1b}$$
$$overlap = s_0 \; and \; s_1 \tag{4.3.1c}$$

To detect if the value of the address from an inserted request is less then the one from the current request, the same situations can be used. The inserting request address is less then the current request address when equation 4.3.2 is true when $req1$ is the request currently present in the sorting array and $req2$ the inserted request. For the sorting the same resources as for the overlap detection could therefore be used with an overhead of just one inverter and and-gate.

$$not(s_0) \; and \; s_1 \tag{4.3.2}$$

4.3.2 Situations where Inconsistencies Occur

As stated before the inconsistency problem is occurring when overlapping requests are sorted. In this subsection all situations are given where data inconsistencies can occur and where not. It is obvious that for non-overlapping situations no inconsistency is possible, so this situation is not further discussed. Four situations can be seen in case of the overlapping requests:

- **Read after read.** In this situation two overlapping read requests are sorted in a different order then intended. Both requests are reading from the same location in the memory, but they are not altering the read data. This means that it does not matter when both requests are processed, they will still read the same values and **no inconsistencies** occur as seen in figure 4.3.4.

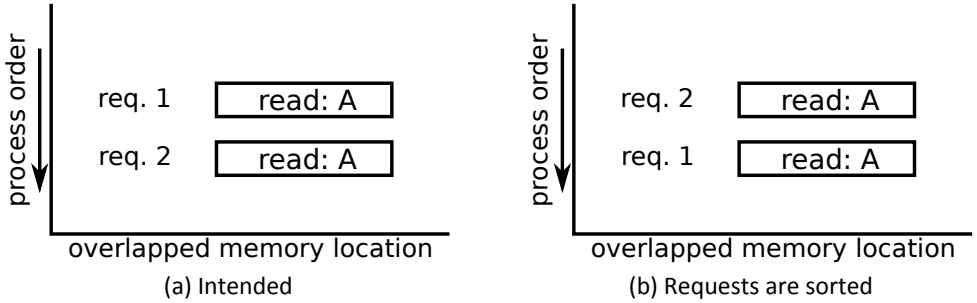


Figure 4.3.4: Two overlapping read requests are sorted differently then intended.

- **Read after write.** Another inconsistency is created when a read is issued after an overlapping write request, instead of before. The value at the overlapping locations is altered before the read in this situation, as can been seen in figure 4.3.5. The read value is then **not correct**.

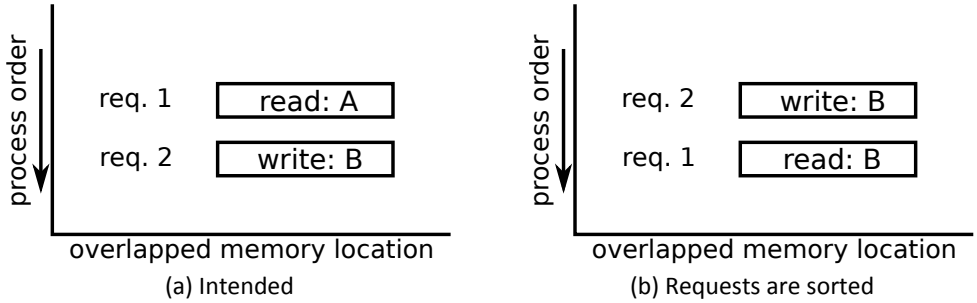


Figure 4.3.5: A read request is sorted after an overlapping write request.

- **Write after read.** In the situation where a write is issued after a read, instead of before, there is also an inconsistency. The overlapping locations are supposed to be altered before the read request is processed, but because of the sorting this is not the case. The write request is altering the data to late and the read data is **incorrect**. This is visualized in figure 4.3.6.

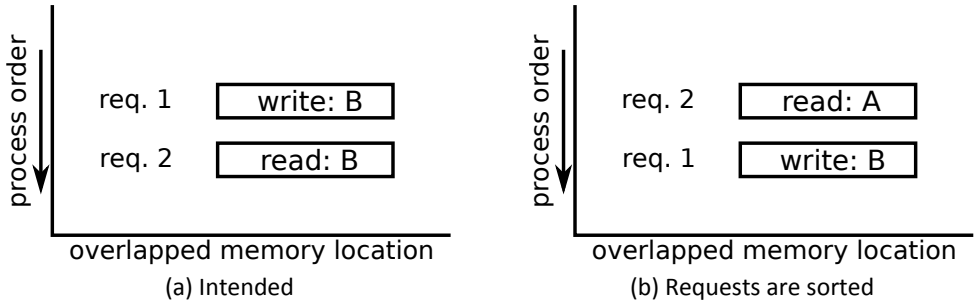


Figure 4.3.6: A write request is sorted after an overlapping read request.

- **Write after write.** In the last situation there are two overlapping write requests sorted in a different order then intended. Both requests are altering the overlapping locations in the memory, but at the end the wrong values are stored. As visualized in figure 4.3.7 the last write is altering the data at the overlapping location in the memory as last which can lead to a **wrong value** when the requests are sorted.

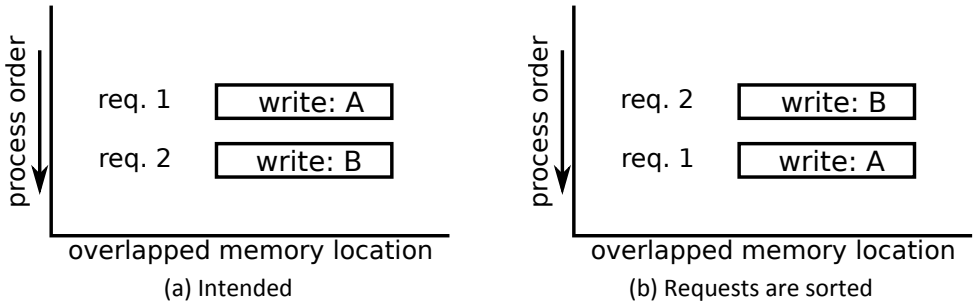


Figure 4.3.7: Two overlapping write requests are sorted differently then intended.

All overlapping requests are causing memory inconsistencies with exception of the first situation, the read after read. In all other situations the data in the memory at the overlapping locations is altered at the wrong time by write requests.

4.3.3 Solutions for the Inconsistency Problem

In the previous subsection the situations were described where the sorting of requests is causing memory inconsistencies. In this subsection the solutions for these problems are described with a short solution for each situation as first and a complete solution after that.

- **Read after read.** Two overlapping read requests are not causing any problems as stated in the previous subsection.
- **Read after write.** When an overlapping read request is sorted after a write request the read data is not correct. The write request should not be sorted before the read request when it has an overlap. In that way the original value is read before it is overwritten by the write request.
- **Write after read.** Also in this situation the read data is not correct. The read request that is intended to be processed after the overlapping write request should therefore not be sorted. The read request is then processed after the write request and the intended data is read.
- **Write after write.** The last write request that is entering the sorting array should not be sorted, just like in the previous situations.

There will be no memory inconsistency when only read requests are being sorted. When there is a write request however, the inconsistencies can occur. With that in mind a general solution is to process all read request in a sorted way until a write request should be sorted. All incoming requests are hold until the sorting array is empty and all read requests are processed. All new requests are then processed in order until a read request is entered and all write requests are issued to the Back-End. The new read requests can then be sorted again without interfering with a write request.

The bus is unnecessary stalled until the sorting array is empty with this simple solution. The write requests have to wait until the sorting array is empty, even when there is no possible overlap at all. The positive side is that there is no need for a check at each location in the sorting array if there is an overlap or not.

A more suitable solution is only not to sort the request that is causing a problem. All sorted elements in the sorting array have to check if there is an overlap when a request is entered. The request that is entering the sorting array has to be sorted in such a way that it is processed after the request that has the overlap. This will need a significant amount of logic, every element in the sorting array has to be able to compare its address and size with the request entering the array. On the other hand, it is possible to process almost all requests in the optimal way the sorting policy is offering and the bus is not unnecessary stalled.

4.3.4 Using the Inconsistency Check for Possible Extra Optimizations

It is possible to have extra optimizations when checking for inconsistencies. A part of a request can be deleted when two write requests are overlapping. The last value that is written to the memory, before a read is taken place, is the value that matters. All other overlapped write requests can be changed in such a way that they do not write to the overlapped location anymore. Depending on the kind of the overlap some optimizations are possible when two write requests are overlapping:

- **Overlap at edge**. The changing of a request can be done by altering the address or size of the request in case of an overlap at an edge (see figure 4.3.2a at page 38). This is easy to do as only one value of the request has to be changed.

- **Overlap in the middle**.The request has to be converted into two smaller requests when the overlapped request lies in the middle, as shown in 4.3.2b at page 38. An extra request has to be created besides altering the original request with a new size or address.

- **Fully overlapped**. The overlapped request can be deleted entirely when a full overlap is the case (see figure 4.3.3 at page 38), no request will ever read the value of that overlapped write request. Deleting a request is easy to do, the new request is inserted at the place of the overlapped request.

These optimizations will clearly give an extra optimization because not all data has to be processed.

These optimizations are however probably not that beneficial. Extra hardware is needed to detect the different kind of overlaps and alter, delete or create requests. The data location of the request needs also to be changed when the starting address is changed for example. The allocated space used by the request in the data buffer should also be altered, the data buffer should be able to de-allocate a part of the space of the request that is not used anymore. All this extra hardware is needed when an application is writing multiple times at the same location without reading the value whiles writing, something that could have been taken care of by the programmer or compiler.

Requests can also be split in case there is a different overlap then two writes. The new parts of the requests could then be reordered separately. This is however not beneficial to do. Besides the extra space that is taken by the new request, there is also no performance gain. For example: there is an extra stall when an overlapping read request is sorted after a write request. One could state that there is a performance gain by sorting the non-overlapping part of the read request in front of the write request. However, the overlapping read request part is always sorted after the write request to prevent a memory inconsistency. That means that the write to read delay will never disappear and no extra optimization is gained.

Non of the extra optimizations is implemented because of the extra hardware which gives little to no benefits.

5 Preliminary Analysis

In this chapter the preliminary analysis that is used to decide what reordering policy has to be implemented is described. A fair comparison can be made on which policy would be the best to implement by using a software based simulator together with an estimation on the hardware costs. More time is spend when all policies are implemented in hardware and the comparison is made afterwards.

At first an introduction is given for the software based simulations in section 5.1. In this section the assumptions are described that are used for the simulations and how the simulation model is approaching reality. The results of the simulations and the estimated hardware costs for the several policies are described in 5.2.

5.1 Software Simulation Solutions

Two software based simulators are used for the preliminary analysis and they are described in this section. These simulators will estimate the amount of clock cycles needed to process an input file with requests and give also some extra information. An example of this extra information is the amount of row changes that were made in the same bank while reading or writing from that bank.

The first software simulator is described in section 5.1.1 while a more advanced simulator is described in section 5.1.2. The first four reorder policies from section 4.1 are implemented in both simulators. The Sorting Array per Bank policy described in 4.1.6 is only implemented in the advanced simulator because this simulator has a more modular setup and a better approach on reality regarding the sorting array.

Both simulators accept arguments like where the row and bank address bits are starting, how long the Back-End should wait before reading the first request of the sorting array, how often a request is inserted, and how large the sorting array is. The simulators will read the memory requests from an external file generated by another program, described in section A.1. Different algorithms and applications can therefore easily be tested. These generation programs generate a file with the memory requests for a 3D-FFT method, Conjugate Gradient method, or random read and writes.

5.1.1 Basic Software Simulation

The basic software simulator developed by us is described in this section. A simple simulation model of the Bridge and Back-End is used for the simulation of the reordering policies. For this model a few assumptions are made by looking at the request-level of the hardware simulations of the Bridge and Back-End. This means that the model decides how many clock cycles are needed for a request based on the previous request. The following assumptions have been made for the basic software simulator:

- Inserting a read request in the request buffer costs one clock cycle.
- A write requests for n words costs n cycles to be inserted. That a write request takes more time then a read request is because the corresponding data has to be transferred to the data buffer.
- Processing a request costs at least n clock cycles, where n is the size in words of the request that is being processed.
- Extra delay is added depending on the last request:

	same bank, different row	different bank or same row
write after a write	7 cycles	0 cycles
read after a read	6 cycles	0 cycles
read after a write	11 cycles	4 cycles
write after a read	2 cycles	0 cycles
first request	5 cycles	

Table 5.1: The extra delays that are dependent on the previous request for the basic software simulator

- It is necessary to wait a few extra clock cycles when there is a read, then a short read (4 words or less) in another bank and after that a read from the first bank but in a different row. This is because the closing and opening of the bank takes more time then the transfer in the other bank. However, it is assumed that this will not happen and no extra clock cycles are accounted in this simulator.
- The simulator assumes that requests can be read and written to the sorting array at the same time, just like the proposed solution.
- A full duplex system is assumed for the data buffer, data can be read and written to the data buffer at the same time by the Front-End and Back-End.
- The software simulator assumes an unlimited data buffer, so the reordering is not influenced by this.
- DRAM refreshes are not taken into account.

The basic software simulator will simulate all policies and the base line (no reordering done) at the same time in just a few functions. These functions are running in parallel to save simulation time. The results are printed when all policies are done simulating.

5.1.2 Advanced Software Simulation

A model that lies at a deeper level then the request level of the basic simulator is used for the advanced software simulator described in this section. This advanced software simulator is developed by us, just like the basic software simulator. A more detailed model is made by looking at the commands that have to be send to the memory to start

reading or writing instead of just looking at the previous request. Which commands have to be send and at what time is decided in a function that tries to replicate the behavior of the OpenCores Back-End. This means that new commands are issued as soon as possible and in such a way that the data flow is not interrupted, just like the Back-End that is used in the proposed solution (see section 3.3).

Some assumptions had to be made to make the model not too complex, others to make the model more realistic. The following assumptions are made:

- Inserting a read request in the request buffer costs one clock cycle.
- A write requests for n words costs n cycles to be inserted. That a write request takes more time then a read request is because the corresponding data has to be transferred to the data buffer.
- Processing a request costs at least n clock cycles, where n is the size in words of the request that is being processed.
- A request can be processed when the correct row in the bank is activated and the read or write command is issued.
- The following latencies are used for the commands:

command	latency	comment
activate	3 cycles	
precharge	3 cycles	
read	2 cycles	
write	2 cycles	
read-to-write	1 cycle	after last read
write-to-read	3 cycles	after last write
write-to-precharge	2 cycles	minimum time after write before precharge

Table 5.2: The latencies that the advanced simulator is using for each command to control the memory

- Only one command can be issued to the memory at each clock cycle.
- Commands can be issued in advance, but the reading or writing should start exactly after the latency time of the read or write command.
- The simulator assumes that requests can be read and written to the sorting array at the same time, just like the proposed solution.
- A full duplex system is assumed for the data buffer, data can be read and written to the data buffer at the same time by the Front-End and Back-End.
- The software simulator assumes an unlimited data buffer, so the reordering is not influenced by this.
- DRAM refreshes are not taken into account.

The Back-End function will, just like the hardware implementation, read several requests and try to precharge and activate the needed rows during data transfers. The read and write commands are issued when a row is activated and the current data transfer lasts for the amount of cycles of a write or read command. This will make sure that the data transfer will stay continuous and the model approaches reality. Besides the amount of clock cycles that are needed to process all requests, the amount of precharges and activates can also be counted because of the simulation at command-level.

A new policy is easily implemented in the software simulation because of its modular setup. The Back-End functions are not influenced by the sorting array and the other way around. This simulator will simulate just one policy each time, when another policy has to be simulated the program has to be started again. The amount of clock cycles, row changes in the same bank, amount of precharges, and the amount of activates are printed after fully simulating the policy.

5.2 Several Sorting Policies

In this section the simulation results and estimated hardware costs are given for the several policies. With these software results and estimated hardware costs a considered decision can be made on which reordering policy has to be implemented. The results of the simulations can be found in section 5.2.1 while the estimated hardware costs are discussed in section 5.2.2.

5.2.1 Simulation Results

In this subsection the results of the two software simulators are given and discussed. For each reordering policy the amount of clock cycles is simulated by both simulators and form the most important results. Less clock cycles needed to process all requests means a higher bandwidth of the DRAM. The amount of precharges are measured by the advanced simulator only and can give an explanation on how a certain speed-up is gained.

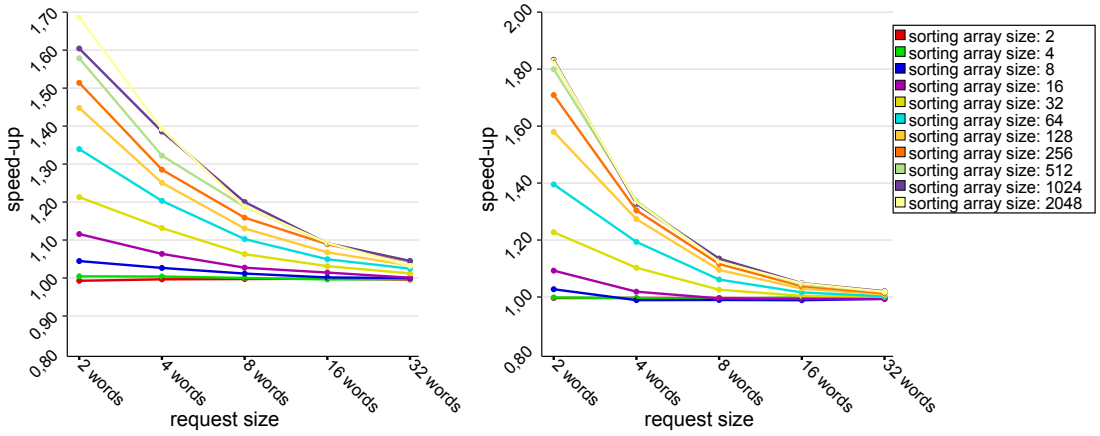
Also the amount of row changes in the same bank while processing from that bank are measured. These are the changes of rows that produce an immediate stall that could not be taken care of by scheduling the commands during a previous data transfer.

All graphs that are given in this section are made by normalizing the results with the base where no reordering is done, unless stated otherwise. A 3D-FFT algorithm is used to generate the necessary memory requests and is always using the same amount of data. This means that less requests are issued when requests are used with a larger size. Because less requests are issued, and thus a smaller amount of overhead is produced, this will influence the speed-ups in the results.

