Improving DRAM performance and energy efficiency

Bogdan Spinean*, Sander Geursen*, Georgi Gaydadjiev*†
† Department of Computer Science and Engineering, Chalmers University of Technology, Sweden
*Computer Engineering Laboratory, EEMCS, Delft University of Technology, The Netherlands
b.spinean@tudelft.nl, a.a.j.geursen@student.tudelft.nl, g.n.gaydadjiev@tudelft.nl

***Abstract*—In many core systems with shared DRAM memory a clear performance dissbalance exists between the requirements of the processors and the bandwidth that the memory system can provide. Very often the utilization of the memory interface is poor even for well understood and regular workloads. In this paper we propose a method to reorder the in-flight requests by the multiple processing elements at the level of the memory controller and significantly reduce the number of row changes in a fashion transparent to the workload, the CPU ISA or the bus protocol. Several reordering policies have been considered and after a thorough analysis one was selected for implementation. We have obtained application speedups of 1.58x for 3D FFT and 1.4x for Conjugate Gradient workloads. In addition, our reordering proposal reduces the activation power of the memory by up to 40% while the total energy reduction per application is 26.6% for 3D FFT and 13.2% for CG.**

I. Introduction

In this paper we study the influence of memory access reordering on performance and energy efficiency in the context of many core systems running workloads that access in parallel a block of data located in shared DRAM memory. The proposed memory access reordering does not require changes to the ISA, the bus or memory interfaces making it totally transparent to the overall system architecture.

The goal of this work is to propose a memory controller that will operate the DRAM as close as possible to the maximum theoretical bandwidth while also saving energy. We have opted for a modular design that can easily adapt to a large variety of buses or DRAM devices. Also, our optimization block is designed in such a way that both reordering policies and data buffer allocation policies can be easily changed without impacting the rest of the design. Several reordering policies have been considered and after a thorough preliminary analysis we have implemented the one with the best estimated performance-cost ratio. Reordering of memory accesses brings two complications. The first is memory consistency: we must ensure that accesses to the same memory location are executed in program order. This is aggravated by the fact that the memory requests can have any size and multiple requests might overlap either partially or completely. The second complication is data buffer management. Data is allocated in the data buffer in the order that the requests are received but the data is deallocated out of order. Since the memory requests have arbitrary sizes, the allocation policy must have flexible de-fragmentation capabilities. Our proposal:

- reduces memory transfer time by up to 1.4x for 3D-FFT and 1.5x for Conjugate Gradient;
- reduces the memory controller and DRAM total energy by 26.6% for 3D-FFT and 13.2% for CG;
- does not require changes in the architecture, bus interfaces or data mapping into the DRAM.

The paper is organized as follows: Section 2 presents the background and related work; Section 3 describes the general architecture of the proposed memory controller; In Section 4 we elaborate on the reordering policy, the data buffer allocation and we address memory consistency; Section 5 discusses our results and Section 6 presents our conclusions.

II. Background and Related Work

DRAM devices are organized typically in 4 independent banks, each bank is a matrix of rows×columns memory elements, each one byte in size [8]. To access a particular element first a row needs to be open, then within the open rows any number of column accesses can be performed. When accessing data on a different row, a precharge operation is required before a new row can be open. There is a single command interface and one data interface shared by all banks. While data is transferred to/from one bank, other banks can be executing commands that don't involve data transfers like row activation or precharge. The key to obtaining high performance is minimizing the time that the data interface is not transferring data because the DRAM is waiting for row changes or other timing constraints. For low power consumption it is important to minimize row changes.

