Efficient Generation of Application Specific Memory Controllers

Marco V. Natale
Optimization Research Group,
TU Kaiserslautern
Kaiserslautern, Germany
natale@mathematik.uni-kl.de

Matthias Jung
Fraunhofer Institute for Experimental
Software Engineering (IESE)
Kaiserslautern, Germany
matthias.jung@iese.fraunhofer.de

Kira Kraft
Microelectronic Systems Design
Research Group, TU Kaiserslautern
Kaiserslautern, Germany
kraft@eit.uni-kl.de

Frederik Lauer
Microelectronic Systems Design
Research Group, TU Kaiserslautern
Kaiserslautern, Germany
flauer@rhrk.uni-kl.de

Johannes Feldmann
Microelectronic Systems Design
Research Group, TU Kaiserslautern
Kaiserslautern, Germany
jfeldman@rhrk.uni-kl.de

Chirag Sudarshan
Microelectronic Systems Design
Research Group, TU Kaiserslautern
Kaiserslautern, Germany
sudarshan@eit.uni-kl.de

Christian Weis
Microelectronic Systems Design
Research Group, TU Kaiserslautern
Kaiserslautern, Germany
weis@eit.uni-kl.de

Sven O. Krumke
Optimization Research Group,
TU Kaiserslautern
Kaiserslautern, Germany
krumke@mathematik.uni-kl.de

Norbert Wehn
Microelectronic Systems Design
Research Group, TU Kaiserslautern
Kaiserslautern, Germany
wehn@eit.uni-kl.de

ABSTRACT

The increasing gap between the bandwidth requirements of modern *Systems on Chip* (SoC) and the I/O data rate delivered by *Dynamic Random Access Memory* (DRAM), known as the *Memory Wall*, limits the performance of today's data-intensive applications. General purpose memory controllers use online scheduling techniques in order to increase the memory bandwidth. Due to a limited buffer depth they only have a local view on the executed application. However, numerous applications, especially in the embedded systems domain, have regular or fixed memory access patterns, which are not yet exploited to overcome the memory wall. In this paper, we present a new methodology to generate the configuration for an *Application-Specific Memory Controller* (ASMC), which has a global view on the application and utilizes application knowledge to decrease the energy and increase the bandwidth. Therefore, we analyze the DRAM access pattern of the application offline by solving an instance of the *Min-k-Union* problem and generate a configuration for a reconfigurable address mapper. For several applications we show an improvement in energy efficiency of up to 8.5× and sustainable bandwidth of 8.9×.

CCS CONCEPTS

• **Hardware → Dynamic memory**; **Software tools for EDA**;

KEYWORDS

Application Specific Memory Controller, DRAM, Address Mapping, Optimization, Min-k-Union, Combinatorics, Embedded Systems

ACM Reference Format:
Marco V. Natale, Matthias Jung, Kira Kraft, Frederik Lauer, Johannes Feldmann, Chirag Sudarshan, Christian Weis, Sven O. Krumke, and Norbert Wehn. 2020. Efficient Generation of Application Specific Memory Controllers. In *The International Symposium on Memory Systems (MEMSYS 2020), September 28-October 1, 2020, Washington, DC, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3422575.3422796

MEMSYS 2020, September 28-October 1, 2020, Washington, DC, USA

ACM ISBN 978-1-4503-8899-3/20/09…$15.00
https://doi.org/10.1145/3422575.3422796

1 INTRODUCTION

The achievable bandwidth and energy efficiency of *Dynamic Random Access Memories* (DRAMs) strongly depends on the access patterns to the memory devices. Many applications have a regular or fixed memory access pattern, either temporal (the commands follow a well-defined rule) or spatial (the sequence of the accessed addresses is deterministic). Especially in embedded systems, which are designed for specific purposes, we find applications with deterministic memory access patterns. Among them: streaming, real-time image or signal processing, and the inference of many neuronal networks.

On the computing architecture side this inherent application-specific knowledge has been heavily utilized by techniques like *Hardware Accelerators* (HWA) and *Application-Specific Instruction-Set Processors* (ASIPs). However, on the memory side there is a limited amount of research that exploits the application knowledge for customized application-specific memory controllers (cf. Section 3). Thus, we propose the concept of designing an *Application-Specific Memory Controller* (ASMC), which largely increases the overall efficiency of the DRAM subsystem. This ASMC can be implemented in systems based on *Field Programmable Gate Arrays* (FPGA) and in *Application-Specific Integrated Circuits* (ASICs), as shown in Section 6. In contrast to commercial off-the-shelf DRAM controllers (e.g. [22, 24]), which are targeting general purpose systems, an

ASMC is lean, highly energy and area efficient while it can improve the achievable bandwidth for the specific target application.

General purpose memory controllers use online scheduling algorithms and quality of service methods [2, 28] in order to increase the memory bandwidth. Due to a limited buffer depth they only have a *local* view on the executed application. For applications with deterministic memory access pattern an optimized DRAM address mapping[1] (often called *scrambler*), can supersede the online scheduler because it was designed beforehand with a *global* application view. Such an address scrambler is a lean memory controller frontend, because it can easily be realized by means of a simple lookup table or a network of multiplexers in order to maximize the number of row buffer hits and exploit the bank level parallelism of the DRAM device (see Section 2).

This work is based on our previously presented *ConGen* approach at MEMSYS'16 [12], where we showed for the first time how an ASMC can be designed by an application-specific address scrambler. The challenge for the scrambler design is to find a bijective boolean function that scrambles the address bits according to application knowledge in order to minimize the number of row misses, as shown in Figure 3a. This function can be estimated by solving an NP-hard mathematical optimization problem. Therefore, the previously presented solution was based on an approximate heuristic, where the size of the generated hardware architecture was unfortunately unbound.

In this paper, we present an hybrid algorithm called *ConGen2* which consists of two parts. The first part minimizes the penalties of read write switches by using an online write buffer. The second part presents an exact offline-algorithm for the address mapping problem that minimizes the number of row misses under given hardware constraints. Figure 1 shows a configurable address scrambler, which can also be found in commercial DRAM controllers such as [22] or [24]. The goal of our optimization is to find a configuration for this scrambler, such that the number of row misses is minimal for a specific application. We show that this optimization problem is equivalent to the NP-hard *Min-k-Union* problem. Furthermore, we present, to the best of our knowledge, for the first time an exact algorithm to efficiently solve this problem. The final bandwidth and energy is estimated with the design space exploration framework DRAMSys [14], as shown in Figure 4. For several applications we can show an improvement in energy efficiency of up to 8.5× and bandwidth of 8.9× by using an ASMC, compared to a state-of-the-art controller that uses only online scheduling. With this approach we bring relief to the system designer, who can fully concentrate on the application itself and does not have to deal with the details of the DRAM subsystem. Moreover, ConGen2 can be integrated into the flow of *High Level Synthesis* (HLS) tools to facilitate fast implementations of systems with DRAM.

The paper is structured as follows: Section 2 explains the background on DRAM. The related work is discussed in Section 3. Section 4 shows the ConGen2 approach. It is proven that the problem is equivalent to the *Min-k-Union* problem (cf. Appendix A) and an exact algorithm based on a branch and bound approach is presented in order to solve this problem efficiently. Section 5 discusses the implementation of the algorithm and presents results for several benchmarks. A hardware implementation of the proposed address scrambler and memory controller, including the PHY is shown in Section 6. Finally the paper is concluded in Section 7.

[1] The address mapping, performed in the memory controller, defines the physical location of the data elements in the DRAM (cf. Section 3.2).

2 DRAM BACKGROUND

In the following we explain the basic DRAM architecture and functionality, to understand the limiting factors with respect to energy and bandwidth. As depicted in Figure 2, a DRAM device is organized as a set of memory banks (e.g. eight) that include memory arrays (e.g. two). Each memory array has row and column decoders, master wordline drivers and *Secondary Sense Amplifiers* (SSA). Busses, buffers, control signals, voltage regulators, charge pumps and other peripherals are shared between the different banks. The memory arrays are formed in a hierarchical structure out of *sub-arrays* (SA) for efficient wiring, increased speed and reduced power consumption. Therefore, each SA is equipped with *Primary Sense Amplifiers* (PSA). A typical memory SA consists of e.g. 512 cells · 512 cells = 256 Kb[2]. For instance, a 64 Mb DRAM bank is formed out of two memory arrays, where each memory array consists of $8 \cdot 16 = 128$ SAs. A single memory cell is built as a transistor capacitor pair where the data is stored in the capacitor as a charge. The individual cells in each sub-array are connected to *Local Wordlines* (LWL) and *Local Bitlines* (LBL). The LBLs and LWLs are connected to global *Master Bitlines* (MBL) and *Master Wordlines* (MWL), respectively, which span over the complete memory array.

In the following, the basic DRAM operations are explained on a single read transaction, noting that writing works similar. To read data from the memory, a precharge command is issued by the memory controler (PRE) to prepare the LBLs to a halfway voltage level. The time interval for this operation is called t_{RP} (*Row Precharge*). After the PRE an activate command (ACT) is issued to drive the LWL high and transfer the charge between the memory cells and the connected LBLs. The voltage difference caused by this transfer of charge (data) is sensed by the PSAs. The complete targeted row in that bank, called *DRAM Page* is now latched in the PSAs. The time interval between the ACT command and data ready at the PSAs is called t_{RCD} (*Row to Column Delay*. Then, a read (RD) command can be sent in order to move data from the PSAs to the SSAs (t_{CL}) and finally read specific columns of data from the SSAs ($t_{BURST} = t_{CCD}$), which are interacting with the I/Os. Once finished, the wordlines can be switched off, the cell capacitors disconnected, and the LBLs can be precharged (PRE) again.