5.2.1.1 Results Smallest Address First, Static Boundary Policy

A speed-up is gained when looking at the results from the software simulation for the Smallest Address First, Static Boundary policy in figure 5.2.1 for both software simulators. This speed-up decreases for test-sets with a larger request size, independent of which

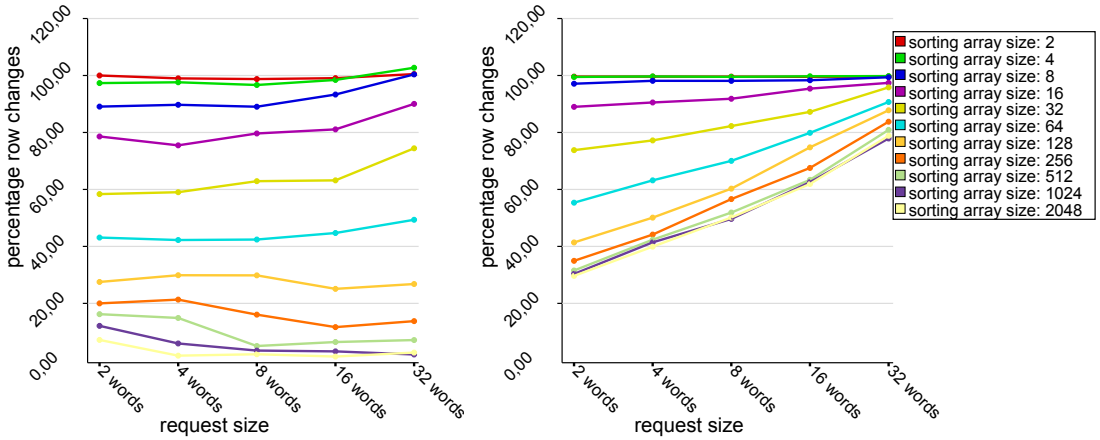
simulator is used. More data has to be transferred for each request and the overhead for each request that can be optimized stays the same.



(a) basic simulator (b) advanced simulator

Figure 5.2.1: The speed-up that is gained by using the Smallest Address First policy with a static boundary. The simulations are made with several sorting array and request sizes.

The speed-up is increasing when a larger sorting array size is used, but starts to saturate after a sorting array size of 128 elements. Both simulators show the saturation and decreased speed-ups, but the basic software simulator is more optimistic on how the speed-ups decreases with larger request sizes. An explanation of this can be that the overhead assumed for the basic software simulator is larger as the one in the more advanced software simulator. This would mean that the decreasing because of larger request sizes is lower in comparison with a smaller overhead.



(a) basic simulator (b) advanced simulator

Figure 5.2.2: The percentage of row changes in the same bank that is gained by using the Smallest Address First policy with a static boundary. The simulations are made with several sorting array and request sizes.

The amount of row changes in the same bank, while reading or writing in that bank, are decreasing as expected after calculating the speed-ups. This can be seen in figure 5.2.2. It is also visible that the amount of gain in the row changes is less when larger request sizes are used. A reason for this can be that banks are changed more frequently for the non-reordered case because of the larger requests.

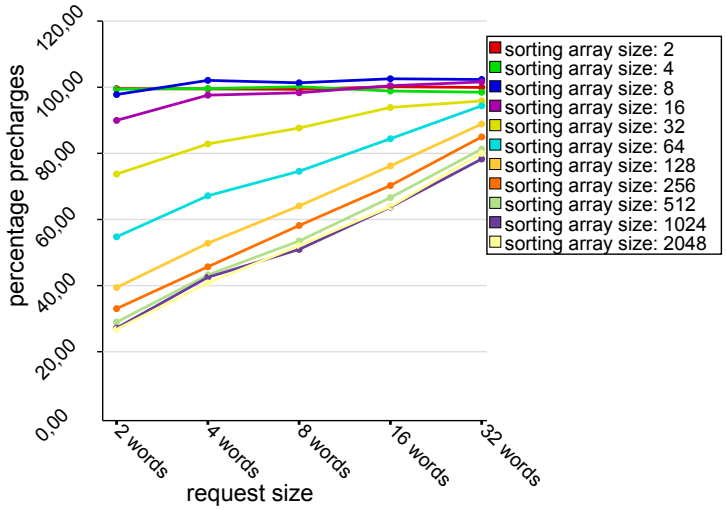


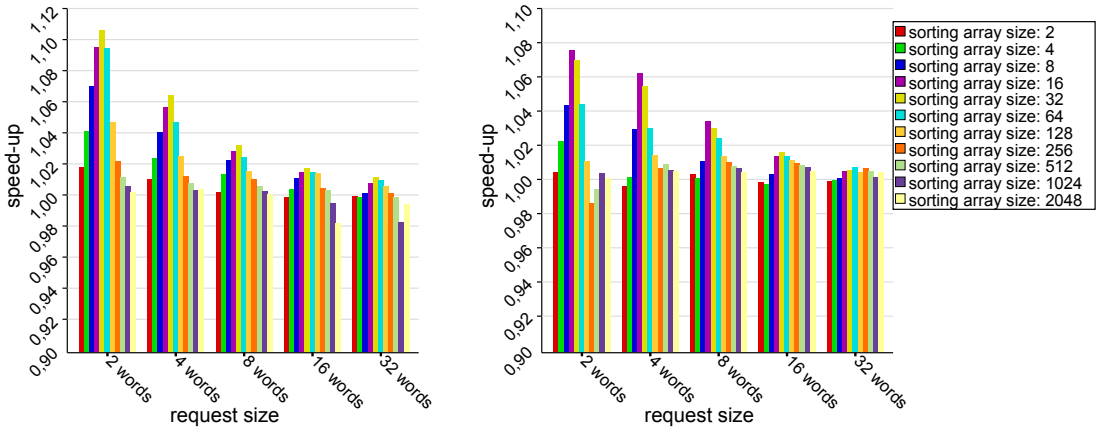
Figure 5.2.3: The percentage of precharges that is gained by using the Smallest Address First policy with a static boundary for the advanced software simulator. The simulations are made with several sorting array and request sizes.

The amount of precharges that are issued to the DRAM are also less when a Smallest Address First policy is used with a dynamic boundary. This can be seen in figure 5.2.3. The reduction of the precharges is high for smaller request sizes, but this decreases for smaller sorting array sizes and larger request sizes. As stated in section 5.1.2 the precharges can only be simulated by the more advanced software simulator.

5.2.1.2 Results Smallest Address First, Dynamic Boundary Policy

The dymanic boundary version of the Smallest Address First policy has a slight improvement in comparison to the static boundary as can be seen in figure 5.2.4. The results for the mid-range sorting array sizes, ones around $32 - 64$ elements, are showing the largest improvement. An extra improvement up to 10% is possible. A dynamic boundary is not that beneficial for extreme large sorting array sizes, sizes of 256 elements and larger, when looking at these results. Using a dynamic boundary instead of a static boundary does not decrease the speed-up in any case.

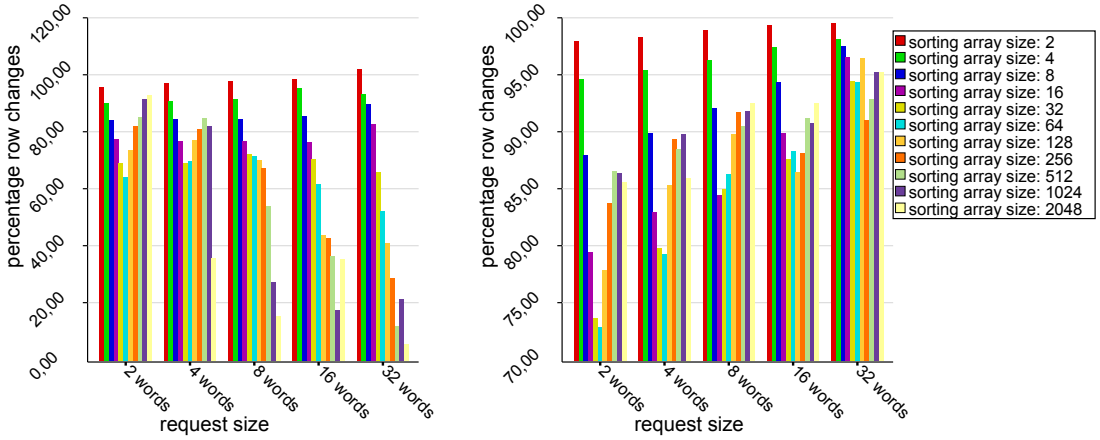
The improvement of the dynamic boundary is also visible when looking at the reduction of row changes in the same bank while processing in that bank. It shows that the dynamic boundary is beneficial for sorting array size around $32 - 64$ elements when the percentage of row changes of the Smallest Address First policy with a dynamic boundary is compared to the one with a static boundary. Around those sorting array sizes the amount of row changes in the same bank are more decreases as seen in figure 5.2.5



(a) basic simulator (b) advanced simulator

Figure 5.2.4: The extra speed-up that is gained by using the dynamic boundary in comparison with a static boundary when a Smallest Address First policy is used. For clarity a bar graph is used instead of line graph.

where the dynamic boundary results are normalized with the static boundary results. Also here it is seen that the dynamic boundary does not worsen the results compared to the static boundary.



(a) basic simulator (b) advanced simulator

Figure 5.2.5: The percentage of row changes in the same bank while processing from that bank. The results shown is the dynamic boundary in comparison with a static boundary for a Smallest Address First policy. For clarity a bar graph is used instead of line graph.

The amount of precharges that are issued to the DRAM are also less when a Smallest Address First policy is used with a dynamic boundary. This can be seen in figure 5.2.6. In comparison with a static boundary there is a improvement for the middle large sorting array sizes , other sorting array sizes do not show a large improvement.

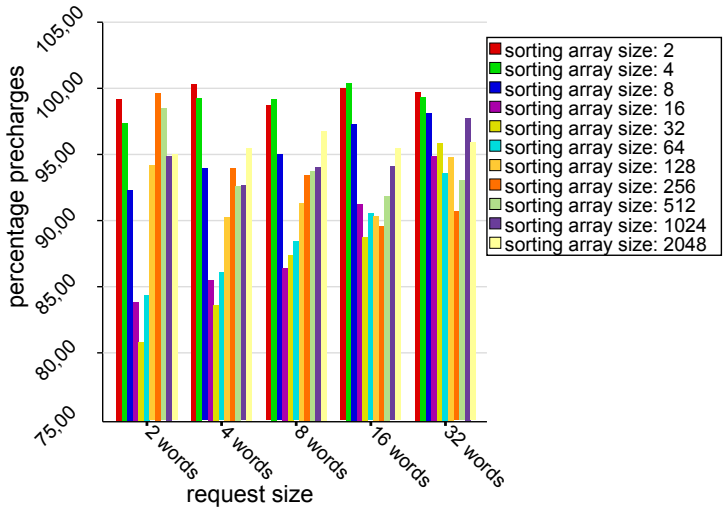


Figure 5.2.6: The percentage of precharges in the same bank while processing from that bank using the advanced simulator. The results shown is the dynamic boundary in comparison with a static boundary for a Smallest Address First policy. For clarity a bar graph is used instead of line graph.

5.2.1.3 Results Largest Block First, Static Boundary Policy

The results for both simulators for the Largest Block First policy with a static boundary differ more then the Smallest Address First policies when looking at the speed-ups. This is visible when looking at figure 5.2.7. The advanced software simulator is showing a minor improvement compared to the case where no reordering policy is applied.

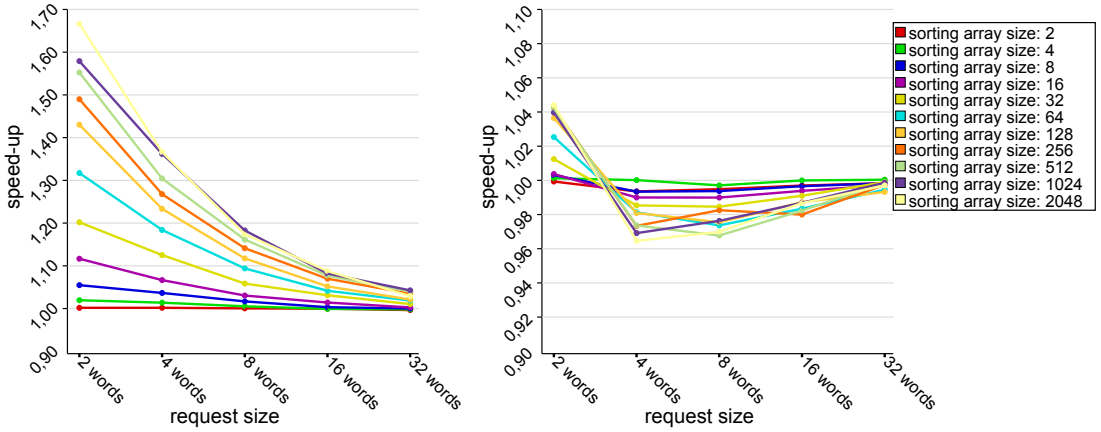


Figure 5.2.7: The speed-up that is gained by using the Largest Block First policy with a static boundary. The simulations are made with several sorting array and request sizes.

The basic software simulator is showing only a minor improvement compared to the Smallest Address First policy with a static boundary for the sorting array sizes of

8 elements or less. The speed-up for the Largest Block First policy is reduced in all other cases. This is made visible in figure 5.2.8.

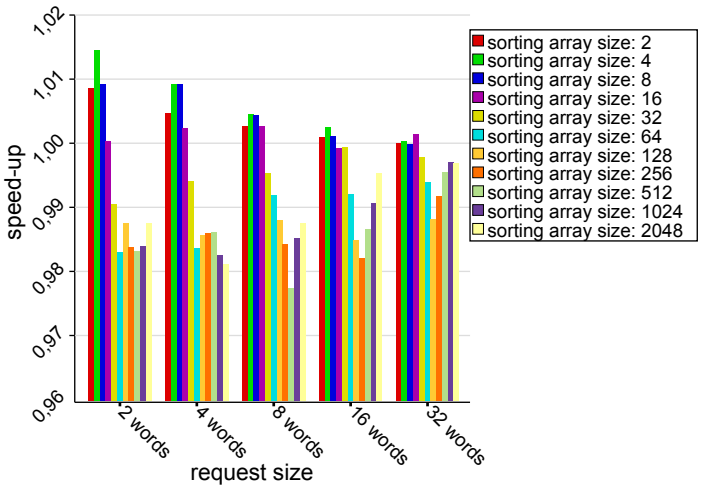


Figure 5.2.8: The speed-up that is gained by using the Largest Block First policy with a static boundary instead of a Smallest Address First policy, also with a static boundary. This result is for the basic software simulator and for clarity a bar graph is used instead of line graph.

The small to no improvement of the speed-up for the advanced software simulator does not show in the reduction of row changes in the same bank. The improvement in the amount of row changes in the same bank while processing from that bank is not as low as the speed-up of the advanced software simulator is suggesting as seen in figure 5.2.9. In this figure the reduction of row changes for both software simulators is presented.

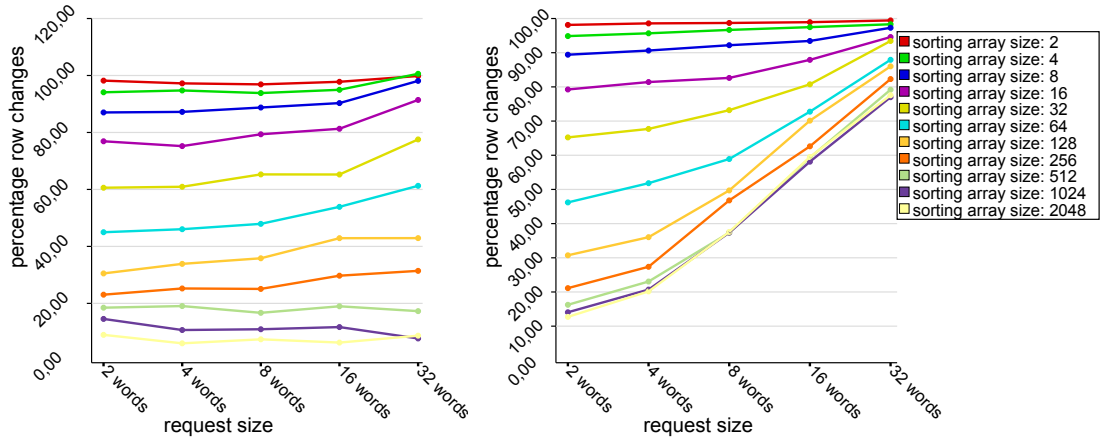


Figure 5.2.9: The percentage of row changes in the same bank that is gained by using the Largest Block First policy with a static boundary. The simulations are made with several sorting array and request sizes.

The reduction of row changes in the same bank while processing from that bank shows a similar behavior as with the Smallest Address First policy with a static boundary for the advanced software simulator. This can be seen in figure 5.2.10 where the Largest Block First policy is normalized with the Smallest Address First policy.

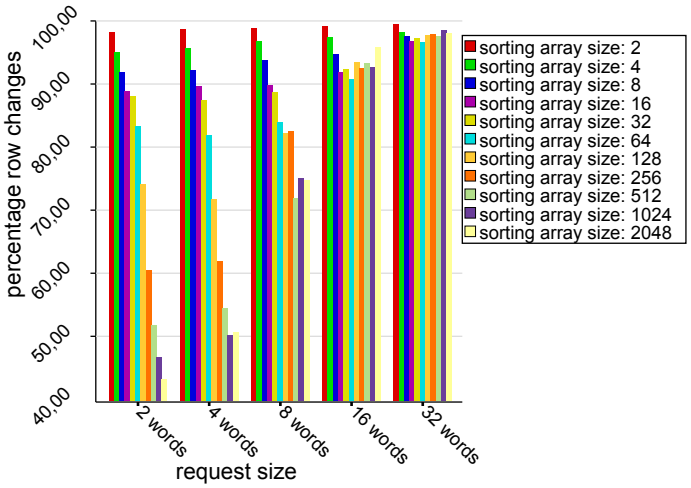


Figure 5.2.10: The percentage of row changes in the same bank that is gained by using the Largest Block First policy with a static boundary compared to the Smallest Address First policy. The simulations are made with the advanced software simulator and use several sorting array and request sizes.

There are no large differences between the Smallest Address First and Largest Block First in case of large request sizes for the advanced software simulator. In case of small request sizes the Largest Block First has even less row changes, but this does not show in the speed-up.