Works related to improving memory bandwidth have approached the following areas: software improvements, operating system support, memory controller optimizations and DRAM chip optimizations. Pichel et al. [10] propose a software solution for efficient memory access in the case of sparse matrices. A technique for increasing the locality is introduced that consists on reorganizing the data at the application side without hardware changes. Another software based approach is proposed by [15] where a library is used abstract the memory bandwidth improvement techniques from the rest of the software application. The Impulse memory controller focuses on the improvement in the cache and bus utilization [6] and requires OS support. Certain regions in the address space can be mapped into an extended address space called shadow addresses. For instance, for a matrix

stored in virtual memory, its diagonal can be mapped in the shadow address space as a continuous region and thus the cache will not be polluted by unneeded data. The Impulse memory controller remaps the shadow addresses to the correct non-continuous data locations in the memory. Compared to the above approaches, our performance enhancements do not require any changes of software or OS. Optimizations that are localized in the memory controller have been proposed by Rixner [12] and Lin [7]. These two approaches target media applications while our proposed solution is application domain independent and and offer improved flexibility of busses and memory interfaces. Hitachi [14] proposes an access optimizer implemented in the same silicon as the DRAM cells. Langemeyer et al. [11] propose a novel mapping scheme that for 2D-FFT workloads that requires very frequent and predictable row changes that can be hidden while transferring data from different banks. Bandwidth is increased at the cost of using more energy. Compared to the above approaches, ours does not require any changes of the DRAM devices or in the address mapping while at the same time improving total energy used.

III. Proposed Solution

Figure 1 shows the general organization of the proposed memory controller. The Front-End handles communication with the bus interface, the Optimization Block reorders memory accesses, buffers the data and handles memory consistency while the Back-End translates memory requests int actual DRAM commands. The design is flexible and easy interchangeable with different bus interfaces and memories.

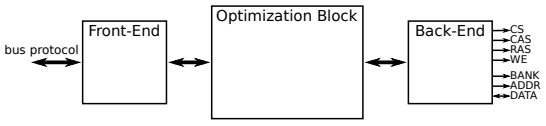


Fig. 1: The proposed organization.

The Front-End is built for buses resembling the AMBA AXI bus with the following bus interface features [1]: 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; the request size can range from 1B to 16kB and requests of any size can be interleaved.

The Back-End is used to ensure that different types of DRAM devices can be used with the optimization implemented in the controller. The Back-End is the part of the memory controller that controls the physical DRAM inputs like CAS, RAS, bank number and so on. The Back-End also enforces the DRAM timing like CL, t_{RCD}, t_{RC} and so on. It will also read data from and write data to the DRAM. The Optimization Block need not change when a different DRAM is used, a modified Back-End will suffice. The Back-End that we have used in our experiments is a DRAM controller from OpenCores [13]. 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. The stalls are avoided by issuing commands during data transfers.

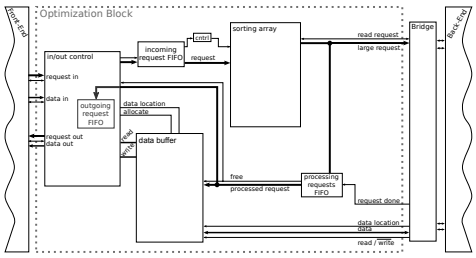


Fig. 2: The Optimization Block.

The optimization block has two major components: the sorting array, that implements the reordering policy and the data buffer that allocated data for write requests. In the following paragraphs we present the flow of a request from the Front-End through the Optimization Block towards the Back-End. Then we present the functioning of the optimization block, depicted in figure 2. When the bus interface receives a new request, before acknowledging it, two checks are done. First, the data buffer must have enough space to allocate the data. The size can be anywhere from 1Byte up to 16kBytes and varies with memory requests. The second requirement is that the sorting array has enough room to receive one extra request. If the above checks are met, the sorting array will save the request information: address, size and read or write while the data buffer will allocate the required size. In case of a write request, the data will be transferred from the bus to the data buffer immediately. The sorting array will reorder the memory requests and then send them to the Back-End that is responsible for the appropriate control signals towards the DRAM. In case of a write, the data is transferred to the DRAM and the space in the data buffer can be de-allocated. In case of a read, data from the DRAM will be stored in the data buffer. Once this is done, the request leaves the sorting array and the Front-End is responsible for sending the data back through the bus. De-allocation is performed only after all the data has been sent through the bus.

The sorting array is the central element in the optimization process. Sedition IV describes in detail the different sorting policies we have considered for implementation and motivates our choices. It is important to note that differences between the sorting policies are localized in the sorting array, the rest of the design remains unchanged.