The combination of primary and secondary sense amplifiers of the memory arrays in one bank are often conceived as a, so called, *row buffer*. This row buffer is a model, which abstracts the real physical DRAM architecture, i.e. PSAs and SSAs. The buffer has usually a size ranging from 1KB to 8KB (called DRAM page size, see Figure 2). It acts like a small cache that stores the most recently accessed row of the bank. The latency of a memory access to a bank largely varies depending on the state of this row buffer. If a memory access targets the same row as the currently cached row in the buffer (called *row hit*), it results in a low latency and low energy memory access. Whereas, if a memory access targets a different row as the current row in the buffer (called *row miss*), it results in

[2] We use JEDEC's notation for storage capacity: $K = 2^{10}$, $M = 2^{20}$, $G = 2^{30}$

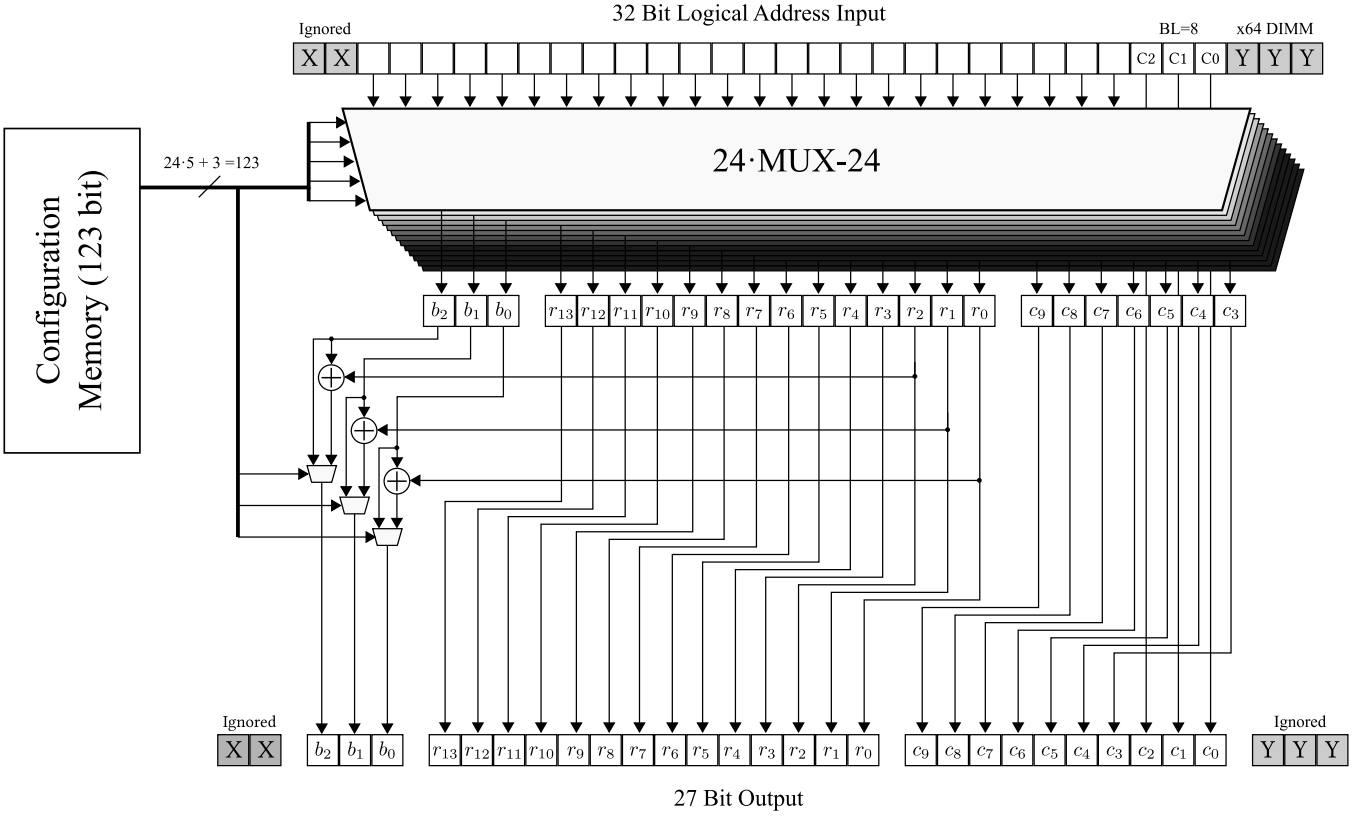


Figure 1: A Configurable Address Scrambler

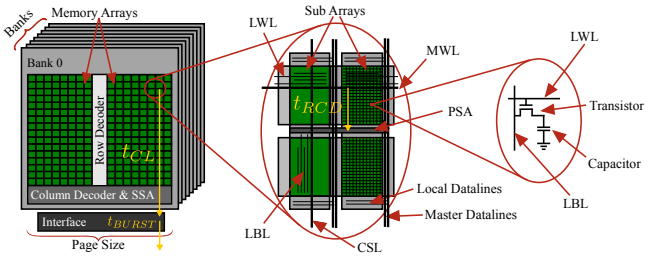


Figure 2: DRAM Architecture

higher latency and energy consumption: A precharge command (PRE) must be issued before the required row can be loaded (ACT) from the DRAM array into the row buffer. Activated rows in two different banks can be accessed in parallel without a penalty. This, so called *Bank Parallelism*, can be exploited for better performance.

In the following we explain the important DRAM performance and energy penalties, where Table 1 shows the key timing and current parameters for a DDR3 DRAM device:

- **Row Misses:** A row miss causes a large penalty on the DRAM's databus, e.g. in the worst case $t_{WR} + t_{RP} + t_{RCD} + t_{CL} = 42$ cycles and additional energy, e.g. $E_{Miss} = V_{DD} \cdot (I_{DD0} \cdot t_{RC} - I_{DD3N} \cdot t_{RAS} - I_{DD2N} \cdot t_{RP}) = 1.78$ nJ for a DDR3-1600 Device [13].
- **RD/WR-Switching:** Switching from a RD to a WR on the same row needs a bus turnaround time (e.g. 2 cycles on DDR3-1600). However, switching from a WR to a RD needs a much longer turnaround time which results in a larger penalty on the databus (e.g. $t_{WTR} + t_{CL} = 16$ cycles).
- **Refresh:** DRAM has to be refreshed regularly due to its charge-based bit storage property (capacitor) which suffers from leakage currents. The refresh performance penalty for a 1 Gb DDR3-1600 device is 1.5%. However, it is a fact that 40% to 50% of the bandwidth in future 32 or 64 Gb DRAMs has to be attributed to refresh commands [20].

Techniques for reducing refresh and RD/WR-switching penalties are presented in [9, 15]. Since row misses and RD/WR switches have the highest performance and energy penalty we focus on their minimization in this paper.

3 RELATED WORK

Several DRAM controller approaches exist to mitigate the aforementioned penalties, which are covered in this section.

3.1 Online Scheduling

General purpose DRAM controllers use online scheduling techniques to reduce the number of row misses. Usually, they have two major modes, called *Open Page Policy* (OPP) and *Closed Page Policy* (CPP). The OPP keeps the current row active after a RD

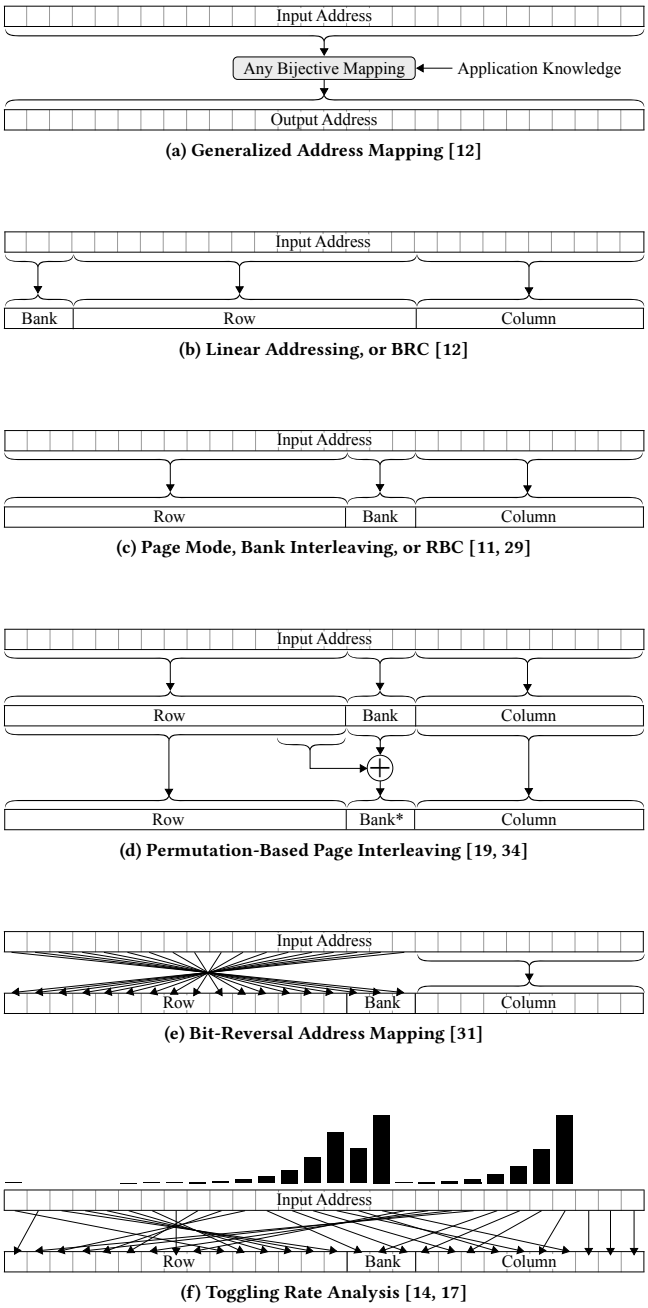
Table 1: Key Parameters for a DDR3-1600 Device [11, 23]