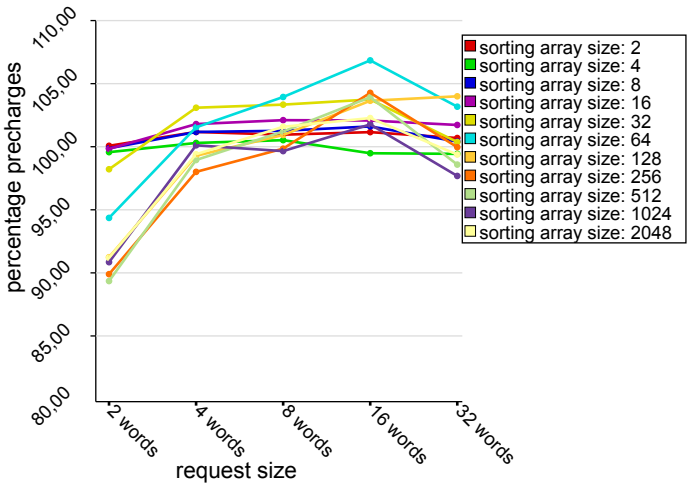
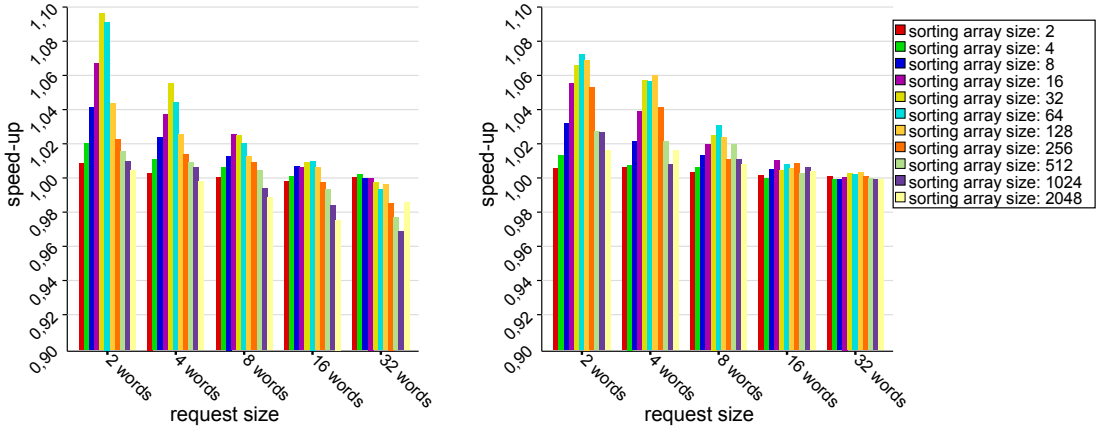


Figure 5.2.11: The percentage of precharges that is gained by using the Largest Block First policy with a static boundary for the advanced software simulator. The simulations are made with several sorting array and request sizes.

The small improvement in speed-up for the advanced software simulator can be explained more when a look is taken to the reduction of precharges in figure 5.2.11. This shows that despite of the reduction of row changes in the same bank, the amount of precharges is not reduced as much as it is with the Smallest Address First policy. These precharges should however been issued during a transfer of data and not stall the memory as much as it is doing at the moment.

5.2.1.4 Results Largest Block First, Dynamic Boundary Policy

The dynamic boundary shows a similar improvement as with the Smallest Address First policy in comparison with a static boundary version, as can be seen in figure 5.2.12 for the basic and advanced software simulators. This means that the speed-ups for the advanced software simulator are still minor, but the improvement that is gained by using a dynamic boundary is similar as with the Smallest Address First policy. The speed-up is still increased up to 1.08 for the dynamic boundary compared to the static boundary.

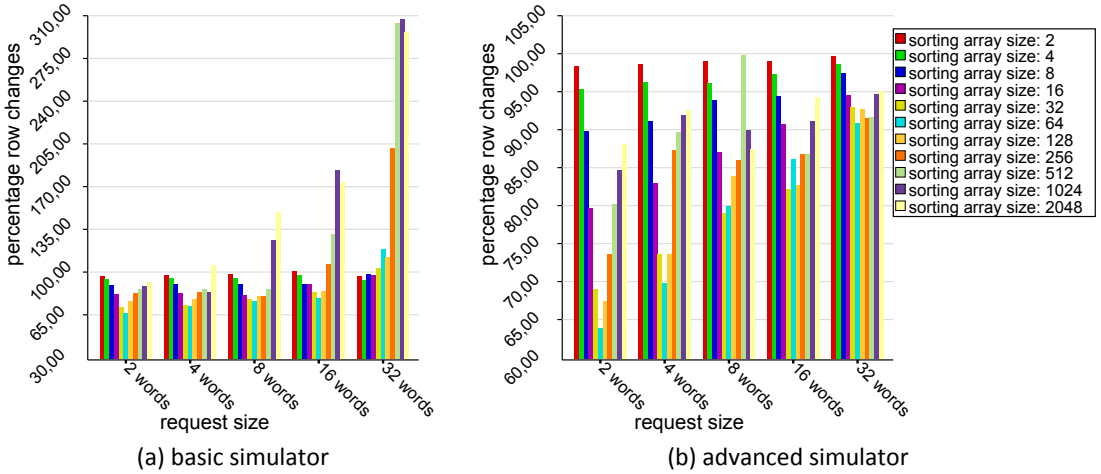


(a) basic simulator (b) advanced simulator

Figure 5.2.12: The extra speed-up that is gained by using the dynamic boundary in comparison with a static boundary when a Largest Block First policy is used. For clarity a bar graph is used instead of line graph.

Also for the Largest Block First policy an extra reduction is possible for the row changes in the same bank while processing from that bank. The amount of row changes further decreases when a dynamic boundary is used instead of an static boundary. For the Largest Block First this is shown in figure 5.2.13. The results from both simulators are showing an improvement for the middle large sorting array sizes around $32-64$ elements, the same as with the Smallest Address First policy.

Besides the lower amount of row changes in the same bank, the precharges are also reduced as can be seen in figure 5.2.14. A dynamic boundary clearly improves the results in comparison with a static boundary, for all policies.



(a) basic simulator (b) advanced simulator

Figure 5.2.13: The percentage of row changes in the same bank while processing from that bank. The results shown is the dynamic boundary in comparison with a static boundary for a Largest Block First policy. For clarity a bar graph is used instead of line graph.

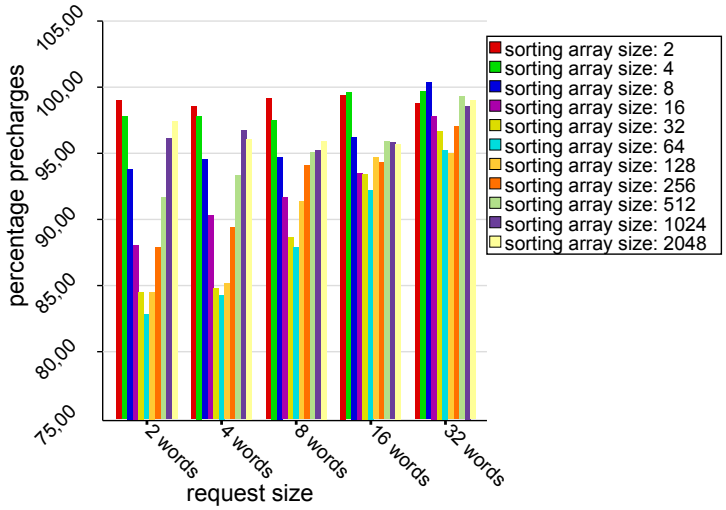
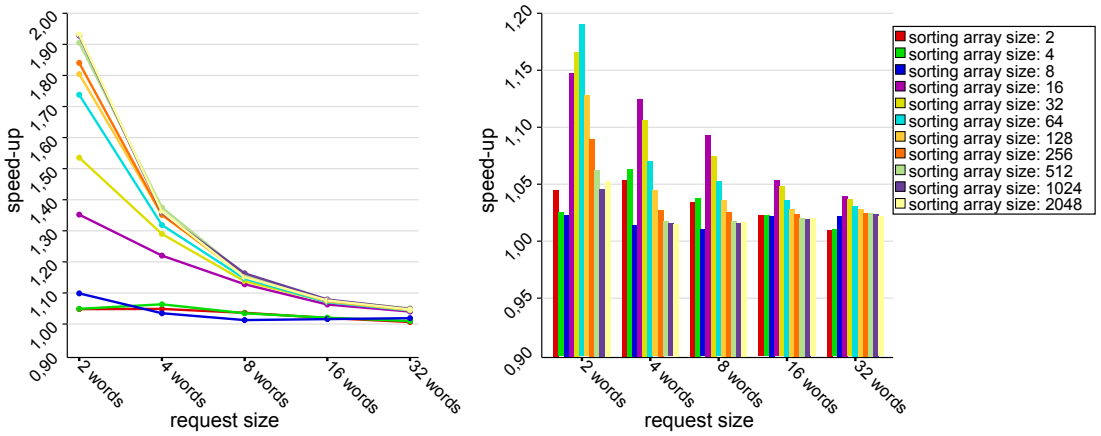


Figure 5.2.14: The percentage of row changes in the same bank while processing from that bank using the advanced simulator. The results shown is the dynamic boundary in comparison with a static boundary for a Smallest Address First policy. For clarity a bar graph is used instead of line graph.

5.2.1.5 Results Sorting Array per Bank Policy

The Smallest Address First policy with a dynamic boundary is used as policy for the sorting arrays that are used for the Sorting Array per Bank as described in section 4.1.6. As stated before this implementation is not an policy on its own, but it still interesting to consider as the results can be interesting. This policy is only implemented in the advanced software simulator. As expected the result for this policy are an improvement in comparison with the normal Smallest Address First policy. However, when an input generator is used that

uses more variation in the bank addresses then the 3D-FFT algorithm, better results are expected.



(a) normalized with no reordering policy (b) normalized with SADB

Figure 5.2.15: The speed-up that is gained by using the Sorting Array per Bank policy in comparison with no reordering policy and the Smallest Address First policy with a dynamic boundary. For clarity a bar graph is used instead of line graph.

The extra improvement in the speed-up is starting from a total sorting array size of 16 elements, which means a sorting array of 4 elements for each bank. This can be seen in figure 5.2.15. Around 32 elements for the total sorting array size the largest improvement is visible in this case.

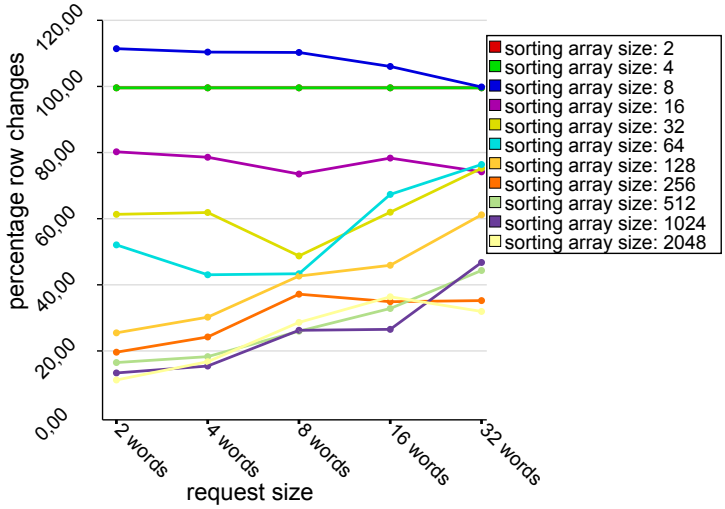


Figure 5.2.16: The percentage of row changes in the same bank that is gained by using the Sorting Array per Bank policy.

Figure 5.2.16 shows the percentage of row changes compared with no reordering policy. For a total sorting array size of 8 elements, a sorting array of 2 elements for

each bank, there are more row changes in the same bank done then in the case without reordering policy. All other total sorting array sizes have an improvement.

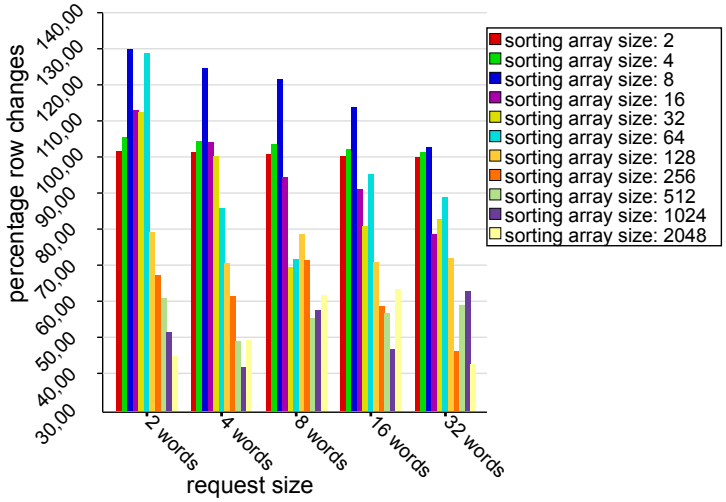


Figure 5.2.17: The percentage row changes that is gained by using the Sorting Array per Bank policy in comparison with the Smallest Address First policy with a dynamic boundary. For clarity a bar graph is used instead of line graph.

The amount of row changes in the same bank while processing from that bank are less after a total sorting array size of 128 elements in comparison with the Smallest Address First policy with a dynamic boundary. This is shown in figure 5.2.17.

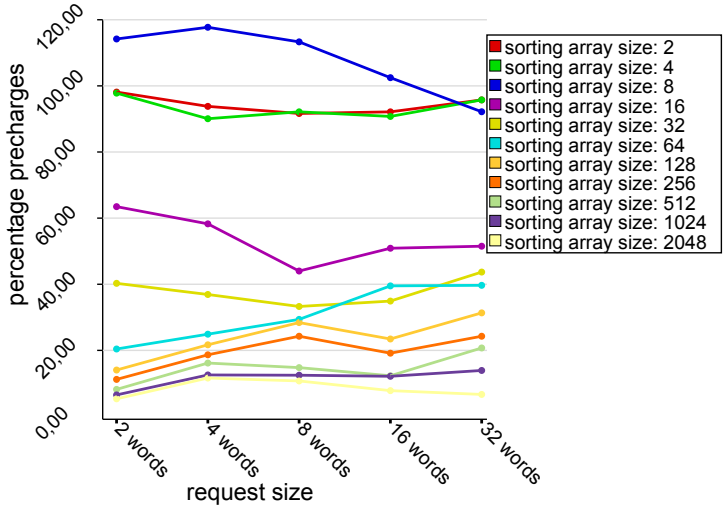


Figure 5.2.18: The percentage of precharges in the same bank that is gained by using the Sorting Array per Bank policy.

The amount of precharges issued to the DRAM are higher then the case where no reordering is done for the total sorting array size of 8 elements, the same with the amount of row changes in the same bank. This is shown in figure 5.2.18. The amount of

the precharges is lowered to values of 10% and less of the amount in the case where no reordering is done.

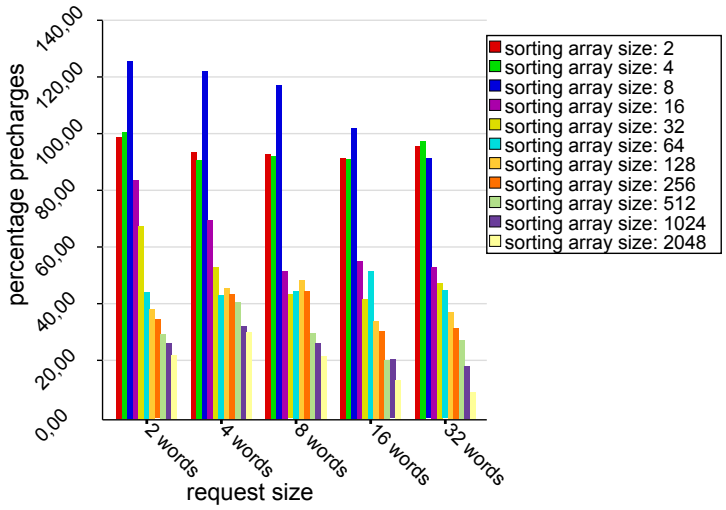


Figure 5.2.19: The percentage precharges that is gained by using the Sorting Array per Bank policy in comparison with the Smallest Address First policy with a dynamic boundary. For clarity a bar graph is used instead of line graph.

In comparison with the Smallest Address First policy with dynamic boundary only the total sorting array size of 8 elements is performing worse, all other total sorting array sizes are showing an extra improvement. Figure 5.2.19 is showing these improvements.

5.2.2 Estimated Hardware Costs

In this subsection the hardware costs are estimated for the reordering policies. The reordering policies are introduced in section 4.1 and can be differentiated into three categories:

- **Smallest Address First policy** with a static or dynamic boundary. With this policy the requests with the smallest row and bank address are sent as first to the Back-End.
- **Largest Block First policy** with a static or dynamic boundary. With this policy the largest block of requests with the same row and bank address are sent as first to the Back-End.
- **Sorting Array per Bank** where a separate sorting array with a reordering policy is used for each bank. An arbiter for the input and the output decides which request is sent to the Back-End.

The hardware costs are estimated in section 5.2.2.1 for the Smallest Address First policy while in section 5.2.2.2 the estimation for the Largest Block First policy is described. In section 5.2.2.3 an estimation is made for the Sorting Array per Bank implementation.

5.2.2.1 Smallest Address First Policy

To have a functional implementation of the Smallest Address First policy a few parts are necessary for each position where a request can be inserted. For each position the following is at least necessary:

- One register of size *N* for the request that is being stored. *N* is the amount of bits of a request.
- One two-to-one multiplexer of width *N* to route the incoming request or the request of the previous position to the input of the register.
- An overlap controller to check if there is an overlap for the current element. This overlap controller needs at least:

  One *M* sized adder to add the size of the request inside the register to its address. Where *M* is the width of the address.

  Two *M* sized comparators that compare the address of one request with the address added with its size of another request.
- Two one-bit registers for the current boundary and occupation (if a position is occupied or not).
- Minor amount of gates to combine all of the above.

The address of the incoming request is added with its size by a separate *M* sized adder for the overlap control. This value is independent of the position in the sorting array and can therefore be done just once. Other logic that is independent of the number of positions is the output register and a comparator that is used in case of a dynamic boundary version. The row and bank address of the incoming request is compared to the row and bank address of the request inside the output register with this comparator. The result of this comparator is used at each position in the sorting array by the logic.

5.2.2.2 Largest Block First Policy

To have a functional implementation of the Largest Block First policy a few parts are necessary for each position where a request can be inserted. For each position the following is at least necessary:

- One register of size *N* for the request that is being stored. *N* is the amount of bits of a request.
- One register of size *M* where the block size is stored. *M* is the amount of bits needed to store the maximum block size possible.
- A comparator to check if the request that is being inserted belongs to the same block as the request that is present at the current location.

- An overlap controller to check if there is an overlap for the current element. This overlap controller needs at least:

  One *L* sized adder to add the size of the request inside the register to its address. Where *L* is the width of the address.

  Two *L* sized comparators that compare the address of one request with the address added with its size of another request.

- Two one-bit registers for the current boundary and occupation (if a position is occupied or not).

- Intensive amount of logic to decide if the current request has to be sent to the Back-End.

- Extra logic and possible register for each position is necessary to prevent memory inconsistencies.

The inserted request should get an size of the block that is already present in the sorting array plus one, so an incrementing block plus logic to get the block size out of the sorting array is needed. The comparator that is used to check if the request that is being inserted can be used to see if a request belongs to an block inside the boundary, in case a dynamic boundary is used.

5.2.2.3 Sorting Array per Bank

The Sorting Array per Bank implementation is using a sorting array for each bank, as described earlier. An input and output arbiter are used besides the sorting arrays. For each sorting array the estimated hardware is the same as described in the previous sections. This means that all banks together use slightly more hardware than just one large sorting array.

The logic that is used independently of the number of positions in the sorting array, like the output register, is extra. Another extra component is a comparator for each sorting array. This comparator is used to see if the request that will be read next has a different row address then the one currently in the output register. With this information the output arbiter can decide which sorting array should be read.