The data buffer has the role of decoupling the transfers through the bus with the transfers through the memory interface. This is because either the bus can have interruptions (potentially higher priority requests) or the DRAM can require a row change or a refresh. Because these interruptions are unpredictable, we have chosen to decouple the two interfaces. Therefore, when reading data from the DRAM it has to be stored in the data buffer before sending it through the bus and consequently, before writing to the DRAM, the data from the bus is stored in the data buffer. Requests are processed

out of order and therefore the data buffer will be de-allocated in a random order leading to fragmentation and therefore a structure more complex than a FIFO is required. The data buffer allocates space with a size given by the request entering the Front-End. If there is enough room the data buffer will acknowledge the memory request that will be passed further to the sorting array. The allocation policy details are discussed in Section IV.

The design can easily adapt to new reordering policies because the sorting array has an interface that is comparable with a FIFO. Also, a new allocation policy does not require adaptations of any other part of the design. This approach keeps the changes localized and makes the design highly flexible. The modular design also brings some limitations like the fact that the Optimization Block has no direct control over the DRAM. Interacting with the DRAM directly would give more possibilities in optimizing the transfers. Stalls could be potentially avoided if the Optimization Block could send the precharge and activate commands in advance before issuing the request to the Back-End. Another disadvantage is that separate the Front-End, Optimization Block and Back-End requires additional glue logic and FIFO's.

IV. Main functionalities

In this section we discuss the details of a memory controller that reorders memory accesses. We start by presenting the reordering policies that we have considered. Since memory access can now complete out of order, two complications are introduced. The data buffer will be de-allocated out of order and a complex allocation policy is required in order to avoid data buffer fragmentation. Also, by reordering accesses we can have data hazards of type read after write, write after read or write after write. In such cases conflicts must be detected and reordering must be prevented.

A. Reordering Policy

The goal of the memory access reordering is to have as few as possible cycles when the data bus is not transferring (stalls). There are two major sources of stalls that can be avoided: First is when a row change is required. For example if we have 3 memory accesses on r_1, then r_2 and then again on row r_1. Without reordering, the DRAM needs 2 row changes while if we reorder the requests to be r_1, r_1 and r_2 then the DRAM needs a single row change. The second source of dead cycles is accesses to the same row with reads and writes alternating and incurring the CAS latency multiple times (the delay in clock cycles between the registration of a read command and the availability of the first piece of output data). For example the sequence rd_1, wr_1, rd_2 can be re-ordered to rd_1, rd_2, wr_1 to save CAS clock cycles. Therefore, the goal of the reordering policy is to minimize the number of row changes and the number of changes from writes to reads while maintaining memory consistency.

All policies that we have analyzed use an implementation of the insertion sort algorithm that executes in $O(1)$ [3] and are based on the memory address. The row and bank addresses are used for sorting and the column address is used for memory consistency checks. A new request entering the sorting array is compared in parallel with all the requests present in the array. All the requests that have larger row addresses will be shifted by one position towards the end of the array and the entering request will be inserted in the newly created empty array slot. Detection of consistency issues or policy rules can alter the insertion position. All address comparisons are performed in a single cycle and a new request is inserted in the sorting array.

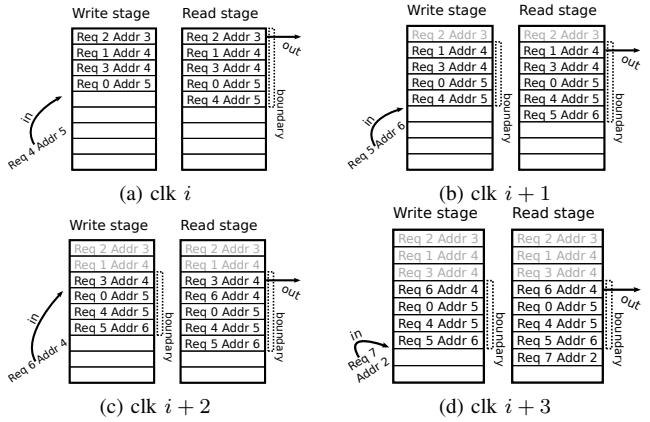
We have chosen to add a separate sorting array for each DRAM bank. While processing requests, when a row change is needed in the current bank, the arbiter switches to a different bank that has several requests queued up for a particular row. This approach has the added benefit that while processing requests from one bank, requests in the other banks are accumulated and sorted. Special care must be taken to prevent starvation, ie. having requests waiting disproportionately long in one queue while another queue is constantly filled and emptied. We handle this issue by forcing a bank switch after a predefined number of requests.