Name	Explanation	Value
t_{BURST}	*Data Burst Duration*: The time period that a data burst occupies the data bus.	4 clk
t_{CCD}	*Column to Column Delay*: The minimum column command timing, determined by internal burst (prefetch) length.	4 clk
t_{RP}	*Row Precharge*: The time interval that it takes for a DRM array to be precharged (PRE) and prepared for antoher row access.	10 clk
t_{RCD}	*Row to Column Delay*: The time interval between row access and data ready at PSAs, in other words: The time interval between ACT and RD on the same bank.	10 clk
t_{RAS}	*Row Access Strobe*: The minimum active time for a row, in other words: The time interval between row access command and data restoration in a DRAM array.	28 clk
t_{RC}	*Row Cycle*: The fastest time to ACT and PRE the same row ($t_{RC} = t_{RAS} + t_{RP}$), in other words: The time interval between accesses to different rows in a bank.	38 clk
t_{CL}	*CAS Latency*: The time needed to transfer the data from the PSAs to the SSA and the interface, in other words: The delay from a RD/WR command until data can be read/written at the interface.	10 clk
t_{WR}	*Write Recovery*: The minimum time interval between the end of a WR burst and a PRE command.	12 clk
t_{WTR}	*Write to Read*: The minimium time interval betwen the end of a WR burst and a RD command.	6 clk
I_{DD0}	*One Bank Active Precharge Current*: Measured across ACT and PRE commands to one bank (other banks remain precharged).	70 mA
I_{DD2N}	*Precharge Standby Current*: Measured when all banks are precharged (PRE).	45 mA
I_{DD3N}	*Active Standby Current*: Measured when one bank is active (ACT).	45 mA
V_{DD}	*Supply Voltage*	1.5 V

or WR, whereas the CPP precharges the row automatically (Auto-Precharge: RDA/WRA). The CPP is often used for server applications (e.g. webserver), where the accessed DRAM addresses are uniformly random. The OPP is used in desktop PCs and mobile devices where row hits are more likely to a higher data locality. For further improvement of row buffer hits in the OPP, online scheduling techniques are used. The most common online DRAM scheduler, called *First Ready First Come First Served* (FR-FCFS), has been presented by Rixner et al. [28]. The FR-FCFS scheduler places incoming requests into a queue in such a way that they are placed next to requests that target the same row. By using this strategy, groups of row hits are formed (*row-hit-first* policy). If there are no row hits in the queue of the scheduler, the oldest request in the scheduler will be issued (*oldest-first* policy). This approach has been improved for general purpose processors by [9] and for multicore and heterogeneous systems in [27] and [2]. In order to reduce the number of RD/WR-switches DRAM controllers buffer RD and WR commands in two distinct queues. An arbiter switches between RD and WR mode to diminish the penalty. The before presented techniques are used in commercial off-the-shelf DRAM controllers such as [24] and [22].

3.2 Address Mapping

Another approach to reduce the number of row misses is an optimized physical DRAM address mapping, that can assist or even outperform an online scheduler in specific cases. The straight forward address mapping, called *Linear Addressing*, shown in Figure 3b, is rarely used in practice. The most common addressing scheme, called *Page Mode* or *Bank Interleaving* [11, 29], brings the upper bank bits of the address down between the row and the column bits (shown in Figure 3c) to ensure a better bank utilization. State-of-the-art memory controllers like [22, 24, 25] support both, the linear



(a) Generalized Address Mapping [12]

(b) Linear Addressing, or BRC [12]

(c) Page Mode, Bank Interleaving, or RBC [11, 29]

(d) Permutation-Based Page Interleaving [19, 34]

(e) Bit-Reversal Address Mapping [31]

(f) Toggling Rate Analysis [14, 17]

Figure 3: Different Address Mappings and Scramblers

Bank-Row-Column (BRC) and interleaved *Row-Bank-Column* (RBC) address mapping scheme. Sophisticated memory controllers used in high end CPUs support the *Permutation-Based Page Interleaving* technique [19, 34]. Figure 3d explains this advanced mapping scheme: the accesses to different rows in the same bank are transformed into accesses to different banks by XORing the bank bits with selected row bits. The authors of [31] present a *Bit-Reversal* address mapping, shown in Figure 3e, in order to improve the DRAM

bandwidth and access latency. However, as we will show in Observation 2 in Section 4.2.1 the reversing of the bits does not change the number of row misses, but results only in a renumbering of the banks, rows and columns. Virtual platform tools like *Synopsys DesignWare DDR Explorer* [17] or *DRAMSys* [14] provide capabilities to analyze the toggling rates of each address bit for a specific application, as shown in Figure 3f. This information can help to create custom address scramblers used as a frontend for e.g. the MIG (Memory Interface Generator) memory controller on FPGAs or it can assist to configure the address mapping of memory controllers that target ASIC implementations, such as [24] and [22]. The authors of DReAM [8] analyze the toggling rates during runtime and they change the address mapping on the fly. However, this approach has a large overhead due to data movement when the address mapping is changed. In [3] the authors proposed a scrambling mechanism based on substitution-permutation networks to reduce the vault/bank conflicts in HMC and to enable robust operation even in the presence of pathological traffic patterns. The authors of [1] presented *HAMLeT*, an address remapping memory controller architecture for 3D-Stacked DRAMs. However, formal analysis of the address remapping was beyond the scope of their work. Recently, Hur et al. [10] presented an heuristic approach called *Bank Flip Address Map*. With their approach they focus mainly on simple image processing tasks e.g. rotation. Since the approach is an heuristic they do not target the optimal solution.

3.3 Application-Specific Approaches

The authors of [5] present *PARDIS*, a programmable memory controller based on a standard *Reduced Instruction-Set Computer* (RISC). However, implementing a sophisticated address mapping with this controller results in a large program and therefore in a large latency. Bayliss and Constantinides present an offline method for applications specific reuse and reordering [4]. In their approach they try to cache reoccurring addresses in an on-chip buffer to create a monotonic address pattern for the DRAM in order to maximize the number of row hits. The *Impulse* memory controller [6] is using physical address space, which is not backed by DRAM in order to constitute a *shadow address space* that allows application-specific optimizations by restructuring data. However, additional software, compiler and operating system support is needed for this controller. Techniques to overcome strided memory access pattern have been presented in [30]. Furthermore, the authors of [7] and [18] present an specific access pattern optimization for FFT based applications in order to optimize the bandwidth. In the previously presented *ConGen* approach [12] was based on an approximate heuristic. Therefore, a minimal number of row misses could only be approximated. The heuristic found a boolean function for the address mapping, that could be used by a synthesis tool. However, the boolean function could lead to a arbitrarily complex hardware architecture. Software and compiler techniques for access pattern optimizations are presented in [16, 26]. However, in software-less systems like ASICs or FPGA these approaches cannot be applied. In [15] it is shown that DRAM refresh can be switched off for applications with a data lifetime that is smaller than the currently required refresh period to gain additional bandwidth.

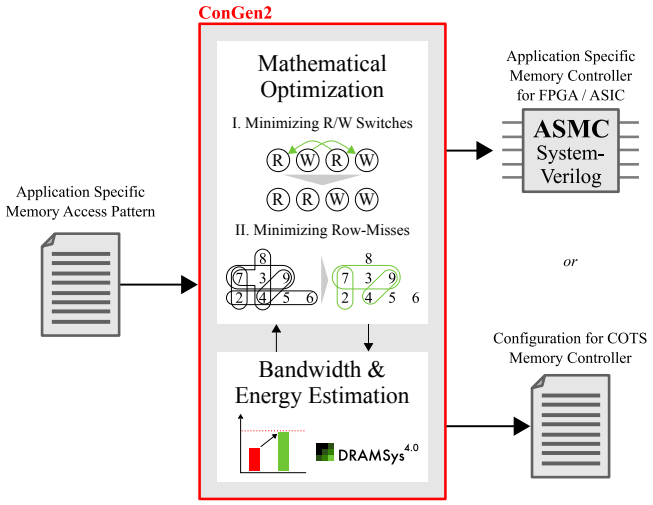


Figure 4: ConGen2 Methodology

4 CONGEN2 APPROACH

In this section, we explain our novel ConGen2 approach for applications that feature a deterministic DRAM access pattern, shown in Figure 4. The methodology receives the deterministic DRAM access trace as input and processes this trace in two consecutive steps:

I. **Minimization of RD/WR switches:** The behavior of an online write buffer, which minimizes the number of RD/WR switches by buffering the WR accesses, is anticipated and the input trace is transformed into a new trace that reflects the behavior of the online write grouper (c.f. Section 4.1). The new trace is used as input for the optimization algorithm in Step II.

II. **Minimization of row misses:** The presented algorithm calculates an application specific address mapping such that the number of row misses is minimal. This is the core of the presented ConGen2 approach and it is described in detail. Therefore, first, Section 4.2.1 defines a mathematical model of DRAMs as foundation of the further sections. Then, we show in Section 4.2.2 how the number of row misses can be minimized with an artificial DRAM that consists only of one single bank and prove that our problem is an instance of the *Min-k-Union* problem. In Section 4.2.3 we show how DRAMs with more than one bank (e.g. 8 and 16) can be transformed into an artificial DRAM with only one bank in order to solve the address mapping problem for real DRAM devices. The Sections 4.2.5 and 4.2.4 present a *Branch & Bound* algorithm in order to estimate a configuration for the address scrambler shown in Figure 1 such that the number of row misses is minimal.

The calculated minimization of RD/WR switches and row misses is then evaluated by the DRAM subsystem simulator DRAMSys to obtain the actual metrics like bandwidth, latency and energy, as shown in Section 5. In the following sections, we will detail our two steps.