For the input arbiter just a few gates are needed to control the signals like *write_enable*. The input is just wired to all inputs of the separate sorting arrays.

The output arbiter uses extra logic for the reading then the input arbiter. A multiplexer is used to wire the correct sorting array output to the output. The sorting arrays can be controlled with minor logic, a block that decides what bank should be read next, and a small register of size K. K is the amount of bits that are needed to represent all banks, $log(\#banks)$. The block that decides what bank should be read next does not have to have complex logic, the next block can easily be decided with just the information of the sorting arrays if a sorting array is empty or not.

5.2.3 Summary

In this section a small summary is given for the software simulation results and the estimated hardware costs. With the simulation results and hardware estimations a founded decision can be made on what reordering policy should be implemented. In section 5.2.3.1 the summary of the simulation results is described while in section 5.2.3.2 the estimated hardware costs summary is given. The chosen reordering policy is given in section 5.2.3.3.

5.2.3.1 Summary Simulation Results

For the simulations described in this section the 3D-FFT algorithm was chosen as input as described earlier. Although other algorithms or random data input generators will produce other results, the relationships between the policies are the same.

The results of the basic software simulations show that the Largest Block First policy has a similar or smaller speed-up compared to the Smallest Address First policy. Both policies could be implemented when only the software simulation results of the basic simulator are considered and not the hardware costs. The advanced software simulator is less positive about the Largest Block First policy, the speed-ups for this policy are a fraction of the speed-ups for the Smallest Address First policy.

The differences between a static and a dynamic boundary for both policies and both simulators are quite the same. Using a dynamic boundary over a static boundary shows an improvement, especially for the sorting array sizes around $32 - 64$ elements.

A Sorting Array per Bank implementation is showing an extra improvement over the Smallest Address First policy with a dynamic boundary. The results with this implementation is depending on the amount of variation in bank addresses for the requests, more then the policies without a sorting array for each bank. This because a sorting array for a bank is full sooner when only requests of bank *a* are entering the memory controller in comparison with a policy with the same total sorting array size. Either way, the results look promising with the Sorting Array per Bank implementation.

5.2.3.2 Summary Estimated Hardware Costs

In the hardware estimation it is easily seen that the Largest Block First policies use an higher amount and more complex logic then the Smallest Address First policies, even without mentioning the exact number of gates. Complex logic is needed to decide to what block a request belongs or how a memory inconsistency is prevented in case of the Largest Block First policies. In case of the Smallest Address First policy the memory inconsistency prevention costs a few simple gates, besides the overlapping controller that is also present in the Largest Block First policy.

The difference between a static or dynamic boundary hardware wise is small, only one comparator for the whole sorting array and one or two gates for each position. This means that a dynamic boundary is interesting, even for a small improvement.

Extra logic is necessary when the Sorting Array per Bank implementation is used. This is however not a lot, just one multiplexer, a small register, and a few gates. A small price

to pay, considering the extra improvement that the software simulation is suggesting for this implementation.

5.2.3.3 The Chosen Reordering Policy for the Memory Controller.

The Smallest Address First policy with a dynamic boundary is implemented in the memory controller. Even though the Sorting Array per Bank suggests a larger improvement, this implementation is an extension and not a policy on its own. This means that once the Smallest Address First policy with the dynamic boundary is implemented it takes little effort to use this for a Sorting Array per Bank implementation.

The simulation results for the Smallest Address First policies outweigh the Largest Block First policies, for the basic and the advanced software simulator. With also the extra hardware costs in mind, the decision to take the Smallest Address First policy is taken easily. A dynamic boundary is chosen over the static boundary do to its extra improvement with small extra hardware costs.

6 Hardware Implementation

The implementation of the memory controller with the data allocation and reordering policies is described in this chapter. These policies where researched in the previous chapters and are implemented in the data buffer and sorting array respectively. The implementation of the design focuses on the modularity of the design, where a sorting array is easy interchangeable. The same holds for the data buffer with a different allocation policy. Those two blocks form the most important part of the memory controller and their implementation is therefor described independently.

The implementation of the sorting array is described in section 6.1 and the implementation of the data buffer in section 6.2. The bridge where large requests are converted into smaller requests that the Back-End can handle, is described in section 6.3. In section 6.4 is described how everything is put together.

6.1 Sorting Array

In this section is described how the sorting array is implemented in the hardware. The Smallest Address First policy with a dynamic boundary is the policy that is chosen to be implemented in the sorting array, as was described in the previous chapter. For the sorting in this policy an insertion sorting algorithm is used where a request is immediately inserted in the correct position. In section 6.1.1 an overview is given of the sorting array. A more detailed description is given in section 6.1.2 of the sorting elements that are used for the reordering.

6.1.1 Overview of the Sorting Array

A FIFO like interface and control is chosen to make the sorting array easily interchangeable. Because of that the sorting array needs logic and signals such as read and write enable, read and write acknowledgment, and full and empty signals besides the logic that implement the reordering policy. A figure of the overview of the sorting array is given in figure 6.1.1.

Because the reading is done with the Smallest Address First policy a simple incrementing pointer can be used that makes one output of a sorting element hot. The sorting elements make sure that no request is overwritten with this reading pointer plus an extra pointer that points to the element after the last occupied element, the next write pointer. These pointers are also used to generated the *full* and *empty* signals for the outside world, besides the controlling of the sorting elements.

These *full* and *empty* signals are also used to acknowledge a write or read. If the sorting array is not full the request can be acknowledged as it is known for sure the

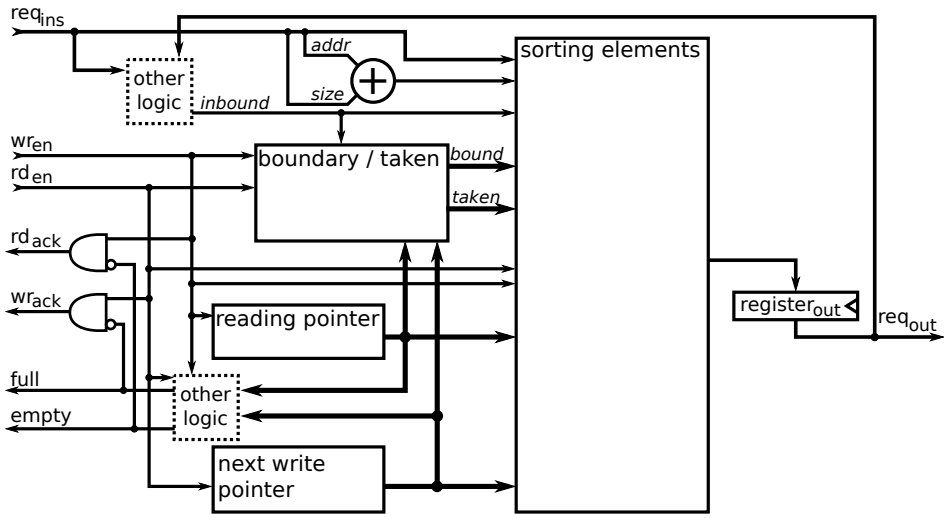


Figure 6.1.1: The implementation of the sorting array with the sorting elements, boundary generation block, reading and writing pointers, and minor logic represented by gray clouds.

request is being inserted somewhere. The same holds for the acknowledgment of the read, when the sorting array is not empty it is known that a request can be read.

For the boundary feature a block is used that generates the boundary and taken signals. These signals are used by the sorting elements to see if an element is inside the boundary and/or occupied. With that information the sorting elements can decide where the new inserted request has to be placed. The generation of these *bound* and *taken* signals is possible with the reading and next writing pointers and the *inbound* signal. This *inbound* signal is used for the dynamic boundary and tells if the inserted request should be placed inside the boundary or not. This signal is driven by information like the address values of the inserted and last outputted request.

The adder that adds the address and size of the incoming request is used by the overlap controller inside the sorting elements. This information is used for the determination if the inserted request is overlapping with a request already present in the array. Just one adder can be used by moving this operation out of the overlapping controller, this is possible because the value is the same for each sorting element.

6.1.2 The Sorting Elements

The sorting elements are used for the actual reordering inside the sorting array. Each element has a register to store the request, a multiplexer that is used for wiring the incoming request or the previous request in the register, and a control block that controls the register and multiplexer. A scheme of this is given in figure 6.1.2.

Not visualized but still present is the logic that selects the correct output of the element that is currently being read. The correct output is chosen by the reading pointer that was mentioned before. Also the logic that drives the $overlap_{in}$ signal is not present. This signal goes to 1 when there is at least one sorting element that signals a overlapping by putting a 1 on its $overlap_{out}$ signal.

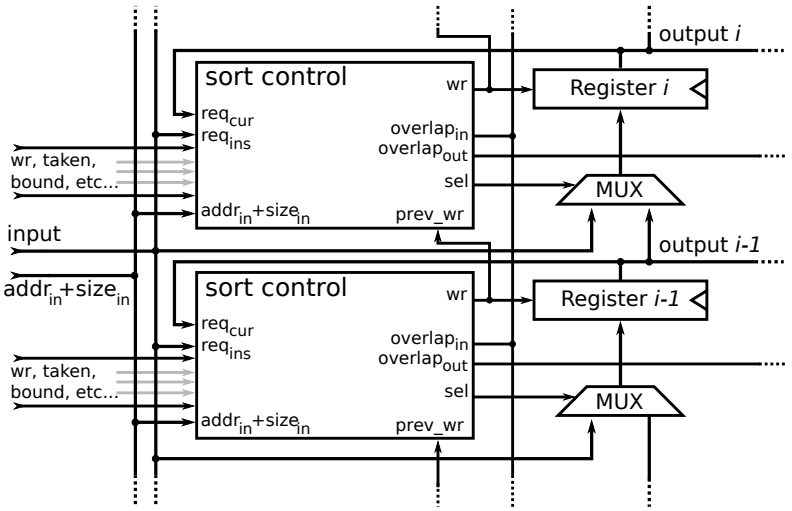


Figure 6.1.2: The implementation of the sorting elements with a sort control block, register, and multiplexer for each position.

In section 6.1.2.1 a detailed description is given of the logic that is implemented for the sort control block. The overlapping control is also present in this sort control block and is described in section 6.1.2.2.

6.1.2.1 The Sorting Control Block

The sorting control block forms the core of the reordering process, it decides how the multiplexer and register should be driven to sort the incoming request. A scheme of this control block is given in figure 6.1.3. The whole gate structure that is used to combine all signals to the *select* and *wr* signals is represented as a simple block in the scheme, but is described in more detail in this section. The overlap control block that is also present in this scheme is described in an other section, namely section 6.1.2.2.

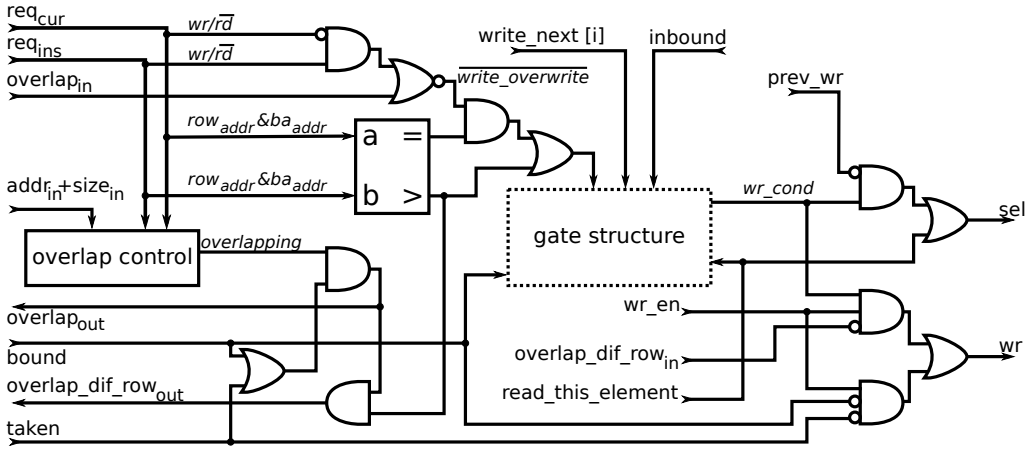


Figure 6.1.3: The implementation of the control block of each sorting element with the overlap control block, a comparator and logic gates.

One of the conditions when the sorting element has to write a request to the register is when the row and bank address of the current request are larger than the row and bank address of the request that is being inserted. The row and bank address can also be equal when the $\overline{write\_overwrite}$ signal is high. In that case there is no overlap at that row and bank address and the writing conditions of the requests are in such a way that a write to read delay does not appear.

Another condition when the sorting element has to write is when the current element has the *next writing pointer* pointing to it, or when the element is not inside the boundary but the *inbound* signal is high. In those cases the element has to write the request of the previous sorting element or the inserting request to its register. A write is finally checked if the current element is not taken or if the inserting request is not overlapping with a request that has a different row and bank address.

This overlap with a different row and bank address is checked by using the *overlap* signal of the overlap control and the *greater than* signal of the comparator. Memory consistency cannot be guaranteed when a request is overlapping with an other request that has a different row and bank address. Not acknowledging this request and waiting for the overlapping request to disappear is than the solution to protect the memory consistency.

6.1.2.2 The Overlap Control Block

The overlap control block is responsible for checking if there is an overlap with the request in the current element and the request that is being inserted. The value of the address added with the size of the request being inserted is used in this block. This value is calculated outside the sorting element as it has to be calculated only once.

A request is overlapping and causing a possible inconsistency when at least one write request shares a data location with an other request. All of the following is true when there is an overlap:

- The starting address of request A is smaller then the address of request B added with the size of request B.
- The starting address of request B is smaller then the address of request A added with the size of request A.
- At least one of the requests is a request that writes to the memory.

These simple rules where obtained in section 4.3.1 and result in a simple implementation which is given in figure 6.1.4.

One adder is used to add the size with the address of the request currently in the element. Two comparators will do the comparison between the values of the starting address of one request with the ending address of the other request. With just 3 or-gates and a three way and-gate everything is put together to generate the *overlapping* signal.

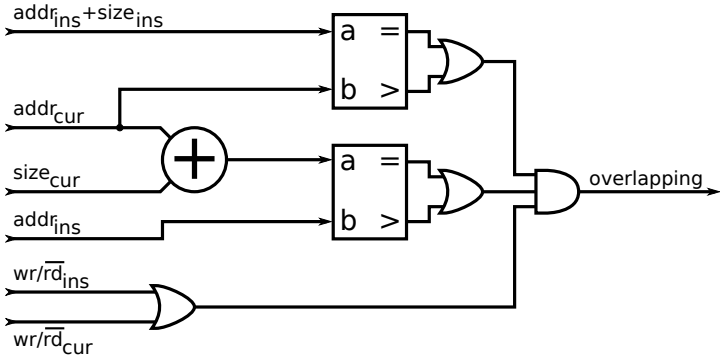


Figure 6.1.4: The implementation of the overlap control block with one adder, two comparators and a few gates.

6.2 Data Buffer

How the data buffer is implemented in hardware is described in this section. This data buffer has the allocation policy implemented besides the data storage as described earlier. The policy that is implemented for the allocation of the requests is the *more advance allocating and simple de-allocating* policy described in section 4.2.3. This allocating and de-allocating functions separately from the data storage as can be seen in figure 6.2.1.

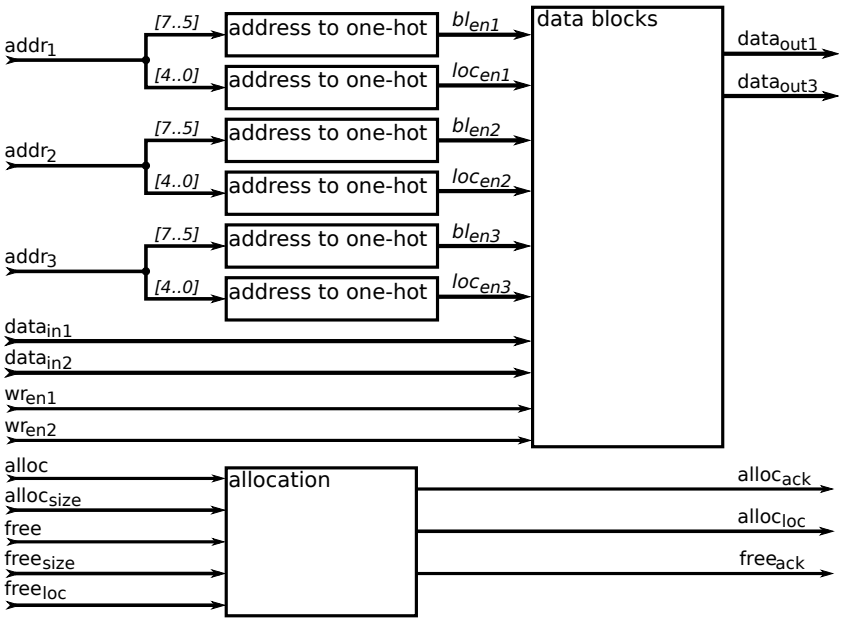


Figure 6.2.1: The data buffer with the data-blocks and allocation blocks. In this case a total size of 2^8 locations, organized in 2^3 data blocks of size 2^5, is used to store the data.

The reason for the multiple in- and output ports in the data buffer is because of the full-duplex property the optimization block should have, as was described in section 3.4.3.1. This results in two data inputs, two data outputs, and three address ports for all the data

accesses.

The addresses that used for the addressing of the data in the data buffer are converted to block-enable (bl_{en}) and location-enable (loc_{en}) signals. This conversion is done by the *address to one-hot* blocks. The data blocks uses the output of these signals to select the correct block and location in that block for the reading and writing of the data. The data blocks are further described in section 6.2.1. The allocation policy that is implemented in a separate block is described in section 6.2.2.

6.2.1 Data Blocks

The data blocks with the data registers are driven by the block- and location-enables besides the normal read and write signals. The amount of hardware needed for the *address to one-hot* blocks is reduced by using an enable signal for a block of data registers. These enable signals are used for enabling the correct data register when writing or reading. One of these registers with surrounding hardware is given in figure 6.2.2.

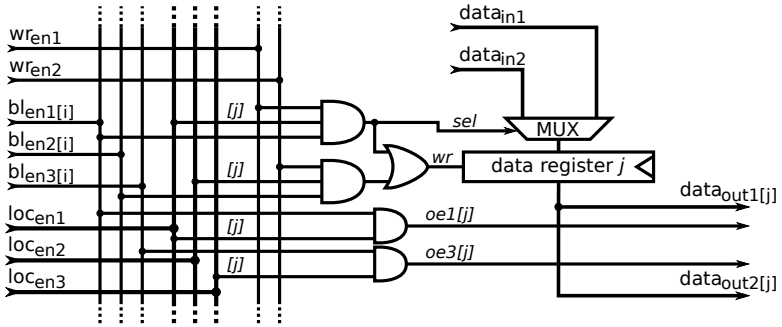


Figure 6.2.2: The entry where the data is stored, in this case entry *j* from block *i*.