We assume that we are switching to a new bank and that the sorting array has received a number of requests sorted by their row address. The sorting policy answers two questions, first, "How do we decide which of the requests in the sorting array we should send to the DRAM?" We have considered two answers: of the requests in the sorting array either pick the row that has the most requests, irrespective of the address (we call this Largest Block First) or pick the first entry in the sorting array (we call this Smallest Address First since entries are sorted by their addresses).

The second question is "New requests entering the array are placed at the end of the sorting array or are they sorted interleaved with the existing ones?". We introduce the concept of a $boundary$. When switching banks, the entries already in the sorting array of the newly considered bank are freezed (sorting wise), and a $boundary$ is placed after the last request. With a static boundary, a new request entering the sorting array is placed after the boundary position and sorting occurs only between the requests below the boundary. When the last request within the boundary has been sent to the DRAM, a new boundary is set at the last entry of the array and the process is resumed. With a dynamic boundary, any new requests for rows smaller than the boundary request row will be inserted before the boundary (see figure 3).

The reason for a boundary is to prevent requests at low address from entering the array at the top position because that would create extra row changes. Suppose that in the current bank there is an ongoing burst on row r_2, there is another request for row r_2 and that there are two new request entering the sorting array for row r_2 and for row r_1. Sorting would place r_1 ahead of row r_2 and this would cause two row changes while only a single row change would be sufficient. With a static boundary both new requests are placed at the bottom of the queue, request for r_1 first, then the request for r_2. With a dynamic boundary, the new request for row r_2 is placed right after the other request for row r_2, the boundary is

moved one position further and the request on row r_1 is placed below the boundary. Figure 3 shows a motivational example of how the Shortest Address First with dynamic boundary sorts a sequence of memory requests. In the figure, the request number is its index while the address is the row number.



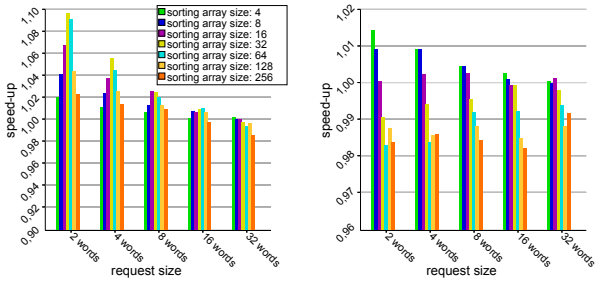
(a) clk i (b) clk $i+1$ (c) clk $i+2$ (d) clk $i+3$

Fig. 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.

In order to compare the four reordering policies, we have developed our own in house software simulator that computes the number of total clock cycles, the number of row changes and all other relevant metrics e.g., implementation complexity. The simulator takes as input the same memory access traces that are used with the Modelsim VHDL simulations in Section V. From figure 4a we can see that the dynamic boundary always performs better than the static boundary and that the best benefits are for small request sizes and sorting array sizes varying from 8 to 128 entries. We estimate that for both the static and dynamic boundary the hardware complexity is very similar. Figure 4b shows the comparison between the Largest Block First policy and the Smallest Address First policy. For sorting array sizes of 16 elements and larger, the Smallest Address First policy wins by about 1.5% while for small sorting array sizes and small request sizes the Largest Block First is faster by up to 1%. The Largest Block first requires slightly more logic to implement because of extra counters and comparators. Based on the performance and hardware cost estimates we have implemented the Shortest Address First approach with dynamic boundary.