4.1 Minimizing RD/WR Switches

In order to reduce the overheads of RD/WR-Switches, memory controllers usually implement a *Write Buffer* semantic. The memory controller is either in a read mode, where only reads (RD) are issued, or in a write mode, where only writes (WR) are issued. Since reads have the highest priority, they are passed through this buffer and write transactions are buffered until the buffer is full or a RD/WR hazard occurs. In both cases the controller will switch from RD- to WR-mode and the write buffer will be flushed. This technique ensures that the number of RD/WR switches is minimized. The usage of such an online write buffer would destroy the offline calculated optimizations of the Step II in our approach (Minimization of row misses, Section 4.2) since the memory transactions are reordered online. Therefore, we introduce the preprocessing Step I in our approach, that is performed before the actual row miss minimization in order to overcome this issue. The preprocessing follows the following idea: Because the write buffer has a deterministic behavior for a deterministic memory access pattern, its behavior can be fully anticipated, like buffer effects including RD/WR hazards. Thus, the deterministic memory access trace is transformed, such that it reflects the behaviour of an online write grouper. The algorithm for this preprocessing of the trace is presented in Algorithm 1.

4.2 Minimizing Row Misses

This section presents the core of the presented ConGen2 approach and it is described in detail including all the mathematical foundations. The main goal of the algorithm is to minimize the number of row misses for the given application specific memory access pattern. In the next subsection we describe a mathematical model which serves as the base for the development of the algorithm in the following subsections.

4.2.1 Mathematical Model. We start with describing the mathematical model for minimizing the number of row misses of a given application specific memory access pattern for some DRAM.

Let $\mathbf{a} = (a_0, a_1, \ldots, a_{m-1})$ be a finite application-specific sequence of memory accesses, where each a_i is a binary vector of length n containing the address of the i-th memory access in the given sequence. Formally, $a_i = (a_{i,0}, a_{i,1}, \ldots, a_{i,n-1})$, with $a_{i,j} \in \{0, 1\}$.

For a memory consisting of $B = 2^b$ banks, $R = 2^r$ rows per bank, and $C = 2^c$ columns per row, the address of the i-th memory access a_i can, according to a BRC mapping, be represented as follows (see Figure 3b): The first b bits $(a_{i,0}, \ldots, a_{i,b-1}) =: \mathbf{b}(a_i)$ are the *bank bits*, the following r bits $(a_{i,b}, \ldots, a_{i,b+r-1}) =: \mathbf{r}(a_i)$ are the *row bits*, and the last c bits $(a_{i,b+r}, \ldots, a_{n-1}) =: \mathbf{c}(a_i)$ are the *column bits*. An example is illustrated in Table 2 for a DRAM with $B = 2$ banks ($b = 1$), $R = 4$ rows per bank ($r = 2$), and $C = 4$ columns per row ($c = 2$).

The request to a memory bank induces a row miss if and only if no row in the corresponding bank has been activated so far (ACT), or if the last activated row in a bank differs from the currently requested. Formally, this leads to the following observation.

Observation 1. *The request of an element a_i, $i \in \{0, \ldots, m-1\}$, in* $\mathbf{a}$ *induces a row miss if and only if one of the following conditions hold:*

Algorithm 1 Offline Preprocessing

Input: A sequence T of n transactions, transferred to the write buffer, where a transaction consists of an address and access type.
Output: A sequence T' of n transactions, transferred from the write buffer to the DRAM.

```
Let B be a Buffer with a depth of k transactions
Let c be the number of valid entries in buffer B
Let S be a state register with the valid states Last Transaction Read (LTR) and Last Transaction Write (LTW)
S := LTR
k := 32
c := 0
for i = 0, ..., n − 1 do
    if T[i].Access = Read then
        if T[i].Address = B[j].Address for some j ∈ {0, ..., c − 1} then
            T'.append(B[j]), j = 0, ..., c − 1
            c = 0
        T'.append(T[i])
        S = LTR
    else
        if S = LTW then
            T'.append(T[i])
        else if c < k then
            B[c] = T[i]
            c = c + 1
        else
            T'.append(B[j]), j = 0, ..., k − 1
            c = 0
            T'.append(T[i])
            S = LTW
T'.append(B[j]), j = 0, ..., c − 1
return T'
```

Table 2: Sequence of Memory Addresses

	$\mathbf{b}(a)$	$\mathbf{r}(a)$		$\mathbf{c}(a)$		
$\mathbf{a}$	$a_{i,0}$	$a_{i,1}$	$a_{i,2}$	$a_{i,3}$	$a_{i,4}$	
a_0	0	1	0	1	0	miss
a_1	0	**0**	0	1	0	miss
a_2	1	1	1	1	1	miss
a_3	1	1	1	0	0	hit
a_4	1	**0**	1	0	0	miss
a_5	0	0	0	1	0	hit
a_6	1	0	1	1	0	hit
a_7	0	**1**	**1**	1	0	miss

(1) *For all $l \in \{0, \ldots, i-1\}$ it holds that $\mathbf{b}(a_l) \neq \mathbf{b}(a_i)$.*
(2) *There exists an $l \in \{0, \ldots, i-1\}$ with $\mathbf{b}(a_l) = \mathbf{b}(a_i)$ and $\mathbf{r}(a_k) \neq \mathbf{r}(a_i)$ for $k := \max\{l \in \{0, \ldots, i-1\} \mid \mathbf{b}(a_l) = \mathbf{b}(a_i)\}$.*

Example 1. Consider the sequence of memory addresses in Table 2. The sequence contains 8 addresses, each of them consisting again of $b = 1$ bank bit, $r = 2$ row bits, and $c = 2$ column bits. The number of row misses in this sequence is 5. Note that there

Table 3: Sequence of Memory Addresses with Bit-Permutation

	$\mathbf{b}(a)$	$\mathbf{r}(a)$		$\mathbf{c}(a)$		
$\mathbf{a}$	$a_{i,0}$	$a_{i,1}$	$a_{i,2}$	$a_{i,3}$	$a_{i,4}$	
a_0	0	1	0	1	0	miss
a_1	0	1	0	0	0	hit
a_2	1	1	1	1	1	miss
a_3	1	**0**	**0**	1	1	miss
a_4	1	0	0	0	1	hit
a_5	0	1	0	0	0	hit
a_6	1	**1**	0	0	1	miss
a_7	0	1	0	1	1	hit

is no row miss from $i = 4$ to $i = 5$, because bank 0 is already active, i.e., we have $\mathbf{b}(a_l) = \mathbf{b}(a_5)$ for $l \in \{0, 1\}$, and memory address a_5 targets the same row as the last access to this bank, i.e., for $1 = \max\{l \in \{0, \dots, i-1\} \mid \mathbf{b}(a_l) = \mathbf{b}(a_5)\}$ we have $\mathbf{r}(a_1) = \mathbf{r}(a_5)$. A row miss is induced for the first access to a bank and for a change in the row bits highlighted in bold.

Example 2. Consider again the sequence of memory addresses in Table 2. Suppose that we switch the address bits for rows and columns, i.e., we permute the bits according to a permutation σ defined by $\sigma(0) = 0, \sigma(1) = 3, \sigma(2) = 4, \sigma(3) = 1, \sigma(4) = 2$ as illustrated in Table 3. Then the number of row misses decreases by 1.

Thus, the goal of our optimization is to find a permutation of the address bits such that the number of row misses is minimized. To this end, we need the following definition.

Definition 1. Let $a = (a_0, \dots, a_{n-1}) \in \{0,1\}^{n-1}$ be a binary vector and $\sigma\colon \{0, \dots, n-1\} \to \{0, \dots, n-1\}$ a permutation. We call $a^\sigma := (a_{\sigma(0)}, \dots, a_{\sigma(n-1)})$ the *vector obtained from a and* σ. For a sequence of binary vectors $\mathbf{a} = (a_0, \dots, a_{m-1})$ we call $A^\sigma := (a_0^\sigma, \dots, a_{m-1}^\sigma)$ the *sequence obtained from* σ.

We are now ready to formulate our optimization problem for minimizing the number of row misses of a given application-specific memory access pattern: Given a sequence of memory addresses $\mathbf{a} = (a_0, \dots, a_{m-1}), a_i \in \{0,1\}^n, i = 0, \dots, m-1$, and natural numbers $b, r, c \in \mathbb{N}$ denoting the number of bank, row, and column bits, find a permutation $\sigma\colon \{0, \dots, n-1\} \to \{0, \dots, n-1\}$ such that the number of row misses of the sequence $\mathbf{a}^\sigma$ obtained from σ is minimum. We call this problem *Min-Row-Misses in the setting* (b, r, c).

There are $n!$ possibilities for a permutation σ, so there are $5! = 120$ different possibilities for the setting in Example 1. For the case of the realistic configurable address scrambler from Figure 1, where $n = 24$ bits can be permuted, this results in $n! \approx 6.2 \cdot 10^{23}$ different possibilities, which is far too much for a simple enumeration.

To reduce the number of possible permutations, note that permuting the bank, row, and column bits within themselves does not change the number of row misses but only results in a renumbering of the banks, rows, and columns. This observation is formalized in the following.

Observation 2. *Let* $\sigma, \pi\colon \{0, \dots, n-1\} \to \{0, \dots, n-1\}$ *be two permutations with the following properties:*

(1) $\sigma(\{0, \dots, b-1\}) = \pi(\{0, \dots, b-1\})$,
(2) $\sigma(\{b, \dots, b+r-1\}) = \pi(\{b, \dots, b+r-1\})$,
(3) $\sigma(\{b+r, \dots, n-1\}) = \pi(\{b+r, \dots, n-1\})$.