The block- and location-enable signals from address 1 and 2 are used for the writing, while the enable signals from address 1 and 3 are used for the reading of data. The correct data that has to be written in the register is selected by the multiplexer and is by default the data from the input port 2. Only when the current location is being written by port 1 the multiplexer will select this data. Writing by both ports at the same location at the same time is not possible.

For the reading a read enable is not needed, changing the address will immediately result in a different location to be accessed. The current location will transfer its contents to the output of port 1 or 2 when the output is enabled by the correct location- and block-enable signals.

It should be easy to see that different blocks of data registers only differ in a different block enable signal at that block. The rest of the logic and connections are the same.

6.2.2 Allocation Block

The allocation is done by a separate part in the data buffer and is based on the *more advance allocating and simple de-allocating* policy as described earlier. The allocation of this policy needs complex logic, while the de-allocation is fairly easy. A calculation

is done to see how much space is available at the next free block when a request is allocated. Clearing one bit is all that is necessary to de-allocate, or free, a space. In figure 6.2.3 a scheme is given of the allocation block.

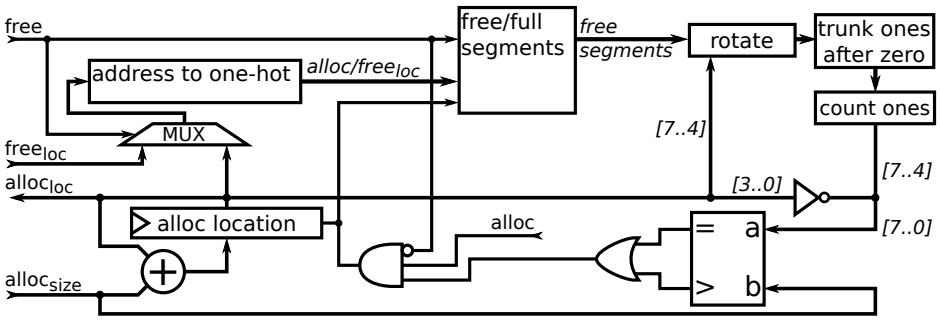


Figure 6.2.3: The allocation block of the data buffer with the block that keeps the allocated segments, a rotate block, trunk-ones-after-zero block, one counter, and other logic to decide if a request can be allocated.

The bits that store the state of the data locations are located in the *free/full segments* block. The output of this block is a signal for each segment of locations that state if there is at least one location in the segment that is marked as used (0) or not (1).

A rotate block will rotate the information of the segments and after truncating all ones after the first zero the size of the next free block can be determined. This is done by counting the ones and adding the number of locations still available in the current segment. The size of the next free block is compared with the size of the request that has to be allocated by the comparator. If this is large enough the location in the *free/full segments* block can be marked as used and the allocation location can be updated by the adder.

The multiplexer in combination with just one *one address to one-hot* block is used because in this implementation it is not possible to allocated or de-allocate at the same time. This saves the amount of logic for the *free/full segments* block which is given in figure 6.2.4. For an implementation that can allocate and de-allocate at the same time at least another *address to one-hot* block and input bus for the *free/full segments* block are necessary.

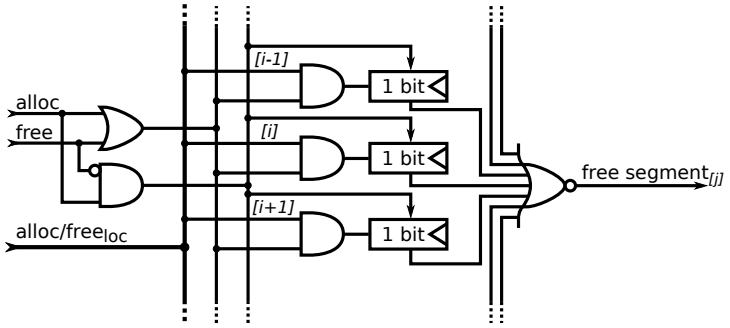


Figure 6.2.4: The allocation status for each segment and location. The register stores if a location is allocated or not, an or-tree for each segment generates the output.

A single bit register is used to store the state of the location. This register is controlled by a single and-gate for each position and an and- and or-gate for the whole block. Each segment marks it state with one large nor-gate or or-tree with an inverter at the end. The output is zero when there is just one register in the segment that has a one stored for marking that location as used.

6.3 The Bridge

The implementation of the bridge is described in this section. The requests that are sorted in the sorting array do not have to be of the size that the back-end can handle, as described earlier. The bridge will convert those requests in smaller requests that the back-end actually can handle. For this conversion some components are used which are also visible in figure 6.3.1.

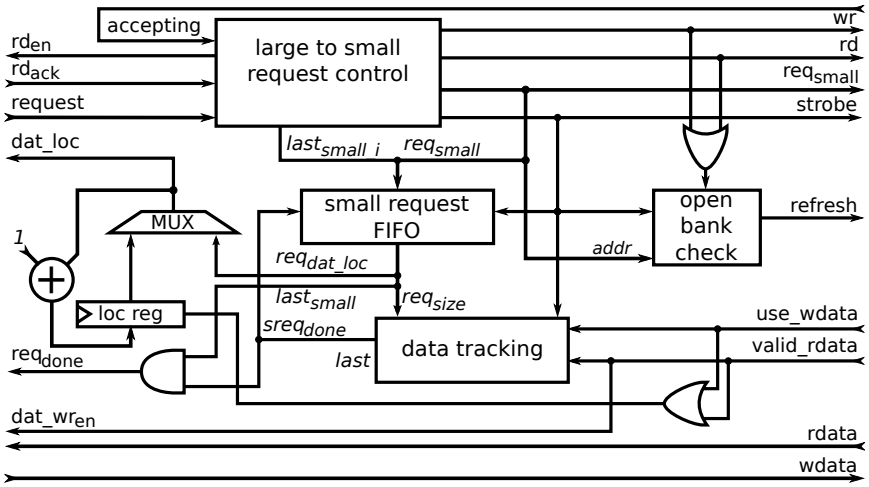


Figure 6.3.1: The implementation of the bridge with the *large to small request control* block, *open bank check* block, and *data tracking* block among other logic.

The multiplexer, register, and increment block that do not belong to a sub-block are used for the location of the data that has to be read or written in the data buffer. The location is incremented each time a word is read or written by the back-end and *use_wdata* or *valid_rdata* is high. The starting location is loaded from a small request when this small request is loaded into the data tracking block and is processed next by the back-end.

The *large to small request control* block reads the requests from the sorting array into the bridge, converts the large request into smaller ones, and issues them to the back-end. This block is described in more detail in section 6.3.1. The *small request FIFO* and the *data tracking* block are used to track the smaller requests and signal the sorting array when an original large request is fully processed. The implementation of the *data tracking* block is described in section 6.3.2.

The $sreq_{done}$ signal from the data tracking is set high when all data is processed from the current small request. The optimization block can be signaled, by setting req_{done}

high, that a large request is done with this property in combination with the *last* signal and an and-gate.

The *open bank check* block is used for the OpenCores back-end as described earlier. This back-end does not provide the check to see if a bank is open for too long and thus this checking block is necessary. The implementation is described in more detail in section 6.3.3.

6.3.1 Large to Small Request Control

The *large to small request control* block converts the large request from the sorting array into smaller request that the back-end can handle. This is done by generating smaller requests based on the size, address, and data location of the original large request. A scheme of the block is visible in figure 6.3.2.

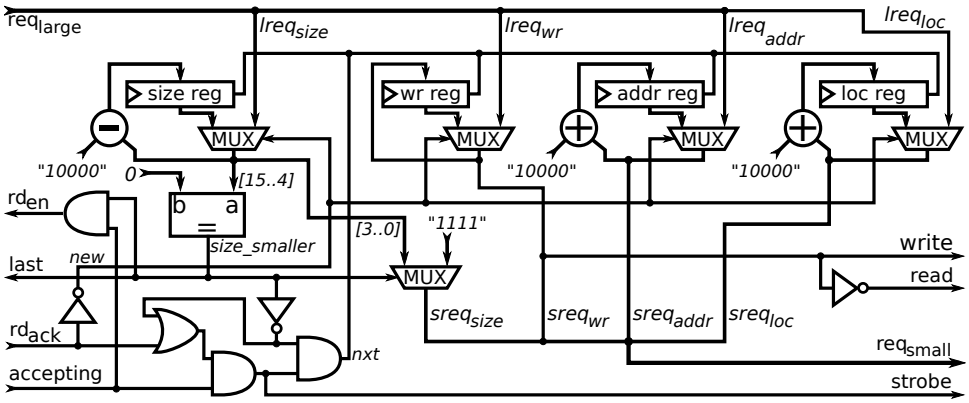


Figure 6.3.2: The long to small request control block of the bridge. Here the request is chopped into smaller requests that the back-end can handle.

The original size is compared with the maximum size possible when a new request enters the bridge. The first small request gets the maximum size and original address and data location. All small requests that are issued after that have an address and data location that has been increased by the maximum size. This is done by the adder, register, and multiplexer for each property. The maximum size of a small request is subtracted from the total size each time a small request is issued to the back-end. This size is only used when the size is smaller then the maximum size and the *size_smaller* signal is high, in other cases a pre-defined maximum size is used.

The *size_smaller* signal is also used to determine if the current smaller request is the last small request that is issued. A new request can be read from the sorting array when this signal is high and the back-end can accept new requests. The smaller requests are strobed into the back-end with the *read* or *write* signal high when the back-end can accept requests and there is currently a request in the bridge.

6.3.2 Data Tracking

It is necessary to keep track of the data after converting a large request into smaller requests. The optimization block should know when a request is fully processed before it

can de-allocate the used space in the data buffer or return the read data to the front-end. This is only possible when the bridge knows for sure that all smaller requests from a large request are processed. The data tracking block is used for tracking the data transfers for each small request and its scheme is given in figure 6.3.3.

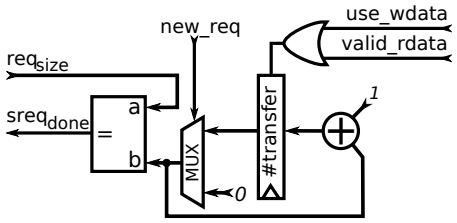


Figure 6.3.3: The data tracking block with single register, multiplexer, comparator, and increment block.

The total number of transfers of a small request is incremented each time there is a transfer and the *use_wdata* or *valid_rdata* signal is high. The counter is set to zero when the *new_req* signal is set and the multiplexer selects the zero as input instead of the previous value. A comparator is used to compare the total number of transfer against the size of the smaller request. The $sreq_{done}$ signal is set high when those are the same.

6.3.3 Open Bank Check

It is necessary to check if a row in a bank is not open for too long to prevent loss of data. This has to be done by the bridge because the OpenCores back-end does not support this feature. A *refresh* is issued to the back-end that will refresh all banks when at least one bank is open for to long. The current row is closed and the counter for that bank can be reset when a row is changed in that bank. The size of the counters for each bank is reduced by using a single counter that divides the frequency for the other counter to count with. In figure 6.3.4 a scheme is given for the open bank check.

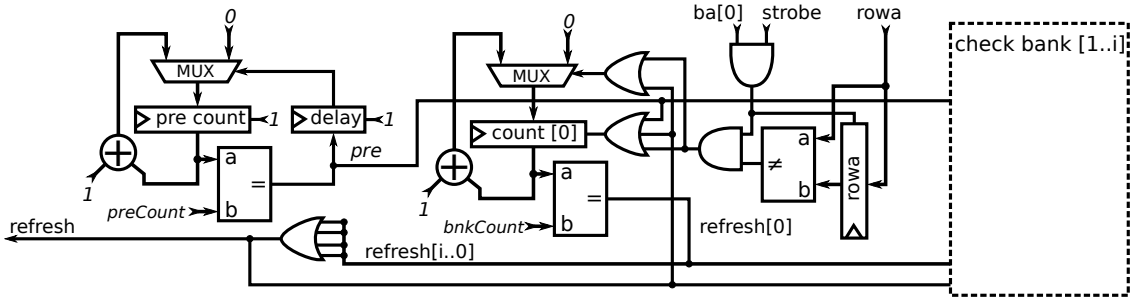


Figure 6.3.4: The implementation of the open bank check with one pre-counter and counters and checking per bank. For one bank in detail, others represented by a block.

The counter that divides the counting time for all other counters is a simple counter with a register, increment block, and multiplexer. This multiplexer will set the counter to zero when the counter reached the predefined *preCount* value. The *pre* signal will also drive the registers of the other counters besides resetting the pre-counter.

The counters that are used for each bank have more driving logic than the pre-counter. The register for these counters has to be written when the *pre* signal is high, a row in that bank changed or when a refresh is issued. In the last to cases the new value is zero, which is selected by the multiplexer. The row in that bank is changed when a new request is strobed in the back-end, the bank address is the one of the counter, and the strobed row address is not equal with the previous strobed request. This checking of this last property is done by a comparator and a register.

Each counter of each bank is connected to a comparator that checks the counted value with a predefined *bnkCount* value. The open bank check will send a refresh when those values are the same. All counters are set to zero and the data in the DRAM is still save.

6.4 Putting it all Together

How the data buffer, sorting array and bridge are connected to each other is described in this section. Some extra components are added because of the modular setup of the design. These are used to interact with the sorting array as a FIFO for example. A scheme of everything put together is given in figure 6.4.1.

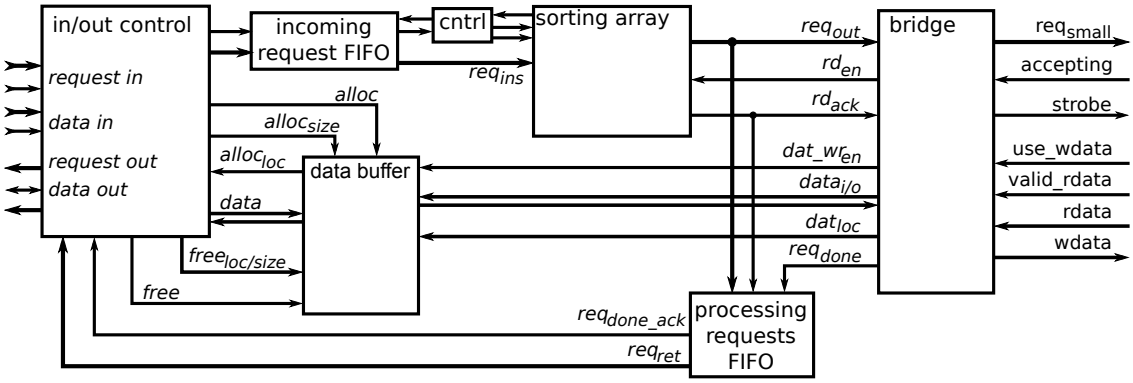


Figure 6.4.1: The optimization block with the bridge.

The sorting array, data buffer, and bridge are already described in previous sections. Two FIFO's and two control blocks are used to combine everything into a working controller. The *in/out control* block is responsible for issuing the allocate and de-allocates to the buffer, receive and return the data to the front-end, and send the incoming requests to the sorting array. This controller is described in more detail in section 6.4.1.

The small controller between the incoming request FIFO and the sorting array is a simple control that reads a request from the FIFO and writes it to the sorting array. The reason to have a FIFO in front of the sorting array is to be able to receive some requests already when the sorting array is full or not accepting a request because of an overlap.

Request that are transferred from the sorting array into the bridge are also copied to the processing request FIFO. The optimization block is able to identify which request with what data location is fully processed when the bridge is signaling that a request

is done with req_{done}. The request is then returned to the *in/out control* block that will return the read data and de-allocated the used space in the data buffer.

6.4.1 In/Out Control

The *in/out control* block is responsible for the receiving and returning requests, allocating the necessary space in the data buffer, and de-allocating that space after processing the request. The controller will also write the needed data into the data buffer in case the incoming request is a write request. The de-allocating and data returning part of the controller is discussed in section 6.4.1.1 while the allocating and writing data part is discussed in section 6.4.1.2.

6.4.1.1 De-allocating and Data Returning Part

After a request is processed the used space in the data buffer has to be de-allocated. The read data has to be returned to the front-end before de-allocating when the processed request was a read of data. The scheme that is used for the de-allocating, or freeing, of the used space is given in figure 6.4.2.

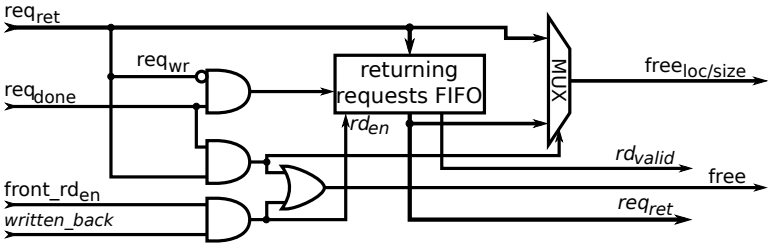


Figure 6.4.2: The freeing (de-allocating) part of the in/out control block.

A FIFO is used to store all requests that have to be returned to the front-end. These requests are thus only requests that had to read data. The requests that had to write will drive the *free* signal immediately. The multiplexer selects the used data location and size for the data buffer. The returning of read data has more complex logic and is given in figure 6.4.3.

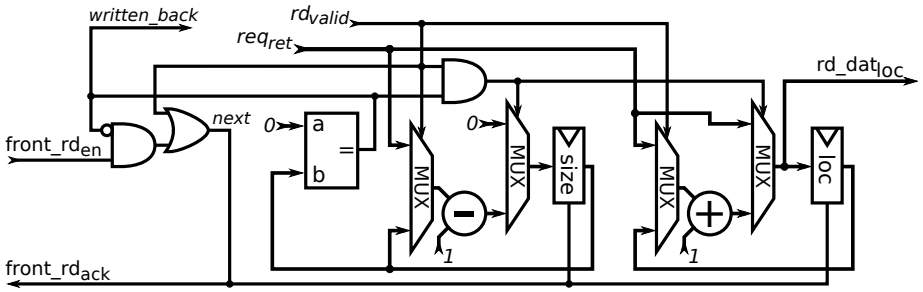


Figure 6.4.3: The data returning part of the in/out control block.

The data location is incremented and the used size is decremented while the front-end is accepting the read data. This is done by a few multiplexers, registers, and adders.

The de-allocating part is signaled that all data is written back by setting the *written_back* signal high when the used size reached zero.

6.4.1.2 Allocating and Data Writing Part

Space in the data buffer has to be allocated before a request can be processed. All needed data is then written to the data buffer in case of a write request. The request can then be put in the system where it will be processed. The allocation request is done by simple logic and given in figure 6.4.4.

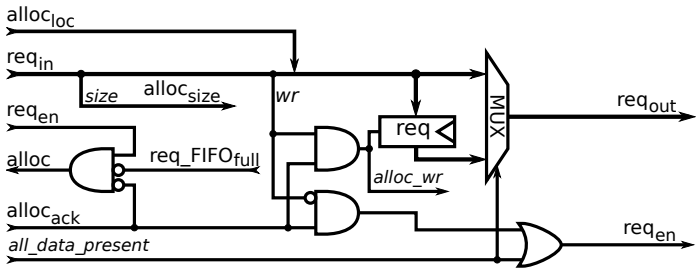


Figure 6.4.4: The allocating part of the in/out control block.