B. Data Buffer Allocation

This section presents the policy used for the data buffer allocation. Because the memory requests are executed out of order and deallocation is done in a different order than allocation, a complex allocation policy is required. Furthermore,



(a) (b)

Fig. 4: Policy comparison: a) dynamic boundary versus static boundary speedup for the Smallest Address First policy; b) Largest Block First policy versus Smallest Address First policy speedup, both with static boundary.

requests can have arbitrary sizes. When a new memory request enters the memory controller we need a way to determine if there is a block in the data buffer that is large enough. An additional requirement is that the availability of space needs to be determined within a single cycle. Software based allocators like *malloc* use linked lists to keep track of all free blocks in the memory [2] and of course, this solution is not suitable for hardware implementation.

We have implemented the approach used in [5] where an or-tree is used to find out if there is a block that is large enough. Each level l of the or-tree can signal if there is space available of size 2^l. The start address of the free block can be found using a 'non-backtracking binary search algorithm' implemented with an and-tree and few multiplexers [5]. After detecting if there is enough space, every location in the allocated data block has to be marked.

C. Memory Consistency

Memory inconsistencies are created when overlapping requests that will write or read entirely or partially at the same location in the memory are executed in the wrong order through reordering. Several types of request overlaps are possible: figure 5a shows the case of two requests having partial overlap while figure 5b shows the case of a full overlap when every location of a request has to read or write is also read or written by another request. The request overlaps can be easily detected using their addresses and sizes. There is

(a) partial overlap (b) full overlap

Fig. 5: Request overlap types.

no memory inconsistency when only read requests are being sorted. However, write request can introduce inconsistencies. When a new write request enters the sorting array, all elements already present check in parallel if there is an overlap with the

new request. If an overlap is detected, the new request must be inserted below any overlapping request. It is important to note that overlaps can occur only between requests on the same row and that the column address will be used to detect overlaps. Therefore, the overlap detection logic can easily exclude consistency comparisons between requests accessing different rows. Another important note is that requests coming over the bus that span over multiple DRAM rows (for instance end of one row and beginning of the next) are split into multiple requests that target the individual rows.

When requests are overlapping, extra optimizations are possible. A part of a request can be deleted when two write requests are overlapping. For instance, in figure 5a if both requests are write requests then the gray part of the first request will be overwritten by the second request. The memory controller detects this overlap and forces the two request to execute in order. Extra bandwidth can be gained by reducing the size of the first request and also the reordering restriction can be eliminated. We have chosen to not implement this kind of optimizations because it would require communication between the sorting array (where overlap of request is detected) and the data buffer (where the data allocation needs to be altered) and this would negatively impact the intended design modularity.

V. Experimental Results

This section discusses our experimental results that are structured in three subsections: speedup, power and energy savings and discussion of the sorting array size and the data buffer size trade-off.

A. Application Speedup

For our simulations we have generated the memory access traces for the studied applications: 3D FFT, Conjugate Gradient and random read/writes. For 3D FFT and random read/writes we have varied the size of the request ranging from 2 words up to 32 words. For CG (which is mainly sparse matrix vector multiplication [4]), we have varied the matrix size from 500×500 elements to 8k×8k elements. We have then used the traces as inputs for the Modelsim simulations and obtained the number of cycles and all other relevant metrics.

The speed-up results for the 3D-FFT are given in figure 6. The comparison is against the memory controller without memory request reordering. These results show a large improvement for the smaller request sizes while the speed-up slowly degrades as the request size increases. This is expected since for small requests the actual time spent transferring data is comparable to the overhead (the amount of time spent activating/precharging rows and other timing constraints). Therefore, optimization through reduction of overhead can bring significant improvements. For larger requests the dead cycles represent a smaller percentage of the total execution time and thus, optimization is restricted. Another important conclusion is that sorting array sizes in excess of 128 elements bring little improvement and do not validate the additional hardware cost.