Then the number of row misses of the sequences $\mathbf{a}^\sigma$ *and* $\mathbf{a}^\pi$ *obtained from* σ *and* π*, respectively, are equal.*

A direct consequence of the above observation is that it suffices to partition the set $\{0, \dots, n-1\}$ into three subsets indicating the bank, row, and column bits. This reduces the number of possible permutations to $\binom{n}{b} \cdot \binom{n-b}{r}$, resulting in only $\binom{5}{1} \cdot \binom{4}{2} = 30$ possible permutations for the setting $(b, r, c) = (1, 2, 2)$ from Example 1 and $\binom{24}{3} \cdot \binom{21}{14} \approx 2.4 \cdot 10^8$ possible permutations for the setting $(b, r, c) = (3, 14, 7)$ from Figure 1. Including the XORs from Figure 1 into this consideration will further increase the number of possible permutations (see Section 4.2.4). Thus, a clever strategy for solving *Min-Row-Misses* is needed.

4.2.2 The Artificial Case of a Single Bank. In this section, we will show how to solve *Min-Row-Misses* in the setting (b, r, c) for the special case where $b = 0$, i.e., we assume that our DRAM only consists of a single bank. As described in the next section, the general case can always be reduced to this single bank case.

Following Observation 2, in case of a single bank, we have to choose a set $R^* \subseteq \{0, \dots, n-1\}$ of size r whose elements are going to be the row bits. Clearly, since $b = 0$ and $n = r + c$, the column bits are then determined by $\{0, \dots, n-1\} \backslash R^*$. Recall that there are $\binom{r+c}{r}$ possibilities for the set R^*.

Example 3. Consider the sequence of memory accesses given in Table 4. The address bits, which toggle from one access to the next, are highlighted as bold numbers. Note that this toggling can be observed column-wise. Therefore, for every column $a_{i,0}, \dots, a_{i,4}$ of Table 4, we define sets $S_j, j \in \{0, \dots, 4\}$, containing exactly those $i \in \{0, \dots, 8\}$ for which there is a bit change in $a_{i,j}$, i.e., the highlighted numbers in the table. For the specific case, we get $S_0 = \{1, 6\}$, $S_1 = \{3, 8\}$, $S_2 = \{2, 6, 8\}$, $S_3 = \{1, 3, 4, 5\}$ and $S_4 = \{2, 3, 6, 7\}$. Note that the size of set S_j gives the number of bit changes for the j-th address bit.

Assume that we want to choose $r = 3$ positions as row bits. To obtain the number of row misses when choosing certain row bits, consider the union of three of the sets S_j, e.g., of the sets S_0, S_1 and S_2 given by $T := S_0 \cup S_1 \cup S_2 = \{1, 2, 3, 6, 8\}$. The elements of T are exactly the positions of bit changes happening combined in columns 0, 1 and 2 and the size of T equals the number of row misses of $\mathbf{a}$ minus 1 for $R^* = \{0, 1, 2\}$, since the initial row miss when activating the bank is not counted.

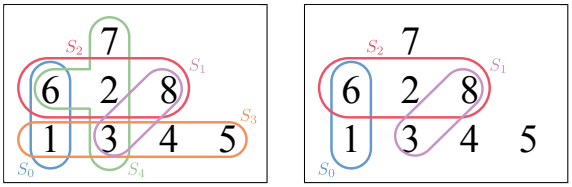
In the following, we will generalize the previous example. To this end, we need the following definition.

Definition 2. Let V be a finite set and $\mathcal{E} \subseteq \mathcal{P}(V)$, where $\mathcal{P}(V)$ denotes the power set of V, be a set of subsets of V. We call the pair $H = (V, \mathcal{E})$ a *hypergraph*, the elements in V *vertices* and the elements of $\mathcal{E}$ *hyperedges*.

Note that, in contrast to edges in a usual graph that connect only two vertices, hyperedges connect an arbitrary subset of V.

Table 4: Sequence of Memory Addresses with Highlighted Bit-Toggling

a	$a_{i,0}$	$a_{i,1}$	$a_{i,2}$	$a_{i,3}$	$a_{i,4}$
a_0	1	0	0	0	1
a_1	**0**	0	0	**1**	1
a_2	0	0	**1**	1	**0**
a_3	0	**1**	1	**0**	**1**
a_4	0	1	1	**1**	1
a_5	0	1	1	**0**	1
a_6	**1**	1	**0**	0	**0**
a_7	1	1	0	0	**1**
a_8	1	**0**	**1**	0	1



(a) Problem (b) Solution

Figure 5: The Hypergraphs Corresponding to Table 4

For a given hypergraph $H = (V, \mathcal{E})$ and natural number $k \in \mathbb{N}$ with $k < |\mathcal{E}|$, the goal of the *Min-k-Union* problem is to choose a subset $\mathcal{F} \subseteq \mathcal{E}$ of size k such that $|\bigcup_{E \in \mathcal{F}} E|$ is minimum.

Theorem 1. *Solving* Min-Row-Misses *in the setting* $(0, r, c)$ *is equivalent to solving* Min-k-Union *with* $k = r$.

A formal proof to Theorem 1 is given in Appendix A.

Example 4. Consider again the setting from Example 3. The corresponding hypergraph with vertices $V = \{1, \ldots, 8\}$, corresponding to the memory accesses from Table 4, and hyperedges $\mathcal{E} = \{S_0, \ldots, S_4\}$, corresponding to the bitchanges of the individual address bits, is illustrated in Figure 5a. It can be verified, that choosing $\mathcal{F} = \{S_0, S_1, S_2\}$ is a solution to *Min-k-Union* for $k = 3$ (see Figure 5b). According to Theorem 1, this means that choosing bits 0, 1 and 2 as row bits is a solution to *Min-Row-Misses* in the setting $(0, 3, 2)$ (see Table 5). Note that the resulting number of row misses equals $|\mathcal{F}| + 1 = 6$.

In [33], Vinterbo showed that the so-called *Max-k-Intersection* problem, which is equivalent to maximizing the number of row hits in the setting $(0, r, c)$ with $r = k$, is NP-hard. Since maximizing the number of row hits is equal to minimizing the number of row misses, we can conclude that *Min-Row-Misses* and, thus, also *Min-k-Union* are NP-hard.

4.2.3 The Case of Multiple Banks. We now show how the general case with multiple DRAM banks, i.e. a realistic DRAM device with 8 or 16 banks, can be reduced to the artificial case of a single bank discussed in the previous section. By Observation 1, we know that whether a memory request admits a row hit or miss only depends on the previous access to the same bank. Hence, we can split our memory access sequence $\mathbf{a}$ into 2^b subsequences, each of which corresponds to a single bank.

Table 5: Partition into Row and Column Bits

	$\mathbf{r}(a)$			$\mathbf{c}(a)$		
a	$a_{i,0}$	$a_{i,1}$	$a_{i,2}$	$a_{i,3}$	$a_{i,4}$	
a_0	1	0	0	0	1	miss
a_1	**0**	0	0	1	1	miss
a_2	0	0	**1**	1	0	miss
a_3	0	**1**	1	0	1	miss
a_4	0	1	1	1	1	hit
a_5	0	1	1	0	1	hit
a_6	**1**	1	**0**	0	0	miss
a_7	1	1	0	0	1	hit
a_8	1	**0**	**1**	0	1	miss

Definition 3. Let $\mathbf{a} = (a_0, \ldots, a_{m-1})$ be a sequence of memory accesses, where each memory access is a length-n bit vector of the form $a_i = (\mathbf{b}(a_i), \mathbf{r}(a_i), \mathbf{c}(a_i))$. For a bit vector $d \in \{0,1\}^b$ let $1 \leq i_0 < i_1 < \ldots < i_g \leq m$ be all indices for which $\mathbf{b}(a_{i_j}) = d$, i.e., all memory accesses with bank bits equal to d. Then we define

$$\mathbf{a}[d] := ((a_{i_0,b}, \ldots, a_{i_0,n-1}), (a_{i_1,b}, \ldots, a_{i_1,n-1}), \ldots, (a_{i_k,b}, \ldots, a_{i_k,n-1}))$$

to be the subsequence of memory accesses with bank bits equal to d. Note that the sequence only includes the row and column bits, while the bank bits are omitted.

Note that the order inside the per-bank subsequences is not altered from the whole sequence, thus, the number of row misses inside each subsequence equals the number of row misses in the corresponding bank. These subsequences $\mathbf{a}[d]$ are then concatenated to a single sequence. The number of row misses in this sequence now reflects the number of row misses in the original sequence - however, errors can occur at the boundaries between the subsequences, as the first access to a bank should always result in a row miss.

Example 5. Consider the sequence of memory accesses $\mathbf{a}$ from Table 2. The result $\tilde{\mathbf{a}}$ after extracting the per-bank subsequences and concatenating them is shown in Table 6. It can be seen that the number of row misses is not equal: While the original sequence from Table 2 yielded 5 row misses, the concatenated sequence without bank bits from Table 6 results in 4 row misses. This is due to the initial row miss for activating Bank 1 not being counted, since the row bits of the last access to Bank 0 and the first access to Bank 1 coincide.

To correctly calculate the number of row misses in the boundary cases, our goal is to eventually invert some of the per-bank sequences $\mathbf{a}[d]$ in case that their first access does not result in a row miss. To this end, we define $\neg a := (\neg a_0, \ldots, \neg a_{n-1})$ to be the negated vector for a binary vector $a \in \{0,1\}^n$. Accordingly, for a sequence of memory accesses, we define $\neg\mathbf{a} := (\neg a_0, \ldots, \neg a_{m-1})$.

Observation 3. *A memory access sequence* $\mathbf{a}$ *induces the same amount of row misses as the negated sequence* $\bar{\mathbf{a}}$.