An allocation is immediately requested when a request enters the control block and the req_{en} is high. The request is transferred to the rest of the system when the allocation is acknowledged and the request is a read. The request is saved in the register when this request is a write request and the controller has to wait for all data to be written in the data buffer. The *all_data_present* signal is set to high when all this data is written in the data buffer.

The logic that will write all data in the data buffer has some similarities as the returning of data part of the controller. The scheme is given in figure 6.4.5.

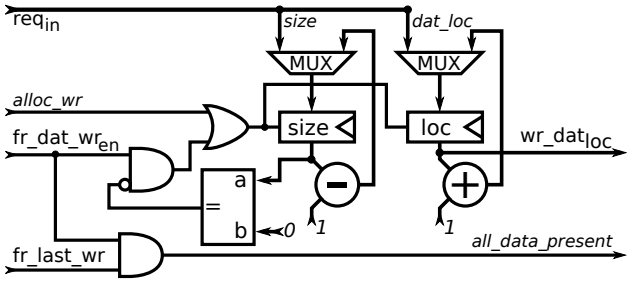


Figure 6.4.5: The part of the in/out control block that writes the data in the data buffer before a write request is put in the optimization block.

The size and data location of the request are loaded in the registers when the writing request is allocated. The used size is decremented and the data location is incremented for each data transfer that follows by the two adders. The allocation part is signaled that all data is written when the used size has reached zero and the last data element is written to the data buffer.

6.5 Summary

In this chapter the hardware implementation of the memory controller was described. The most important blocks like the sorting array and data buffer were described in more detail.

7 Results and Scalability

The results of the implementation in hardware with several configurations are given in this chapter. The sorting array size is varied to investigate the differences in performance. All implementations with different sorting array sizes are compared to an implementation where no reordering is done for the bandwidth, power and energy analysis. The results of the bandwidth and the speed-ups that the reordering implementation is getting is given in section 7.1. How much power the controller is using and how much energy is saved totally is described in section 7.2. The scalability is described with the synthesis results in section 7.3.

Three different input generator types are used, a 3D Fast Fourier Transform (3D-FFT) application, random read and writes, and a Conjugate Gradient Method (CG). Especially the 3D-FFT and CG are simulating a real life situation where the memory controller can be tested in. More is described about these generators in section A.1, while the result gathering is described in section A.2.

7.1 Maximum Bandwidth and Speed-Ups

The maximum bandwidth and speed-ups that the implementation is gaining is given in this section. These results are given for three different input generators and several sorting array sizes, as described earlier in this chapter. The maximum bandwidth is assumed to be the case where the DRAM is only reading and writing. The percentage of the maximum bandwidth is therefore given by equation 7.1.1.

$$bandwidth_{max} = \frac{\#data}{\#clk_{min} * freq} \tag{7.1.1a}$$

$$bandwidth_{used} = \frac{\#data}{\#clk_{used} * freq} \tag{7.1.1b}$$

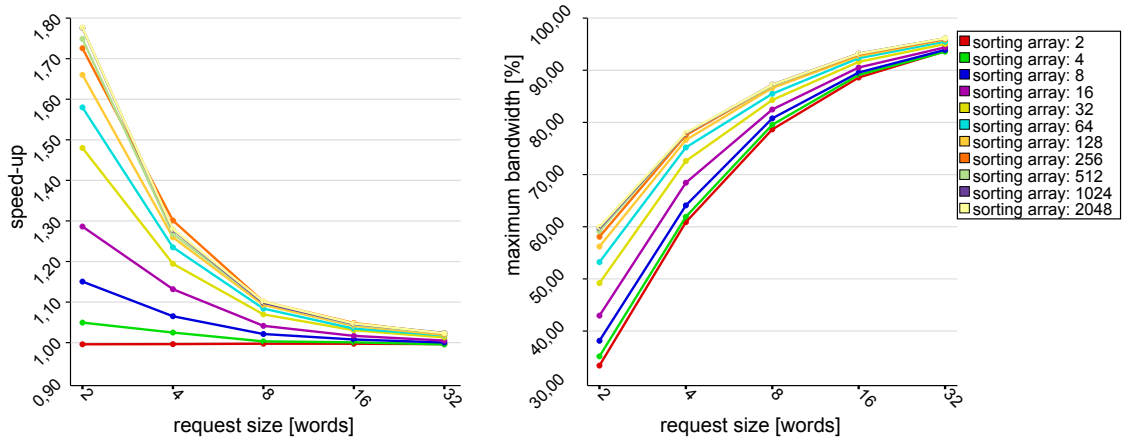
$$\%bandwidth_{max} = \frac{bandwidth_{max}}{bandwidth_{used}} * 100\% = \frac{\#clk_{min}}{\#clk_{used}} * 100\% \tag{7.1.1c}$$

The speed-up is simply calculated by normalizing needed amount of clock cycles for the reordering implementation by the amount of clock cycles for a similar implementation without a reordering policy. The results of these speed-ups are given in section 7.1.1 while a short summary of these results is given in section 7.1.2.

7.1.1 Results for Several Sorting Array Sizes and Inputs

The speed-up and percentage of maximum bandwidth results for the 3D-FFT are given in figure 7.1.1. These results show a large improvement for the smaller request sizes where the used bandwidth is around 50% of the maximum bandwidth. The speed-up is

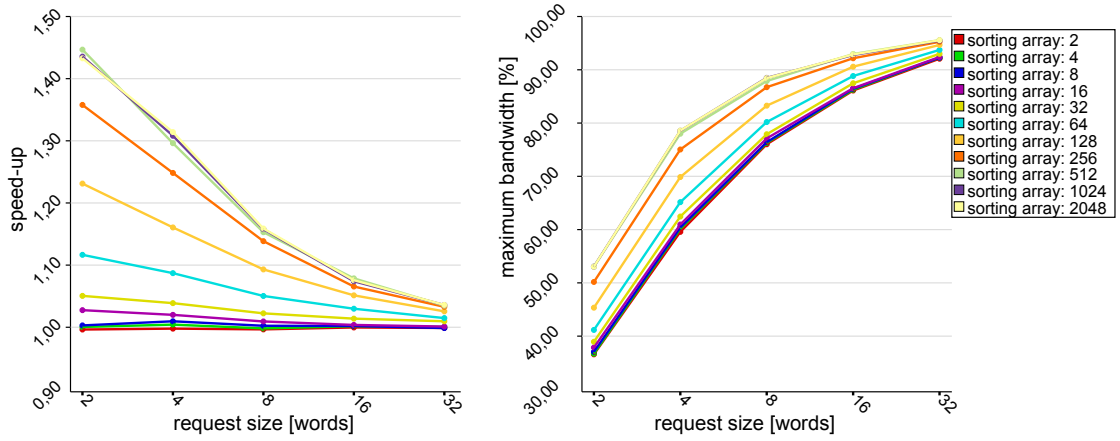
less when the used bandwidth reaches the 90% of the maximum bandwidth, but stays positive. It is easily seen that the speed-ups for sorting array sizes larger then 128 entries show no mentionable improvement anymore.



(a) speed-up (b) percentage of maximum bandwidth

Figure 7.1.1: The speed-up and the percentage of the maximum bandwidth with a 3D-FFT as input.

The speed-up for larger request sizes degrades less for the random read and writes and lies lower then the 3D-FFT input case. This can be seen in figure 7.1.2. A mentionable improvement starts from a sorting array size of 64 entries, compared to a sorting array size of 4 entries with the 3D-FFT.

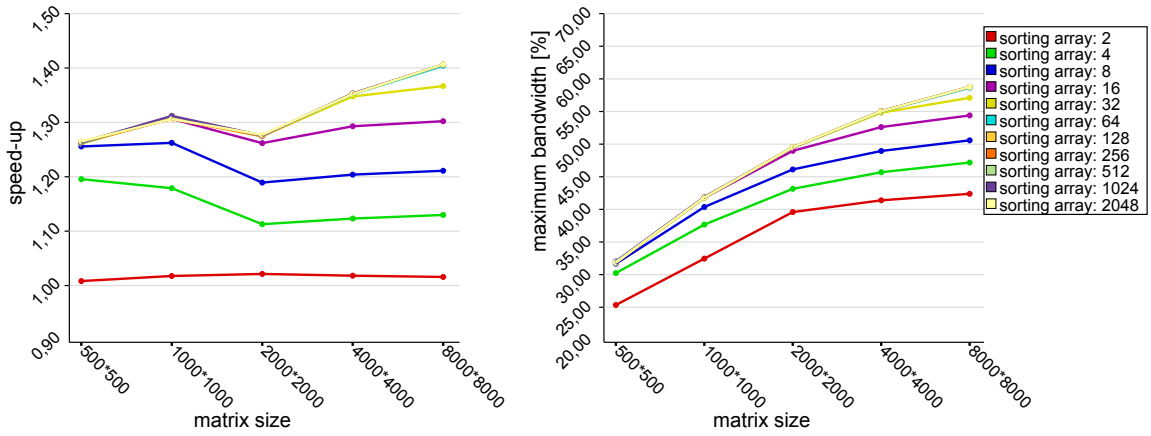


(a) speed-up (b) percentage of maximum bandwidth

Figure 7.1.2: The speed-up and the percentage of the maximum bandwidth with random read and writes as input.

The request sizes are all the same for the CG input generation, only the size of the used matrix is varying. The request size for this input generation is always 2 words, the

size of a float number. This is an inefficient size for the DRAM controller because it has a minimum burst size of 4 [25]. The DRAM controller has to wait for this minimum amount of transfers before an other bank or row can be accessed, which will decrease the used bandwidth a lot.



(a) speed-up (b) percentage of maximum bandwidth

Figure 7.1.3: The speed-up and the percentage of the maximum bandwidth with CG as input.

The speed-ups increase up to a sorting array size of 32 entries for several matrix sizes. Any implementation with a larger sorting array size does not show an extra improvement as can be seen in figure 7.1.3.

7.1.2 Summary

A speed-up is achieved by reordering the requests for all inputs and sorting array sizes. A larger sorting array size is necessary in the case of random read and writes as input before a mentionable speed-up is gained, compared to the 3D-FFT and CG inputs. There is always an improvement, although the speed-up decreases for larger request sizes. In table 7.1 a comparison is made between the input generators for sorting array sizes of 64 and 256 entries.

	Maximum speed-up	Average speed-up
3D-FFT	1.58	1.19
Random R/W	1.12	1.06
CG	1.40	1.32

(a) Sorting array size of 64 entries

	Maximum speed-up	Average speed-up
3D-FFT	1.73	1.24
Random R/W	1.36	1.17
CG	1.41	1.32

(b) Sorting array size of 256 entries

Table 7.1: The maximum and average speed-up that is achieved by the implementation with reordering normalized with an implementation without reordering.

7.2 Power Usage and Energy Savings

The power that the memory controller consumes and how much energy it is saving is described in this section. The OpenCores back-end is not considered in this power analysis. This is a component that is always needed for accessing the DRAM and is not changed when the optimization block and bridge are attached. A document from Micron with the necessary equations for the power calculation of the DRAM is used [18]. The equations that are used for this calculation are given in appendix B.2.

The analysis on the power consumption of the controller is done with PrimeTime, shortly described in section 7.2.1. The results of the power consumption of the controller is given in section 7.2.2 while the energy savings are described in section 7.2.3.

7.2.1 Power Analysis on the Controller

The power analysis on the controller is done with PrimeTime. This is done by first synthesizing the design into a netlist with Synopsys Design Compiler. ModelSim is then able to generate an activity file with this netlist and a testbench that is also used for the speed-up simulations. The activity file that is generated by ModelSim is then analyzed by PrimeTime and the needed values are extracted. All power results of the controller are based on a *90nmSP* process.

7.2.2 Power Usage by the Controller

The power that is used by the controller is divided by the blocks inside the controller. The size of the data buffer and sorting array are varied to see how they scale in power. They are also both the main contributor to the total power dissipation. The results where the sorting array is varies are given in section 7.2.2.1. The data buffer is also varied and these results are given in section 7.2.2.2.

All power results of the controller use the same input file generated from the 3D-FFT input generator. A fair comparison can be made between the different configurations, even though the activity may vary with a different input file.

7.2.2.1 Varying Sorting Array Size

Other blocks of the controller do also dissipate power, besides the sorting array and the data buffer. The results for the blocks that do not vary in size when the sorting array is changed are given in table 7.2. The data buffer is also added to this table, as it is not varied in this section.

	Power
data buffer (256 entries)	$1.56 - 1.80\ mW$
input/output control	$0.23 - 0.32\ mW$
bridge	$0.11 - 0.13\ mW$
other logic	$0.33 - 0.56\ mW$

Table 7.2: The power that is consumed by the blocks that do not vary in size when the size of the sorting array is changed.

The used power by the other components slightly varies with an increasing sorting array size due to different activity. Only the total power of the data buffer is put in the table, how much of this power is part of the allocation is given in section 7.2.2.2.

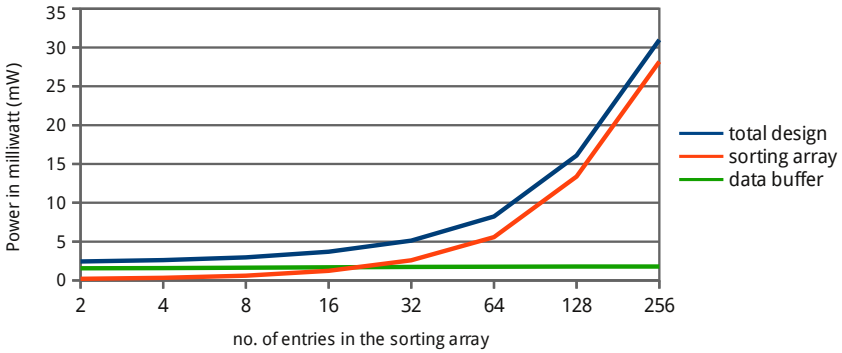


Figure 7.2.1: The power that is used by the sorting array, data buffer, and total controller for several sorting array sizes.

The power of the sorting array is increased slightly lower then two times when the size is doubles, this can be seen in figure 7.2.1. The power of the sorting array is higher then the data buffer (with 256 entries in the data buffer) after just 16 entries in the sorting array. More control logic per entry and more activity per entry are causing this difference.

7.2.2.2 Varying Data Buffer Size

The power that is consumed by the blocks that do not vary when the size of the data buffer is changing is given in table 7.2. These components are also main contributors to the total power dissipation, besides the data buffer. The sorting array is also added to this table, as it is not varied in this section.

	Power
sorting array (64 entries)	$5.66 - 5.82\ mW$
input/output control	$0.28 - 0.31\ mW$
bridge	$0.12\ mW$
other logic	$0.41 - 0.46\ mW$

Table 7.3: The power that is consumed by the blocks that do not vary when the size of the data buffer is varying.

The used power by the other components slightly varies with an increasing data buffer size due to different activity. This variation is less then when the sorting array size is varied. A reason for this is that the data buffer contributes less to the activity in other components then the sorting array.

The results for the power consumption with a varying data buffer size is given in figure 7.2.2. The power that is used by the allocation is also visible in this figure and is $\pm 27\%$ of the data buffer power. The power that is dissipated in the data buffer increases

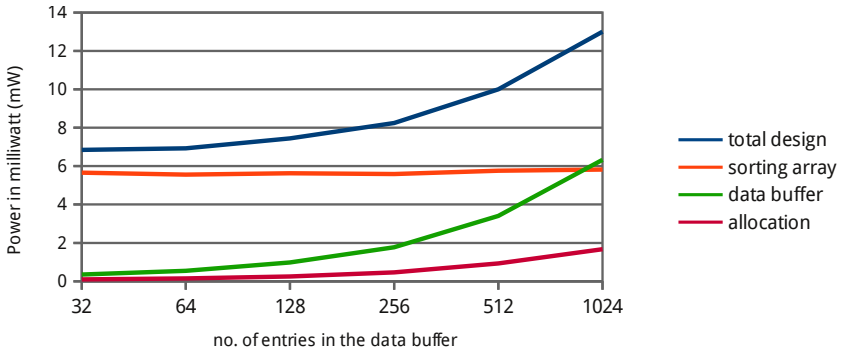


Figure 7.2.2: The power that is used by the sorting array, data buffer, allocation part of the data buffer, and total controller for several data buffer sizes.

less then two times when the data buffer size is doubled. A possible reason is that the activity in each data entry is not the same and thus the total activity in the data buffer does not double when the size is doubled.

7.2.3 Energy Saving by the Controller

The power that the DRAM is using during operation is calculated with the equations from the sheet from Micron [18] and the simulation results of the hardware implementation. With these simulations a varying sorting array size and a static data buffer size of 256 elements is used.

The activation and total power of the DRAM with the controller using a 3D-FFT input is shown in figure 7.2.3. These results are normalized with the results of the implementation that has no reordering policy implemented.

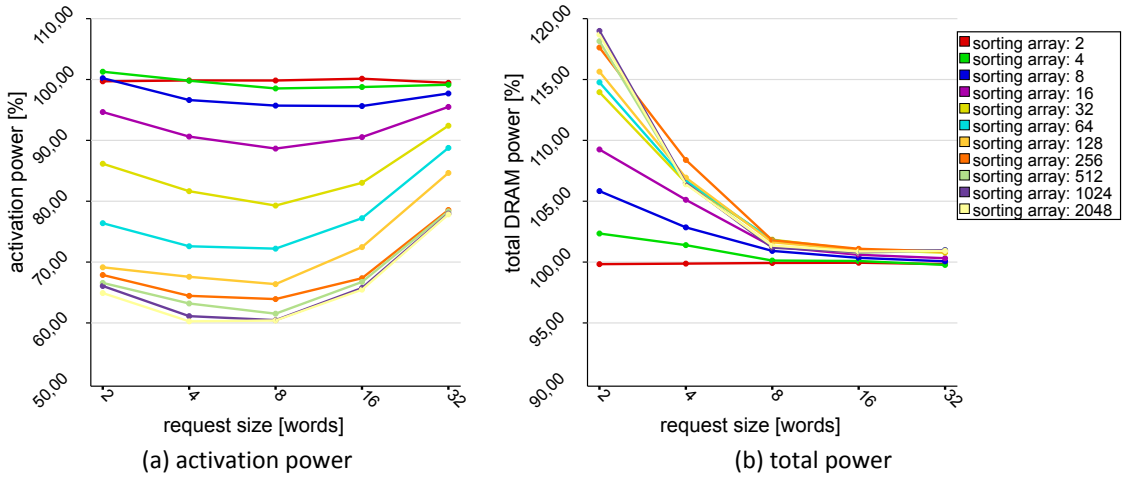
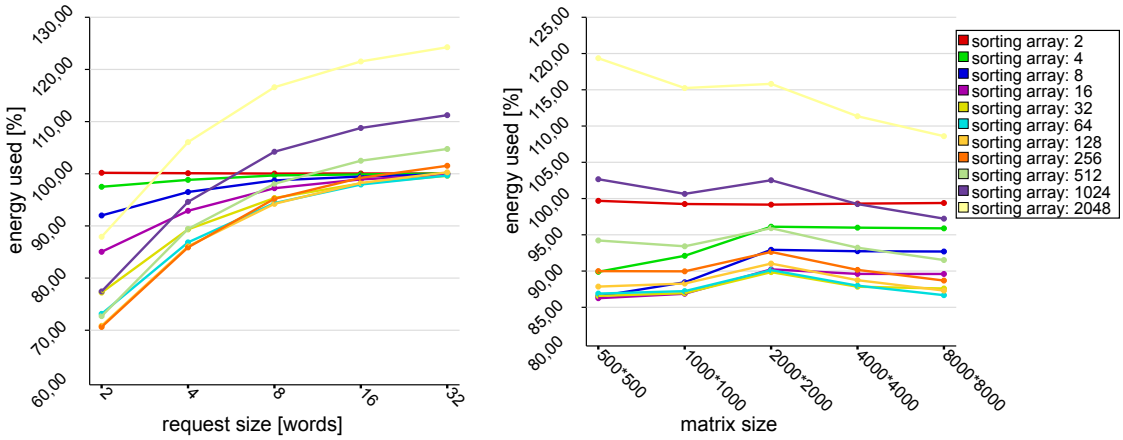


Figure 7.2.3: The reduction in activation power and raise of the total power for a DRAM. The DRAM is connected to the controller with a 3D-FFT as input.