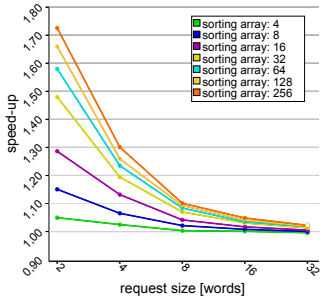
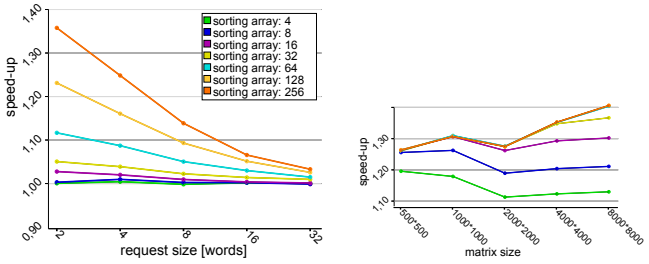


Fig. 6: The speed-up obtained for 3D-FFT.

The results for the random read and write benchmark are shown in figure 7a and show similar behavior. It is interesting to compare these results with the 3D-FFT in terms of the speedup per sorting array size. With any request size, for the 3D-FFT the sorting array with 8 entries brings higher benefits than sorting array of 64 for random reads and writes.



(a) random read writes (b) CG

Fig. 7: The speedup for random read and writes and CG.

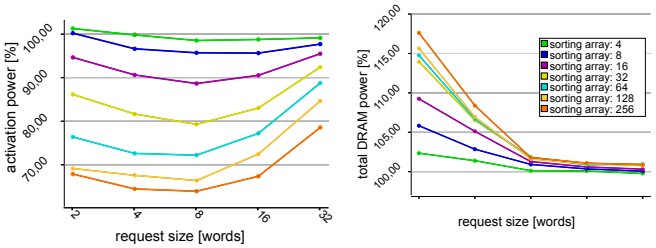
For the Conjugate Gradient benchmark (see figure 7b) we vary the size of the matrix, while the request size depends on the number of non-zero elements on each row. The speed-ups increase up to a sorting array size of 32 entries for several matrix sizes. Larger sorting arrays do not bring any additional improvements, as shown on the figure.

Sorting Array size	64		256	
Speedup	Maximum	Average	Maximum	Average
3D-FFT	1.58	1.19	1.73	1.24
Random R/W	1.12	1.06	1.36	1.17
CG	1.40	1.32	1.41	1.32

TABLE I: Speed-ups due to reordering.

B. Power and Energy

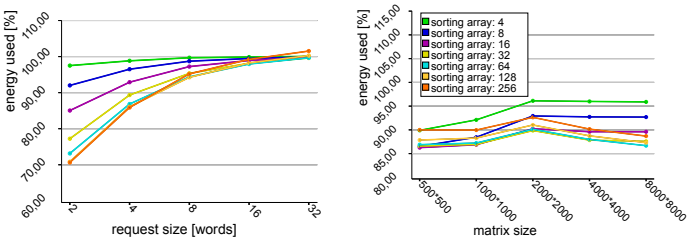
In order to estimate the power and energy usage we have first synthesized the design into a net list with Synopsys Design Compiler. ModelSim is then used to generate an activity file with the net list and the test bench that is also used for the speed-up simulations. The activity file generated by ModelSim is then analyzed by Synopsys PrimeTime and the needed values are extracted. All power results of the controller are based on a *90nmSP* process. The analysis of the power and energy used by the DRAM were performed by using the Micron System Power Calculator [9].



(a) activation power (b) total power

Fig. 8: Activation and total DRAM power for 3D-FFT.

Figure 8a shows that memory request reordering drastically reduces the DRAM activation power (the graph shows results for the 3D-FFT traces). For a sorting array with 128 entries, for request sizes ranging from 2 two 16 words, the DRAM uses about 70% of the activation power consumed without reordering. On the other hand, since the requests are executed faster, the total power increases, especially for small requests sizes (that benefit the most from reordering), as depicted on figure 8b. Figure 9 shows the total energy usage compared to an implementation without a reordering policy. Adding the reordering logic increases the energy used by the memory controller while it decreases the total energy used by the DRAM because of fewer commands, fewer row activations, precharges and so on. The optimal configuration balances the speedup obtained by the reordering logic with the resources consumed. From figure 9a we can observe that energy savings are obtained for every configuration up to a sorting array size of 128 entries. After this point, doubling the sorting array brings little execution time improvements (see figure 6) while consuming significantly more silicon area and energy. We can see that configurations of 256 entries and larger are rather inefficient both in terms of performance improvements and energy savings. A summary of the energy savings for a sorting array size of 64 entries is given in table II. We can thus conclude that request reordering brings both speed-ups and energy savings.