Table 6: Reduction to a Single Bank

		$\mathbf{r}(a)$		$\mathbf{c}(a)$		
	$\mathbf{a}$	$a_{i,1}$	$a_{i,2}$	$a_{i,3}$	$a_{i,4}$	
Bank 0 $\mathbf{a}[0]$	$\tilde{a}_0$	1	0	1	0	miss
	$\tilde{a}_1$	**0**	0	1	0	miss
	$\tilde{a}_2$	0	0	1	0	hit
	$\tilde{a}_3$	**1**	**1**	1	0	miss
Bank 1 $\mathbf{a}[1]$	$\tilde{a}_4$	1	1	1	1	**hit**
	$\tilde{a}_5$	1	1	0	0	hit
	$\tilde{a}_6$	**0**	1	0	0	miss
	$\tilde{a}_7$	0	1	1	0	hit

Table 7: Reduction to a Single Bank with the Negated Sequence

		$\mathbf{r}(a)$		$\mathbf{c}(a)$		
	$\mathbf{a}$	$a_{i,1}$	$a_{i,2}$	$a_{i,3}$	$a_{i,4}$	
Bank 0 $\bar{\mathbf{a}}_0$	$\tilde{a}_0$	1	0	1	0	miss
	$\tilde{a}_1$	**0**	0	1	0	miss
	$\tilde{a}_2$	0	0	1	0	hit
	$\tilde{a}_3$	**1**	**1**	1	0	miss
Bank 1 $\bar{\mathbf{a}}_1$	$\neg\tilde{a}_4$	0	0	0	0	**miss**
	$\neg\tilde{a}_5$	0	0	1	1	hit
	$\neg\tilde{a}_6$	**1**	0	1	1	miss
	$\neg\tilde{a}_7$	1	0	0	1	hit

Definition 4. Let $\mathbf{a}$ be a sequence of memory accesses and for $k = 2^b$ let $\{d_0, \ldots, d_{k-1}\}$ be the set of all bit-vectors of length b. Then we define $\bar{\mathbf{a}} := \bar{\mathbf{a}}_0 \circ \bar{\mathbf{a}}_1 \circ \cdots \circ \bar{\mathbf{a}}_{k-1}$ as the sequence obtained by concatenating $\bar{\mathbf{a}}_0, \ldots, \bar{\mathbf{a}}_{k-1}$, where

$$\bar{\mathbf{a}}_i := \begin{cases} \mathbf{a}[d_i], & \text{if } i = 1 \text{ or the last row bits of } \mathbf{a}[d_{i-1}] \text{ differ} \\ & \text{from the first row bits of } \mathbf{a}[d_i], \\ \neg\mathbf{a}[d_i] & \text{otherwise.} \end{cases}$$

Note that, by definition of $\bar{\mathbf{a}}$, the first request to $\bar{\mathbf{a}}_j$, $j = 0, \ldots, k$, always induces a row miss. By Observation 1, we know that whether a memory access a_i induces a row miss only depends on the last access to the same bank. Hence, the number of row misses of $\mathbf{a}$ in the setting (b, r, c) in the bank corresponding to the bit vector $d \in \{0,1\}^b$ equals the number of row misses of $\mathbf{a}[d]$ in the setting $(0, r, c)$ and thus, by Observation 3, also the number of row misses of $\neg\mathbf{a}[d]$ in the setting $(0, r, c)$.

Observation 4. *The number of row misses of the sequence* $\mathbf{a}$ *in the setting* (b, r, c) *equals the number of row misses of the sequence* $\bar{\mathbf{a}}$ *in the setting* $(0, r, c)$.

Example 6. Following Example 5, Table 7 depicts the sequence $\bar{\mathbf{a}}$, where all accesses to Bank 1 are negated. It can be seen that the initial access to Bank 1 is now correctly counted as a row miss and that the total number of row misses is now equal to the total number of row misses in the original sequence from Table 2.

It follows from Observation 4 that, once we know which bits are the bank bits, the problem is reduced to the case of a single bank from Section 4.2.2 and can thus be solved by solving *Min-k-Union*.

There are

$$\binom{n}{b}$$

different possibilities for choosing the bank bits. In the setting of the configurable address scrambler from Figure 1, this means that there are $\binom{24}{3} = 2024$ different possibilities for choosing the bank bits when omitting the XOR gates.

4.2.4 Using the XOR Gates. To be compliant with the targeted configurable address scrambler in Figure 1, up to now, we would be able to find a solution for the 24 MUX-24 that perform the permutation σ, which is the solution to the *Min-k-Union*. Now, as a last step, the configuration for the x XOR gates has to be found (in the figure, $x = 3$). This step is done by a simple enumeration. To be precise, we need to choose x bank bits and x row bits, i.e., in total $2x$ bits, that will be connected by an XOR. Note that the order of choosing these bits matters, as interchanging the bank and row bits, or XORing different bank and row bits changes the result. There are $\frac{n!}{(n-2x)!}$ different possibilities of choosing $2x$ bits out of n bits. However, as we will see in the following example, some of these combinations yield the same result.

Example 7. Assume that we have chosen $2x$ bits in such a way, that the first x bits $a_0, a_1, \ldots, a_{x-1}$, corresponding to the bank bits, are to be XORed with the second x bits $a_x, a_{x+1}, \ldots, a_{2x-1}$, corresponding to the row bits. Then, any permutation within the first x bank bits, that is also applied to the second x row bits, only permutes the result within the bank and row bits. By Observation 2, this results in the same number of row misses and can thus be neglected.

Following this example, there are $x!$ different possibilities to choose a permutation within x bits. Thus, the total number of choosing the XOR bits reduced to

$$\frac{n!}{x!(n-2x!)}.$$

The overall number of possible configurations of a configurable address scrambler can now be computed by

$$\frac{n!}{x!(n-2x)!} \cdot \binom{n-2x}{b-x} \cdot \binom{n-x-b}{r-x},$$

where the first factor accounts for choosing the $2x$ bits that are connected by an XOR gate, the second factor accounts for choosing the remaining bank bits in the case that $x \neq b$, and the third factor accounts for choosing the remaining $r - x$ row bits out of the remaining total number of bits. For the setting in Figure 1, where $(b, r, c) = (3, 14, 7)$ and $x = 3$, there exist $\sim 5.1 \cdot 10^{11}$ different meaningful configurations.

4.2.5 The Branch and Bound Method for Solving Min-Row-Misses. In this section, we will present our algorithm for finding a configuration of the address scrambler that results in a minimum number of row misses for a given memory access sequence. The heart of this algorithm is a branch and bound method for solving the *Min-k-Union* problem. Recall that, by Theorem 1, solving *Min-k-Union* is equivalent to solving *Min-Row-Misses* in the case of a single DRAM bank. Therefore, our algorithms consists of two steps:

- Enumerate all possibilities for choosing the XOR gates (and the remaining bank bits in the case that $x \neq b$). For each possibility, construct the corresponding one-bank sequence $\bar{\mathbf{a}}$ as described in Section 4.2.3.
- Solve *Min-k-Union* for the sequence $\bar{\mathbf{a}}$ using a branch and bound algorithm.

For the first step, there are

$$\frac{n!}{x!(n-2x)!} \cdot \binom{n-2x}{b-x}$$

different possibilities (cf. Section 4.2.4) for reducing the input sequence to a one-bank sequence. For the configurable address scrambler in Figure 1, this results in 16.151.520 different sequences $\bar{\mathbf{a}}$.

For each of these sequences, the *Min-k-Union* problem has to be solved. To the best of our knowledge, no work has been done so far in solving *Min-k-Union* computationally, so we will present our algorithm in the following. Since *Min-k-Union* is shown to be NP-hard, not surprisingly, the worst case running time is exponential in the size of the input, i.e., the length m of the memory access sequence and the number n of address bits per access.

Our algorithms uses the branch and bound method, i.e., it spans up an enumeration tree, repeatedly computes upper bounds on the optimal solution, and prunes parts of the tree that do not contain the optimal solution in order to reduce the search space. To construct the enumeration tree, let $H = (V, \mathcal{E})$ be the hypergraph with hyperedges $\mathcal{E} = \{E_0, \ldots, E_{m-1}\}$ corresponding to the input memory access sequence (cf. Section 4.2.2). For $i \in \{0, \ldots, m-1\}$, level i of the enumeration tree corresponds to the hyperedge E_i, i.e., depending on which branch we choose at level i, the hyperedge E_i is included in our solution set $\mathcal{F}$ or not. Recall that the goal of *Min-k-Union* is to find a subset $\mathcal{F}$ of $\mathcal{E}$ of size k such that $|\bigcup_{E \in \mathcal{F}} E|$ is minimum.

There are two ways to prune the enumeration tree: *pruning by infeasibility* and *pruning by bound*. To prune by infeasibility, assume that we are currently examining level i of the enumeration tree. Then at most $m - i$ further hyperedges can be included to $\mathcal{F}$. Thus, if $|\mathcal{F}| + m - i < k$, then there are not enough hyperedges left to ensure that the final set $\mathcal{F}$ contains k elements, so the corresponding subtree does not contain a feasible solution and can be pruned. To prune by bound, suppose we have an upper bound α on the optimal value of *Min-k-Union* for the hypergraph H. If, at any point in the enumeration tree, the union of the hyperedges contained in the set $\mathcal{F}$ already contains α or more elements, i.e., $|\bigcup_{E \in \mathcal{F}} E| \geq \alpha$, then no better solution can be found in the current subtree, since adding more hyperedges to $\mathcal{F}$ only worsens the result. Thus, there is no need to further examine the current subtree and it can be pruned. Upper bounds are obtained by two ways in the algorithm: First, every solutions for *Min-k-Union* applied to the one-bank-sequence corresponding to some enumeration for the XOR and bank bits yields an upper bound for the whole optimization problem. Second, for every enumeration for the XOR and bank bits, we create a starting solution for *Min-k-Union* by choosing as row bits exactly those bits that have a minimum toggling rate (cf. Figure 3f).