The amount of power that is needed for the activation of the rows in the DRAM

is reduced up to 40% when the requests are reordered. The total power is increasing with the reordering policy implemented however, because more reads and writes are performed per second. Each read and write costs energy, and thus the power increases when more reads and writes are performed. In figure 7.2.4 is displayed what the total used energy is compared to an implementation without a reordering policy.



(a) 3D-FFT (b) CG

Figure 7.2.4: The energy that is used by the controller and the DRAM with a reordering policy compared the implementation without reordering policy.

The energy that is used to finish an application like the 3D-FFT and CG is less when a reordering policy is used. Only for large sorting array sizes the controller is using more energy with the reordering policy implemented. The consumed energy is less because the time to finish an application is decreased and thus less energy is consumed in the DRAM.

A short summary of the energy savings for an implementation with a sorting array size of 64 entries is given in table 7.4. This shows that a memory controller with a reordering policy implementation saves also energy, besides the speed-up that is gained.

	Minimum saving	Maximum saving	Average saving
3D-FFT	0.1%	26.6%	9.4%
Random R/W	0.1%	5.7%	2.4%
CG	9.8%	13.2%	12.1%

Table 7.4: The maximum, minimum, and average energy savings for an implementation with a sorting array size of 64 entries and several input generators.

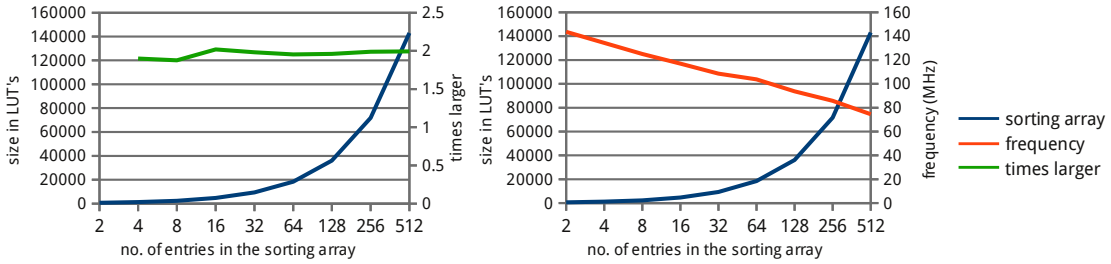
7.3 Synthesis Results

The synthesis results of the memory controller are described in this section. These results are useful to see how the area and frequency is scaling when the size of the sorting array or data buffer is changed. For this synthesis only the optimization block

and bridge are used and not the OpenCores back-end, just like the power analysis. In section 7.3.1 are the synthesis results given of the controller with various sorting array sizes. The synthesis results of the controller with various data buffer sizes are given in section 7.3.2.

7.3.1 Varying Sorting Array Size

How the area of the sorting array and the total implementation scales when the amount of entries in the sorting arrays is varied is described in this section. The maximum frequency that is possible for this design is also measured, although the implementation is not focused on a small area or clock period. The results for the size and frequency scaling of just the sorting array is given in figure 7.3.1.

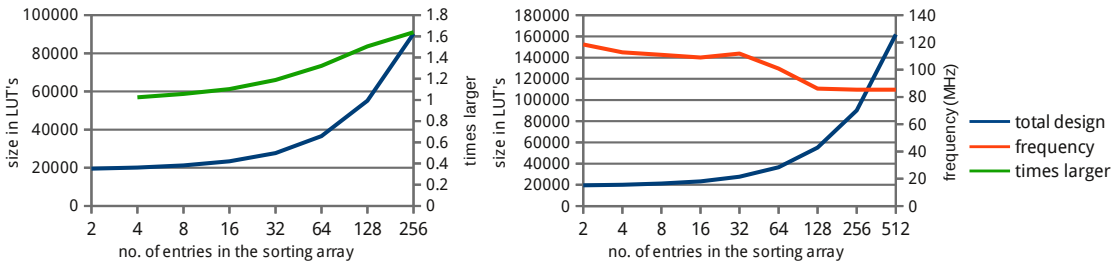


(a) size and scaling (b) size and frequency

Figure 7.3.1: The size and frequency scaling of the sorting array.

The area used by the sorting array is increasing with a constant scaling of 2 when the number of entries in the sorting array is doubled. Each entry in the sorting array has no hardware dependency on the amount of entries, and thus does not change in area. Only the output selecting, by tri-state bus or multiplexer, is subject of change in area. The frequency also shows that there is no dependency of the entries on the amount of entries in the sorting array. A normal small decrement in frequency is visible when the number of entries in the sorting array is doubled.

How the area and frequency of the total design are scaling along with the sorting array is made visible in figure 7.3.2. The number of entries in the data buffer is set constant to 256 entries in this implementation.



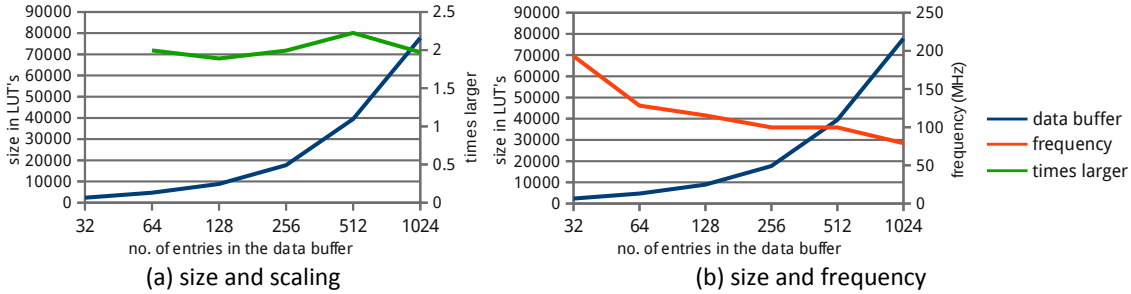
(a) size and scaling (b) size and frequency

Figure 7.3.2: The size and frequency scaling of the total design with various sorting array sizes.

The scaling of the area slowly increases as the size of the sorting array becomes the dominant factor in the design. For the frequency something like that is also happening, the frequency stays fairly constant as long as the slowest path is not in the sorting array.

7.3.2 Varying Data Buffer Size

The area and frequency related to the total amount of entries in the data buffer is described in this section. The results of the area and frequency of just the data buffer is given in figure 7.3.3.

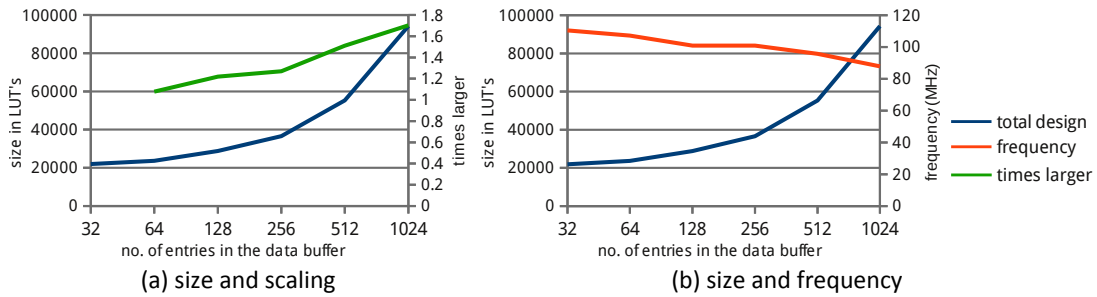


(a) size and scaling (b) size and frequency

Figure 7.3.3: The size and frequency scaling of the data buffer.

The area of the data buffer is increasing with a similar scaling as the sorting array, doubling the number of entries in the data buffer will roughly double the area. The calculation of the next free block in the data buffer as was described in section 6.2.2 is related to the amount of entries in the data buffer. The frequency is decreasing as expected when the number of entries in the data buffer is increased. The blocks for the allocation calculation will increase one bit for each doubling, which results in a small delay increment.

The results of the total design with the number of entries in the data buffer varying is made visible in figure 7.3.4. The number of entries in the sorting array is set constant to 64 entries in this implementation.



(a) size and scaling (b) size and frequency

Figure 7.3.4: The size and frequency scaling of the total design with various data buffer sizes.

The scaling of the area slowly increases as the size of the data buffer becomes a more dominant factor in the design. The frequency is slowly decreasing as the number

of entries in the data buffer is increasing, but the sorting array is more dominant in the frequency when the data buffer has 256 entries or less.

Conclusions and Future Work

8

This chapter describes the conclusions and possible future work for increasing the performance and functionality of the memory controller. The conventional way of accessing the DRAM where requests are issued in order is leading to a used bandwidth that cannot reach the theoretical maximum. The performance gap between the processors and memory has become larger and larger in the past few years. In this thesis a solution is proposed that reorders the requests in the memory controller to access the DRAM in a more efficient way that minimizes the stalling row changes.

The conclusions about the memory controller are described in section 8.1. A few propositions on future work directions are given in section 8.2 that could improve the performance of the memory controller.

8.1 Conclusions

In this thesis several reordering policies have been considered for the reordering of requests. These policies describe where a request has to be inserted and which request should be issued to the back-end. How the memory consistency can be guaranteed with the reordering has also been described in this thesis. The checking for possible memory inconsistencies costs extra hardware but is necessary, applications cannot rely on the data if this check is not implemented.

The data buffer will allocate the space for each request using one of the allocation policies that have been considered in this thesis. The chosen policy, that was described in section 4.2.3, is able to allocate a space in one cycle by looking only to the next free block.

The reordering policy that is implemented in hardware is the *smallest address first* policy with a dynamic boundary described in section 4.1.3. This policy showed the best expected results in the preliminary analysis where the several policies where compared. The expected hardware costs for this policy were also lower then the *largest block first* policies.

The implementation of the memory controller with the reordering policy inside is done in a modular way, as was proposed as a solution. Easy integration in systems with a different bus- or memory interface is possible due to the structure with a front-end, optimization block, and back-end. The optimization block itself is also implemented in a modular way. Different implementations of the sorting array or data buffer can be used without the need of redesigning other blocks. This modularity makes the design flexible and easy to use.

The results of the hardware simulations showed a speed-up when requests are reordered. This speed-up is visible for several applications with different configurations. The speed-up that is obtained by reordering request varies, but is up to 1.58 for a 3D-FFT

and 1.40 for a CG application with just 64 entries in the sorting array. The maximum theoretical bandwidth is still not reached, but further approached with this memory controller.

The synthesis results show that the area scales linear with the amount of entries in the sorting array and data buffer. Doubling the number of entries in the sorting array or data buffer will increase the area with a factor around 2.

A power analysis is also done in this thesis on the optimization block of the memory controller and the DRAM. This shows that the power that is used for the sorting array and the data buffer linear increases with the number of entries. The sorting array uses the majority of the power with a memory controller configuration of 64 entries in the sorting array and 256 entries in the data buffer. This configuration is also used with the speed-up simulations.

The power analysis also shows that the activation power of the DRAM is lowered up to 40% because of the reordering. The total power is increased because of the increased amount of read and writes per second which costs energy. The total energy that is necessary for each application is lowered however because of the speed-up. A maximum energy saving of 26.6% for the 3D-FFT and 13.2% for the CG application is possible by using this memory controller. This energy saving is smaller or there is no energy saved at all when a configuration with a larger sorting array is used.

In this thesis is shown that the used bandwidth of a DRAM is increased and that the total amount of energy is reduced by reordering requests.

8.2 Future Work

The memory controller as proposed in this thesis provides an easy way to change or integrate new features. In this section some propositions are made that can be used for future work to improve the performance.

An improvement in the bandwidth on the system bus can be made by using shadow addressing and translation as proposed by Carter et al. [7]. A translation block should then be added in front of the sorting array plus a controller that can track all translated requests. This control block can then decide when an original request is done and can be returned to the front-end. This means some small extra edits on the controller as it is now, but blocks like the in/out control, bridge, data buffer, and sorting array do not need any changes.

The implementation as discussed in this thesis is not focusing on the power consumption, as described earlier. Redesigning the controller with extra focus on this point will probably save more energy. At the moment all entries in the sorting array will check if the stored requests would overlap for example, even when there is request present. Clock gating and the ability to switch of parts of the design would decrease the power consumption.

A double data rate or multiple channel version can perform differently in performance. A small research can be done in how reordering between multiple channels can be done efficiently without loosing the memory consistency.

Bibliography

[1] S.V. Adve and K. Gharachorloo, *Shared memory consistency models: a tutorial*, Computer **29** (1996), no. 12, 66 --76.

[2] A. Arm, *Axi protocol specification*, mar. 2004, v1.0.

[3] P. Athe and S. Dasgupta, *A comparative study of 6t, 8t and 9t decanano sram cell*, Industrial Electronics Applications, 2009. ISIEA 2009. IEEE Symposium on, vol. 2, oct. 2009, pp. 889 --894.

[4] Prof. David August, *Inner workings of malloc and free*, http://www.cs.princeton.edu/courses/archive/fall06/cos217/lectures/14Memory-2x2.pdf, fall 2006, Consulted on 27th of September 2011.

[5] Michael A. Bender, Martin Farach-Colton, and Miguel Mosteiro, *Insertion sort is O(n log n)*, 2004.

[6] Shekhar Borkar and Andrew A. Chien, *The future of microprocessors*, Commun. ACM **54** (2011), 67--77.

[7] J. Carter, W. Hsieh, L. Stoller, M. Swanson, Lixin Zhang, E. Brunvand, A. Davis, Chen-Chi Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama, *Impulse: building a smarter memory controller*, High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On, jan. 1999, pp. 70 --79.

[8] J.M. Chang and E.F. Gehringer, *A high performance memory allocator for object-oriented systems*, Computers, IEEE Transactions on **45** (1996), no. 3, 357 --366.

[9] C. Ciobanu, X. Martorell, G. Kuzmanov, A. Ramirez, and G. Gaydadjiev, *Scalability evaluation of a polymorphic register file: A cg case study*, Architecture of Computing Systems-ARCS 2011 (2011), 13--25.

[10] A. Cosoroaba, *Memory interfaces made easy with xilinx fpgas and the memory interface generator*, Xilinx WP260 (V1. 0) (2007).

[11] M.D. Hill, *Multiprocessors should support simple memory consistency models*, Computer **31** (1998), no. 8, 28 --34.

[12] Y.C. Hu, A. Cox, and W. Zwaenepoel, *Improving fine-grained irregular shared-memory benchmarks by data reordering*, Supercomputing, ACM/IEEE 2000 Conference, nov. 2000, p. 33.

[13] Infineon Technologies North America Corporation and Kingston Technology Company, Inc., *Intel dual-channel ddr memory architecture white paper*, Kingston Technology, sep. 2003.

[14] B. Keeth, *DRAM circuit design: fundamentals and high-speed topics*, vol. 13, Wiley-IEEE Press, 2008.

[15] Tom Leighton, *Tight bounds on the complexity of parallel sorting*, Computers, IEEE Transactions on **C-34** (1985), no. 4, 344 --354.

[16] Tzu-Chieh Lin, Kun-Bin Lee, and Chein-Wei Jen, *Quality-aware memory controller for multimedia platform soc*, Signal Processing Systems, 2003. SIPS 2003. IEEE Workshop on, aug. 2003, pp. 328 -- 333.

[17] R. Marcelino, H.C. Neto, and J. Cardoso, *A comparison of three representative hardware sorting units*, Industrial Electronics, 2009. IECON ' 09. 35th Annual Conference of IEEE, nov. 2009, pp. 2805 --2810.

[18] Micron, *SDRAM System-Power Calculator*, http://www.micron.com/get-document/?documentId=31, apr. 2001, Consulted on 14th of November 2011.

[19] Micron Technology, Inc., *General DDR SDRAM Functionality*, Tech. report, 2001, TN-46-05.

[20] G.E. Moore et al., *Cramming more components onto integrated circuits*, Proceedings of the IEEE **86** (1998), no. 1, 82--85.

[21] J.C. Pichel, D.E. Singh, and J. Carretero, *Reordering algorithms for increasing locality on multicore processors*, High Performance Computing and Communications, 2008. HPCC ' 08. 10th IEEE International Conference on, sep. 2008, pp. 123 --130.

[22] B. Prince and D.B. Prince, *High Performance Memories: New Architecture DRAMs and SRAMs--Evolution and Function*, Wiley, 1999.

[23] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, and J.D. Owens, *Memory access scheduling*, Computer Architecture, 2000. Proceedings of the 27th International Symposium on, jun. 2000, pp. 128 --138.

[24] R. Sandeep, N.T. Deshpande, and A.R. Aswatha, *Design and analysis of a new loadless 4t sram cell in deep submicron cmos technologies*, Emerging Trends in Engineering and Technology (ICETET), 2009 2nd International Conference on, dec. 2009, pp. 155 --161.

[25] Denis Shekhalev, *High Speed SDRAM Controller With Adaptive Bank Management and Command Pipeline*, http://opencores.org/project,hssdrc, dec. 2009.

[26] IBM Systems and Technology Group, *128-bit processor local bus architecture specifications*, may 2007, v4.7.

[27] Y. Takefuji, K.-C. Lee, and T. Tanaka, *A two step sorting algorithm*, Neural Networks, 1990., 1990 IJCNN International Joint Conference on, jun. 1990, pp. 793 --798 vol.3.