(a) 3D-FFT (b) CG

Fig. 9: The energy used by the controller and the DRAM with reordering as a percentage of the energy used without reordering capabilities.

VI. Conclusions

In this paper we have studied the effect of DRAM access reordering on performance and energy usage. We show that with careful request reordering, we can both reduce execution time and obtain energy savings. We support memory requests of arbitrary sizes ranging from one byte up to 16kB. We have presented a flexible architecture that can be easily adapted to any bus protocol and arbitrary DRAM types. The sorting policy implementation is localized in a single block, adaptation to new sorting policies not influencing any other design blocks. Using our in house simulator, we have compared several reordering policies in terms of performance and hardware complexity and we have opted to implement a policy named Smallest Address First with dynamic boundary. Reordering of memory accesses brings two side-effects. The first is memory consistency: we must ensure that accesses to the same memory location are executed in program order. The second side-effect of memory request reordering is that data allocated in the data buffers will be deallocated out of order and therefore an allocation policy is required that can handle data buffer fragmentation. We have obtained application speedups of 1.58x for 3D FFT and 1.4x for Conjugate Gradient workloads. In addition, our reordering proposal reduces memory activation power of the memory by up to 40% and total energy per application by 26.6% for 3D FFT and 13.2% for CG.

	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 II: Energy savings for sorting array size 64.

References

[1] ARM, A. Axi protocol specification, mar. 2004. v1.0.

[2] AUGUST, P. D. Inner workings of malloc and free. http://www.cs.princeton.edu/courses/archive/fall06/cos217/lectures/14Memory-2x2.pdf, 2006.

[3] BENDER, M. A., FARACH-COLTON, M., AND MOSTEIRO, M. Insertion sort is O(n log n), 2004.

[4] C. CIOBANU, ET AL.. Scalability evaluation of a polymorphic register file: A CG case study. *Architecture of Computing Systems-ARCS 2011*.

[5] CHANG, J., AND GEHRINGER, E. A high performance memory allocator for object-oriented systems. *Computers, IEEE Transactions on 45*, 3 (mar. 1996), 357 –366.

[6] J. CARTER, ET AL. Impulse: building a smarter memory controller. In *HPCA* (jan. 1999), pp. 70 –79.

[7] LIN, E. Quality-aware memory controller for multimedia platform soc. In *SIPS* (aug. 2003), pp. 328 – 333.

[8] MICRON. Synchronous DRAM 512Mb, 2000. Rev. L 10/07 EN.

[9] MICRON. SDRAM System-Power Calculator. http://www.micron.com/get-document/?documentId=31, apr. 2001.

[10] PICHEL, J., SINGH, D., AND CARRETERO, J. Reordering algorithms for increasing locality on multicore processors. In *HPCC* (sep. 2008), pp. 123 –130.

[11] S. LANGEMEYER, P. PIRSCH, H. BLUME. Using SDRAMs for two-dimensional accesses of long $2^n \times 2^m$-point FFTs and transposing. In *SAMOS* (2011), pp. 242–248.

[12] S. RIXNER, E. Memory access scheduling. In *ISCA* (2000).

[13] SHEKHALEV, D. High Speed SDRAM Controller With Adaptive Bank Management and Command Pipeline. http://opencores.org/project,hssdrc, dec. 2009.

[14] T. WATANABE, ET AL. Access optimizer to overcome the 'future walls of embedded DRAMs' in the era of systems on silicon. In *Digest of Technical Papers. ISSCC.* (1999), pp. 370 –371.

[15] Y. C. HU, A. COX, W. Z. Improving fine-grained irregular shared-memory benchmarks by data reordering. In *Supercomputing, ACM/IEEE 2000 Conference* (2000), p. 33.