The whole algorithm is presented in Algorithm 2. Algorithm 3 presents the branch and bound subroutine for repeatedly solving the *Min-k-Union* problem.

Algorithm 2 Solving Procedure

Input: A sequence A of n-dimensional bit-vectors, natural numbers $x, b, r, c \in \mathbb{N}$ with $b + r + c = n$
Output: A partition into sets B_x, R_x, B, R as a solution to *Min-Row-Misses* in the setting (b, r, c) using x XOR gates

$\alpha := \infty$
$B_x, R_x, B, R \leftarrow$ none
for all possibilities of choosing the XOR bits B_x^*, R_x^* and the remaining bank bits B^* **do**
  Create the corresponding one-bank sequence $\bar{\mathbf{a}}$, where the bits corresponding to R_x^* are omitted
  Let $H = (V, \mathcal{E})$ be the hypergraph corresponding to $\bar{\mathbf{a}}$
  Let $\mathcal{E} = \{E_0, \ldots, E_{c+r-x-1}\}$ with $|E_0| \leq \ldots \leq |E_{c+r-x-1}|$
  $\mathcal{F}^* = \{E_0, \ldots, E_{r-x-1}\}$
  $\alpha^* := |\bigcup_{E \in \mathcal{F}^*} E \cup \bigcup_{j \in R_x^*} E_j|$
  $(\mathcal{F}', \alpha') :=$ BBRecursion$(H, \emptyset, r - x, \min\{\alpha, \alpha^*\}, R_x^*)$
  if $\min\{\alpha', \alpha^*\} < \alpha$ **then**
    $B \leftarrow B^*$
    $B_x \leftarrow B_x^*$
    $R_x \leftarrow R_x^*$
    if $\alpha' < \alpha$ **then**
      $\alpha \leftarrow \alpha'$
      Let R be the indices corresponding to $\mathcal{F}'$
    else if $\alpha^* < \alpha$ **then**
      $\alpha \leftarrow \alpha^*$
      Let R be the indices corresponding to $\mathcal{F}^*$

Algorithm 3 BB-Recursion

Input: A hypergraph $H = (V, \mathcal{E})$, a collection of hyperedges $\mathcal{F}$, a natural number $k \in \mathbb{N}$ indicating how many more hyperedges have to be included to $\mathcal{F}$, the value of the currently best solution $\alpha \in \mathbb{N}$, and a set of indices R_x corresponding to the row bits that are connected via an XOR gate
Output: A set $\mathcal{F}$ representing the optimum solution of *Min-k-Union* and the corresponding objective value α

if $k = 0$ **then**
  return $(\mathcal{F}, |\bigcup_{E \in \mathcal{F}} E \cup \bigcup_{j \in R_x} E_j|)$
$\alpha_1 := \alpha_2 := \infty$
Choose $E_1 \in \mathcal{E}$
if $|\bigcup_{E \in \mathcal{F}} E \cup \bigcup_{j \in R_x} E_j \cup E_1| < \alpha$ **then**
  $(\mathcal{F}_1, \alpha_1) :=$ BBRecursion$(H - E_1, \mathcal{F} \cup \{E_1\}, k - 1, \alpha, R_x)$
if $|\mathcal{F}| + m - 1 \geq k$ **then**
  $(\mathcal{F}_2, \alpha_2) :=$ BBRecursion$(H - E_1, \mathcal{F}, k, \alpha, R_x)$
if $\min\{\alpha_1, \alpha_2\} < \alpha$ **then**
  if $\alpha_1 < \alpha_2$ **then**
    return $(\mathcal{F}_1, \alpha_1)$
  else
    return $(\mathcal{F}_2, \alpha_2)$
else
  return $(\mathcal{F}, \alpha)$

5 RESULTS

The results of the Steps I and II of our approach can be used to set up the configurable address scrambler shown in Figure 1 for a memory controller with write buffer, in order to reduce the number of row misses.

To demonstrate the practical applicability we conducted several experiments with the same bandwidth demanding applications used in [12]. All applications produce deterministic memory access pattern, which can be found in typical embedded systems applications:

- **Image Rotation:** An image with a resolution of 1024×576 pixels is rotated by 90°. Therefore, the image is written into the DRAM in x-direction and is read in y-direction, which results in a large number of row misses for a BRC mapping. We varied the pixel size from 8 to 256 bit.
- **Convolutional Neural Network (CNN):** An optical neighborhood operation is performed on an image with a resolution of 1024×576 pixels. We varied the kernel size between 3×3 and 11×11 and pixel size from 8 to 64 bit.
- **3D Image Rotation:** A three-dimensional image with 128 ×128×128 voxels is rotated. We varied the pixel size between 8 and 128 bit.

To generate the appropriate scrambler configurations the Min-k-Union solver was implemented using Python and OpenMPI, which enabled a parallel execution on a high performance computing cluster. In addition to the configurable address scrambler, a read-write buffer with depth of 32 was used to minimize the number of read-write switches.

For the validation of our approach we performed 18×5 runs of Algorithm 2 and 211261 DRAM simulations. There exist several solutions with a minimum number of row misses, as shown in Figure 7. It can be observed that there is only one solution for configurations using no XOR gates. If at least one XOR gates is used, there is always more than one solution. This number varies largely depending on the application. In our tests it ranges from 2 to over 65000.

Since we have mostly more than one solution with respect to the minimum number of row misses, all possible configurations are tested using DRAMSys [14] to determine both the bandwidth usage and the energy consumption of each and therefore to select the optimal solution with respect to bandwidth usage and energy consumption. As shown in Figure 8, the solution quality may vary despite the same minimal number of row misses. This can be explained by looking at the remaining row misses that might be masked by bank parallelism. For example, for the *CNN* applications all solutions show similar bandwidth usage and energy consumption behaviour, whereas Rotation and *3D-Rotation* show a large difference especially for large pixel quantization. In some solutions the row misses are well hidden, since there are read or write accesses on other banks executed in parallel. In the worst solutions most row misses stall the transfer, because the bank level parallelism is not exploited. Thus, their delay has a greater impact on the bandwidth usage and energy consumption.

The benefit of enabling the XOR gates is presented in Figure 9g and 9h for three example benchmarks, namely *3D Rotation 128*, *Filter 11x11x8* and *Rotation 256*. If the application features for example a stride access pattern, a simple permutation without XORs is often insufficient to achieve a near maximum bandwidth usage. This is the case for the Image *Rotation* and *3D Image Rotation* benchmarks. With three enabled XORs ConGen2 is able to increase the bandwidth by over 20% while decreasing the energy consumption. However, the access pattern of all *Filter* benchmarks fits quite well to the DRAM structure because it is primarily a linear access pattern. Therefore, the default address mappings, like RBC, achieve already a high bandwidth usage and more enabled XORs cannot bring any further improvement. This is the reason why it is necessary to test all XOR combinations.

In Figure 9 we selected the best solution for each application with respect to the highest bandwidth usage (CONGEN2-BW) and lowest energy consumption (CONGEN2-E) to compare them with the state-of-the-art address mappings, scheduling (FRFCFS) [28] and the original ConGen1 [12]. It can be observed that we improved our results in every application compared to the predecessor ConGen1 with much less hardware overhead, since ConGen2 is an exact solver under hardware constraints, unlike ConGen1, which is an hardware unconstrained heuristic.

Both standard mappings *Row-Bank-Column* (RBC) and *Bank-Row-Column* (BRC) show for all *Rotation* and *3D-Rotation* applications the same results independent of the scheduler used. However, for the *CNN* applications the FRFCFS scheduler reduced the bandwidth by more than 25% and highly increased the energy consumption if a BRC mapping is used. Furthermore, it can be observed, that the state-of-the-art RBC mapping is able to get close to the optimal ConGen2 solution for all *CNN* applications and *Rotation* applications with small pixel quantization. In contrast, none of the standard mappings performs well for *3D-Rotation* applications.

In general, it can be clearly seen, that ConGen2 outperforms the state of the art online scheduling scheduler by exploiting the full application knowledge. The missing gap towards 100% bandwidth, is due to the refresh overhead.

Furthermore, both solutions *CONGEN-BW* and *CONGEN-E* do not differ significantly in any application and are able to reach more than 95% bandwidth usage without any fluctuations or drops as it can be seen with *Row-Bank-Column* (RBC) and *Bank-Row-Column* (BRC) mappings.

Overall, the generated configurations outperform every other address mapping and scheduler in each application. We achieved an energy reduction up to 88% and a bandwidth increase of a factor 8.9 compared to a RBC mapping.

6 CHIP IMPLEMENTATION

The proposed configurable address scrambler (Figure 1) is implemented in a DDR3 memory controller which is presented in [32]. This controller is integrated in a RISC-V based chip, which is implemented in a UMC 65 nm technology, that targets transprecision computing [21]. Thus, the DDR3 memory controller is designed for low power and low latency embedded applications. The proposed ConGen2 methodology plays an important role for these applications and the proposed configurable address scrambler ensures that the DRAM has minimum number of page misses for a given application and therefore a high energy efficiency. The controller

and it's PHY, operate at a maximum frequency of 500 Mhz, consume an average power of 129.33 mW and a total area of 4.71 mm^2. Figure 6 shows the floorplan of our DDR3 memory controller, consisting of the PHY and the controller logic which includes the configurable address scrambler. The address scrambler occupies an area of 2292.1 μm^2, which is negligibly compared to other blocks of the controller. The proposed configurable address scrambler has a latency of 1.06 ns.