[28] T. Watanabe, K. Ayukawa, S. Miura, M. Toda, T. Iwamura, K. Hoshi, J. Sato, and K. Yanagisawa, *Access optimizer to overcome the ' future walls of embedded DRAMs' in the era of systems on silicon*, Solid-State Circuits Conference, 1999. Digest of Technical Papers. ISSCC. 1999 IEEE International, 1999, pp. 370 --371.

[29] Wm. A. Wulf and Sally A. McKee, *Hitting the memory wall: implications of the obvious*, SIGARCH Comput. Archit. News **23** (1995), 20--24.

Methodology

A

How the inputs for the simulations in this are generated and how the results are gathered is described in this appendix. The input generators with a small description on the simulated application are described in section A.1. The result gathering is described in section A.2.

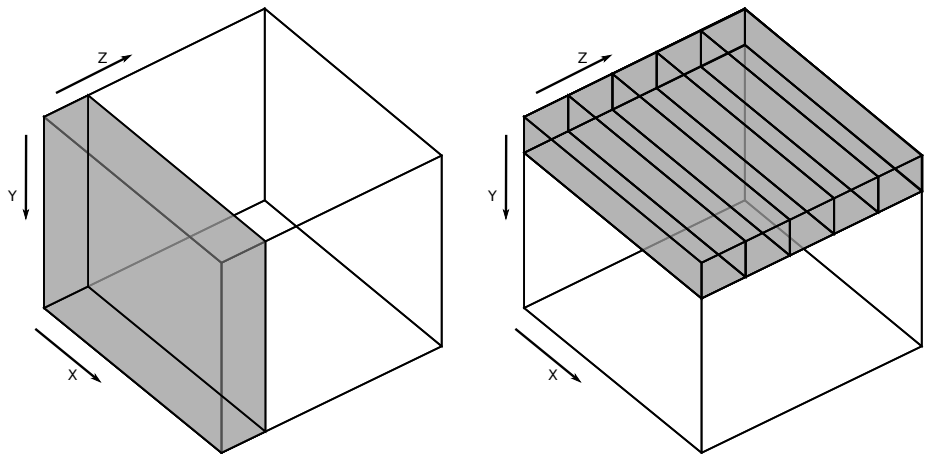
A.1 Input Generators

Three different type of input applications have been used for the simulations. A 3D-FFT application, random read and writes, and a Conjugate Gradient (CG) application. The generators all write the requests to a file that is used as input for the software simulators and VHDL testbench.

A.1.1 3D-FFT Application

A 1D fast fourier transform has to be performed on each dimension of a 3D mesh. This 3D FFT can thus be summarized as equation A.1.1.

$$f(k_x, k_y, k_z) = \sum_z \left[\sum_y \left[\sum_x f(x, y, z) * e^{ik_x x} \right] * e^{ik_y y} \right] * e^{ik_z z} \qquad \text{(A.1.1)}$$



(a) XY plane (b) XZ plane with the stripes

Figure A.1.1: The 3D mesh that is used for the 3D FFT calculation. The gray parts are those that CPU 1 will use as an example. Multiple (small) requests have to be issued for the xz-plane.

The data that has to be transferred to the processors can be visualized in a 3D mesh, like figure A.1.1. The XY-planes are transferred first to the processors and written back to the memory after calculating. The XZ-planes are then transferred to the processors. The maximum size that is possible for the requests when the XZ-planes are transferred lies lower then for the XY-planes.

A.1.2 Random Read and Writes

The random read and write generation is fairly simple. Each request is generated with a pre-divined size, a random address in a data set, and randomly chosen read or write flag. The data set size is also pre-divined and the amount of data that is transferred is each generation the same. This means that with requests that have a larger size, less request are generated. This leads to a pseudo code listed in listing A.1.

```
i = (rand() % data_set_size)<<2 % data_set_size;
while (count < Transf_size)
{
  cpu_addr = cpu_base_addr + i;
  mem_addr = mem_base_addr + i;

  if(rand()%2==0)
    create_req(cpu_addr, mem_addr, req_size, WR);
  else
    create_req(cpu_addr, mem_addr, req_size, RD);

  i = (rand() % data_set_size)<<2 % data_set_size;
  count ++;
}
```

Listing A.1: Pseudo code for the random read and writes.

A.1.3 CG Application

The CG application generator is based on the computation kernel, the Sparse Matrix - Dense Vector Multiplications. This accounts for $\pm 87\%$ of the total execution time [9]. The sparse matrices are stored in a CSR format. The pseudo code of this kernel is given in listing A.2.

```
for (j = 1; j <= number_of_rows; j++) {
  w[j] = 0.0;
  lowk = rowstr[j]; upk=rowstr[j+1];
  for(k = lowk; k < upk; k++) {
    w[j] += a[k]*p[colidx[k]];
  }
}
```

Listing A.2: Pseudo code for the CG kernel.

The requests lie sparsely distributed in the memory with this application, multiple rows and banks are used when the matrix sizes are large enough. Multiple request have to be issued to access all this data.

A.2 Result Gathering

The results of the simulations are gathered by scripts that automate the simulation process. Those scripts transfer the results to a website where the numbers are put in a database. Easy and quick generation of the SVG-graphs is then possible by a user-friendly interface which is displayed as a screenshot in figure A.2.1. This website has been created specially for this thesis and made it possible to combine all the results of multiple computers easily.

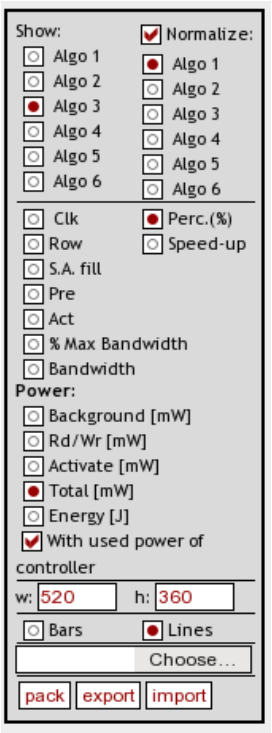


Figure A.2.1: Screenshot of the graph generation menu

Some more information about the simulation script is described in section A.2.1.

A.2.1 Simulation Script

The script will announce to the website that a simulation is started. In exchange it will receive a confirmation on the simulation key, this key is used to identify the results to the appropriate simulation. A input file is generated and the simulators are started for a few iterations per configuration. The results that those simulators are giving are written to a file where all individual values are extracted from. Those values are then written to the website where they will be stored in a MySQL database.

```
key_h=$(date +%s)

body="HW CG SA: $sort"
subject="0#$key_h#$(echo -n "0&$key_h&shared key" | sha1sum)"
```

```
key_h=‘curl "http://www.imageportfolio.nl/simulaties/wget_insert.php"
    -d "subject=$subject" -d"body=$body" -g‘

#CG:
size_start=500;
size_end=8000;

for (( mrqs=size_start; mrqs<=size_end; mrqs=mrqs*2 ))
do
  for ((i=0;i<=2;i++))
  do
    ./gen_CG_mem_req -nc $mrqs -nr $mrqs -ds 4 -nzp 1

    rm h1.dat,h2.dat,h3.dat,h6.dat;

    vsim -do "set sort $sort; do ../1/sim_memControl_OC_file.do" -lib
        "../1/work" -quiet -c>h1.dat&
    vsim -do "set sort $sort; do
        ../2/sim_memControl_OC_file_no_dyn.do" -lib "../2/work" -quiet
        -c>h2.dat&
    vsim -do "set sort $sort; do ../3/sim_memControl_OC_file_dyn.do"
        -lib "../3/work" -quiet -c>h3.dat&
    vsim -do "set sort $sort; do
        ../6/sim_memControl_OC_file_perBank.do" -lib "../6/work" -quiet
        -c>h6.dat&
    wait
    echo "Getting results and curl them to the website..."

    #hardware simulation 1:
    echo ‘gawk -F% ’{ print $2 }’ h1.dat‘ > h1.dat
    algo=‘gawk -F: ’{ print $1 }’ h1.dat‘
    others=‘gawk -F: ’{ print $2 }’ h1.dat‘
    clk=‘echo "$others"|gawk -F’ ’ ’{ print $1 }’‘
    row=‘echo "$others"|gawk -F’ ’ ’{ print $2 }’‘
    wr=‘echo "$others"|gawk -F’ ’ ’{ print $3 }’‘
    act=‘echo "$others"|gawk -F’ ’ ’{ print $4 }’‘
    pre=‘echo "$others"|gawk -F’ ’ ’{ print $5 }’‘
    mini=‘echo "$others"|gawk -F’ ’ ’{ print $6 }’‘
    rd=‘echo "$others"|gawk -F’ ’ ’{ print $7 }’‘
    wr=‘echo "$others"|gawk -F’ ’ ’{ print $8 }’‘
    precnt=‘echo "$others"|gawk -F’ ’ ’{ print $9 }’‘
    precke=‘echo "$others"|gawk -F’ ’ ’{ print $10 }’‘
    actcnt=‘echo "$others"|gawk -F’ ’ ’{ print $11 }’‘
    actcke=‘echo "$others"|gawk -F’ ’ ’{ print $12 }’‘

    body="$algo:$j:$i:$clk:$row:$wr:mrqs $mrqs ts
        $ts:$act:$pre:$mini:$rd:$wr:$precnt:$precke:$actcnt:$actcke"
    subject="1#$key_h#$(echo -n "1&$key_h&shared key" | sha1sum)"
    curl "http://www.imageportfolio.nl/simulaties/wget_insert.php" -d
        "subject=$subject" -d "body=$body" -g

    #hardware simulation 2:
                                        ⋮
  done
```

```
done
```

Listing A.3: Bash script for running the simulations

B Power Calculation

The power analysis is done with PrimeTime for the controller, shortly described in secion 7.2.1. The analysis on the DRAM is done by the equations extracted from a Excel sheet of Micron [18]. Some extra results on the controller are given in section B.1. The equations that are used for the power calculations of the DRAM are given in section B.2.

B.1 Power Results of the Controller

no. entries	2	4	8	16	32	64	128	256
total design (mW)	2.448	2.632	2.982	3.699	5.137	8.243	16.1	31
sorting array (mW	0.223	0.355	0.626	1.25	2.59	5.59	13.4	28.2
data buffer (mW)	1.56	1.6	1.65	1.7	1.74	1.77	1.79	1.8
inout control (mW)	0.229	0.235	0.246	0.261	0.279	0.301	0.308	0.32
bridge (mW	0.11	0.11	0.112	0.114	0.117	0.119	0.12	0.125
rest (mW)	0.326	0.332	0.348	0.374	0.411	0.463	0.482	0.555
% of total	13.32%	12.61%	11.67%	10.11%	8.00%	5.62%	2.99%	1.79%

Table B.1: Power results of the controller with a varying sorting array size. The size of the data buffer is fixed to 256 entries.

no. entries (mW)	32	64	128	256	512	1024
total design (mW)	6.845	6.93	7.442	8.243	10	13
sorting array (mW)	5.66	5.56	5.63	5.59	5.76	5.82
data buffer (mW)	0.359	0.548	0.98	1.77	3.41	6.33
in/out control (mW)	0.286	0.282	0.285	0.301	0.305	0.314
bridge (mW)	0.122	0.116	0.118	0.119	0.121	0.123
allocation (mW)	0.1003	0.1468	0.2584	0.4707	0.9330	1.6788
allocation of data buffer	27.94%	26.79%	26.37%	26.59%	27.36%	26.52%
rest (mW)	0.418	0.424	0.429	0.463	0.404	0.413
% of total	6.11%	6.12%	5.76%	5.62%	4.04%	3.18%

Table B.2: Power results of the controller with a varying data buffer size. The size of the sorting array is fixed to 64 entries.

B.2 DRAM Power Equations

The equation that have been derived from the excel sheet of Micron [18] have been put in the website (written in PHP). The function for calculation the power is given in listing B.1.

```
function
    power($bnks,$clk,$precnt,$precke,$actcnt,$actcke,$rd,$wr,$nracts)
{
  if($bnks==0||$clk==0||$precnt==0||$actcnt==0||$nracts==0)
    return array("Background"=>0,"Active"=>0,"RdWr"=>0,"Total"=>0);

  //device:
  $density=64;//M
  $nr_dqs=32;
  $speedgrade="-7";

  $Vcc_max=3.6;//V
  $Vcc_min=3;//V

  $Idd1=130;//mA
  $Idd2=2;//mA
  $Idd3=50;//mA
  $Idd4=160;//mA
  $Idd6=2;//mA

  $tCK=10;//ns
  $tRRD=14;//ns
  $tRC=70;//ns
  $mintCK=7;//ns

  $Idd0=$Idd1-($Idd4-$Idd3)*2*$tCK/$tRC;//mA

  //Usage:
  $Vcc=3.3;//V
  $freq=133.0;//MHz
  $load=25;//pF

  $perc_time_pre=($precnt/$bnks)/$clk;//%
  $perc_pre_cke=$precke/$precnt;//%
  $perc_act_cke=$actcke/$actcnt;//%

  $act_time=1000*($clk/$nracts)/$freq;//ns
  $perc_rd=$rd/$clk;//%
  $perc_wr=$wr/$clk;//%


  //Power Calcs:
    //worst case
  $pPRE_PDN=$Idd2*$Vcc_max;//mW
  $pPRE_STBY=$Idd3*$Vcc_max;//mW
  $pACT_PDN=$Idd2*$Vcc_max;//mW
  $pACT_STBY=$Idd3*$Vcc_max;//mW
  $pREF=($Idd6-$Idd2)*$Vcc_max;//mW
  $pACT=($Idd0-$Idd3)*$Vcc_max;//mW
```

```
  $pWR=($Idd4-$Idd3)*$Vcc_max;//mW
  $pRD=($Idd4-$Idd3)*$Vcc_max;//mW
  $pDQ=$Vcc_max*$Vcc_max/$tCK/2*$load*$nr_dqs;//mW

    //derated for system usage
  $pPRE_PDN_der=$pPRE_PDN*$perc_time_pre*$perc_pre_cke;//mW
  $pPRE_STBY_der=$pPRE_STBY*$perc_time_pre*(1-$perc_pre_cke);//mW
  $pACT_PDN_der=$pACT_PDN*(1-$perc_time_pre)*$perc_act_cke;//mW
  $pACT_STBY_der=$pACT_STBY*(1-$perc_time_pre)*(1-$perc_act_cke);//mW
  $pREF_der=$pREF;//mW
  $pACT_der=$pACT*$tRC/$act_time;//mW
  $pWR_der=$pWR*$perc_wr;//mW
  $pRD_der=$pRD*$perc_rd;//mW
  $pDQ_der=$pDQ*$perc_rd;//mW

    //scaled
  $vcc_scale=($Vcc/$Vcc_max)*($Vcc/$Vcc_max);

  $pPRE_PDN_scl=$pPRE_PDN_der*$vcc_scale;//mW
  $pPRE_STBY_scl=$pPRE_STBY_der*$vcc_scale*$freq/1000*$tCK;//mW
  $pACT_PDN_scl=$pACT_PDN_der*$vcc_scale;//mW
  $pACT_STBY_scl=$pACT_STBY_der*$vcc_scale*$freq/1000*$tCK;//mW
  $pREF_scl=$pREF_der*$vcc_scale;//mW
  $pACT_scl=$pACT_der*$vcc_scale;//mW
  $pWR_scl=$pWR_der*$vcc_scale*$freq/1000*$tCK;//mW
  $pRD_scl=$pRD_der*$vcc_scale*$freq/1000*$tCK;//mW
  $pDQ_scl=$pDQ_der*$vcc_scale*$freq/1000*$tCK;//mW


  //Totals:
  $P_Background=$pPRE_PDN_scl+$pPRE_STBY_scl+$pACT_PDN_scl+$pACT_STBY_scl;
  $P_Active=$pACT_scl;
  $P_RdWr=$pWR_scl+$pRD_scl+$pDQ_scl;
  $P_Total=$P_Background+$P_Active+$P_RdWr;

  $rd_bandwidth=$freq*$perc_rd*$nr_dqs/8;
  $wr_bandwidth=$freq*$perc_wr*$nr_dqs/8;
  $total_bandwidth=$rd_bandwidth+$wr_bandwidth;

  //Return:
  return array("Background"=>$P_Background, "Active"=>$P_Active,
      "RdWr"=>$P_RdWr, "Total"=>$P_Total,
      "rd_bandwidth"=>$rd_bandwidth, "wr_bandwidth"=>$wr_bandwidth,
      "total_bandwidth"=>$total_bandwidth);
}
```

Listing B.1: Power equations for the DRAM

C Sources

The sources of the C programs and VHDL hardware implementation can be found on the attached CD. A file structure of this CD is given below.

- C/
  - advanced_simulator/
    - back.c
    - main.c
    - simulate.c
    - structs.c
  - basic_simulator/
    - main.c
    - simulate.c
  - memory_request_generators/
    - gen_mem_req_3DFFT.c
    - gen_mem_req_CG.c
    - gen_mem_req_random.c
- VHDL/
  - bridge/
    - bridge.vhd
    - comp_bridge_pack.vhd
    - long_to_small_req.vhd
    - open_back_check.vhd
    - reg_pointer.vhd
    - short_req_buffer.vhd
    - track_data_unit.vhd
  - general_items/
    - components_pack.vhd
    - count_ones.vhd
    - fifo.vhd
    - functions_pack.vhd
    - increase_decrease.vhd
    - muxes.vhd

```
        pointer.vhd
        registers.vhd
        rotate.vhd
        sizes_pack.vhd
    OC_controller/
        core/
            mt48lc2m32b2.v
        doc/
        include/
            hssdrc_define.vh
            hssdrc_tb_sys_if.vh
            hssdrc_timescale.vh
            hssdrc_timing.vh
        rtl/
            hssdrc_access_manager.v
            hssdrc_addr_path_p1.v
            hssdrc_addr_path.v
            hssdrc_arbiter_in.v
            hssdrc_arbiter_out.v
            hssdrc_ba_map.v
            hssdrc_data_path_p1.v
            hssdrc_data_path.v
            hssdrc_decoder_state.v
            hssdrc_decoder.v
            hssdrc_init_state.v
            hssdrc_mux.v
            hssdrc_refr_counter.v
            hssdrc_top.v
        sim/
        testbench/
            hssdrc_driver_class.sv
            hssrdc_bandwidth_monitor_class.sv
            hssrdc_driver_cbs_class.sv
            hssrdc_scoreboard_class.sv
            message_class.sv
            sdram_agent_class.sv
            sdram_interpretator.sv
            sdram_transaction_class.sv
            sdram_tread_class.sv
```

tb_prog.sv
tb_top.sv
optimize/
addr_to_pointer.vhd
block_free_full.vhd
boundary.vhd
compare.vhd
comp_opt_pack.vhd
data_block.vhd
data_buffer.vhd
halt_process.vhd
inOutControl.vhd
next_bank_decision.vhd
optimize.vhd
overlap.vhd
reqLineControl.vhd
sort_control_block.vhd
sortingPerBank.vhd
sorting.vhd
trunk_ones.vhd
total/
memControl_OC_file.vhd
memControl_OC.vhd
memControl_OptBridge.vhd