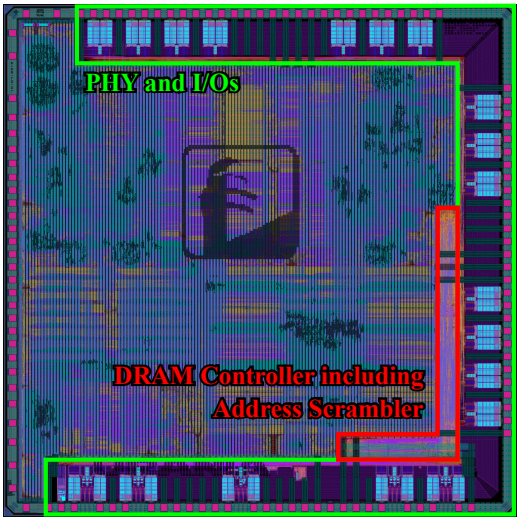


Figure 6: Chip Implementation

7 CONCLUSION

We presented a new methodology to automatically generate an ASMC, which has a global view on the application and can therefore exploit application knowledge to decrease energy consumption and increase bandwidth. The approach consists of two parts, where the first part minimizes the penalty of RD/WR switches, whereas the second part minimizes the number of row misses by using a configurable address scrambler. We showed that the problem to find the optimal configuration for this scrambler for a given application is equivalent to the NP-hard Min-k-Union problem and presented an exact algorithm to efficiently solve this problem. In a last step, we demonstrated the advantages of our approach by using typical DRAM access sequences of today's embedded systems. Overall, our approach could outperform every other address mapping and scheduler. We achieved an energy reduction up to 88% and a bandwidth increase of a factor 8.9 compared to a typical RBC mapping.

A PROOF OF THEOREM 1

Proof. *Min-Row-Misses is an instance of Min-k-Union:* Let $H = (V, \mathcal{E})$ be a hypergraph with vertex set $V = \{v_0, \dots, v_{m-1}\}$ and hyperedges $\mathcal{E} = \{E_0, \dots, E_{n-1}\}$. Moreover, let $k \in \mathbb{N}$ with $k \leq |\mathcal{E}|$. We construct the sequence $\mathbf{a} = (a_0, a_1, \dots, a_{m-1})$ as follows: $a_0 := (0, \dots, 0) \in \{0,1\}^n$ consists of exactly n zeros; for $i = 0, \dots, m-1$, we define $a_i := (a_{i,0}, \dots, a_{i,n-1})$ where

$$a_{i,j} := \begin{cases} \neg a_{i-1,j}, & \text{if } v_i \in E_j \\ a_{i-1,j}, & \text{otherwise.} \end{cases}$$

Finally, we set $b := 0$, $r := k$ and $c := n - r$. Let σ be a permutation of $\{0, \dots, n-1\}$ and consider the sequence $\mathbf{a}^\sigma$. The request of a_i^σ, $i \in \{1, \dots, m-1\}$, induces a row miss if and only if $\mathbf{r}(a_{i-1}^\sigma) \neq \mathbf{r}(a_i^\sigma)$. This is true if and only if there is a $j \in \{0 \dots, r-1\}$ with $a_{i,\sigma(j)} \neq a_{i-1,\sigma(j)}$. By construction, this is the case if and only if $v_i \in E_{\sigma(j)}$. Hence, the number of row misses in $\mathbf{a}^\sigma$ is equal to $|\bigcup_{j=0}^{r-1} E_{\sigma(j)}| + 1$.

Min-k-Union is an instance of Min-Row-Misses: Now, conversely, let $\mathbf{a} = (a_0, \dots, a_{m-1})$ be a sequence of n-dimensional bit-vectors in the setting $(0, r, c)$. Let $H = (V, \mathcal{E})$ be the hypergraph with vertices $V = \{1, \dots, m-1\}$ and hyperedges $\mathcal{E} = \{E_0, \dots, E_{n-1}\}$ where vertex $i \in E_j$ if and only if $a_{i-1,j} \neq a_{i,j}$. Moreover, let $\{F_{j_0}, \dots, F_{j_{k-1}}\} \subseteq \mathcal{E}$ be a subset of the hyperedges of size $k = r$. Finally, let σ be a permutation of $\{0, \dots, n-1\}$ with $\sigma(l) = j_l$ for $l = 0, \dots, r-1$. Clearly, $i \in \bigcup_{l=0}^{k-1} E_{j_l}$ if and only if there is a $l \in \{0, \dots, r-1\}$ with $i \in E_{j_l}$. By construction this holds if and only if $a_{i-1,\sigma(l)} = a_{i-1,j_l} \neq a_{i,j_l} = a_{i,\sigma(l)}$. This in turn is equivalent to $\mathbf{r}(a_{i-1}^\sigma) \neq \mathbf{r}(a_i^\sigma)$ and the request to a_i incurs a row miss. Thus, $|\bigcup_{l=0}^{k-1} E_{j_l}| - 1$ is equal to the number of row misses in $\mathbf{a}^\sigma$ in the setting $(0, r, c)$. □

ACKNOWLEDGMENT

This work was supported within the Fraunhofer and DFG cooperation programme (Grant no. WE2442/14-1) and supported by the *Fraunhofer High Performance Center for Simulation- and Software-based Innovation*. Simulations and solver runs were conducted on the high performance cluster *Elwetritsch* at Technische Universität Kaiserslautern, which is part of the *Alliance of High Performance Computing Rhineland-Palatinate* (AHRP). Furthermore, we thank Synopsys and the anonymous reviewers for their support.

REFERENCES

[1] B. Akin, J. C. Hoe, and F. Franchetti. 2014. HAMLeT: Hardware accelerated memory layout transform within 3D-stacked DRAM. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*. 1–6. https://doi.org/10.1109/HPEC.2014.7040954

[2] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. 2012. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 416–427. http://dl.acm.org/citation.cfm?id=2337159.2337207

[3] E. Azarkhish, C. Pfister, D. Rossi, I. Loi, and L. Benini. 2016. Logic-Base Interconnect Design for Near Memory Computing in the Smart Memory Cube. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* PP, 99 (2016), 1–14. https://doi.org/10.1109/TVLSI.2016.2570283

[4] Samuel Bayliss and George A. Constantinides. 2011. Application Specific Memory Access, Reuse and Reordering for SDRAM. In *Proceedings of the 7th International Conference on Reconfigurable Computing: Architectures, Tools and Applications (ARC'11)*. Springer-Verlag, Berlin, Heidelberg, 41–52. http://dl.acm.org/citation.cfm?id=1987535.1987544

[5] Mahdi Nazm Bojnordi and Engin Ipek. 2012. PARDIS: A Programmable Memory Controller for the DDRx Interfacing Standards. *SIGARCH Comput. Archit. News* 40, 3 (June 2012), 13–24. https://doi.org/10.1145/2366231.2337162

[6] John Carter, Wilson Hsieh, Leigh Stoller, Mark. Swanson, Lixin Zhang, Erik. Brunvand, Al. Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, Lambert Schaelicke, and Terry Tateyama. 1999. Impulse: building a smarter memory controller. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*. 70–79. https://doi.org/10.1109/HPCA.1999.744334

[7] Ren Chen and Viktor K. Prasanna. 2015. DRAM Row Activation Energy Optimization for Stride Memory Access on FPGA-Based Systems. In *Applied Reconfigurable*

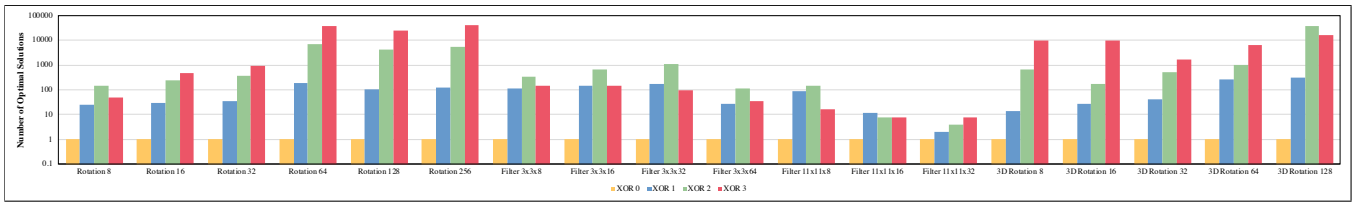
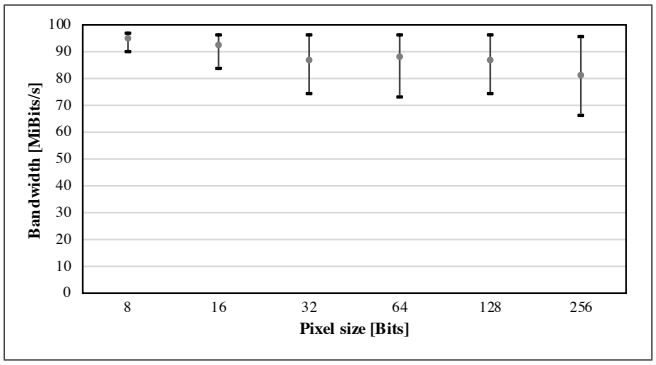
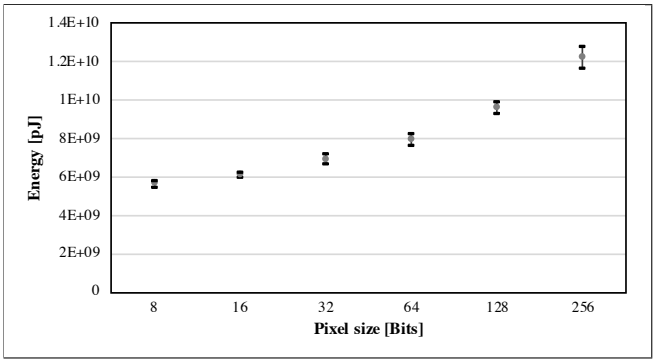


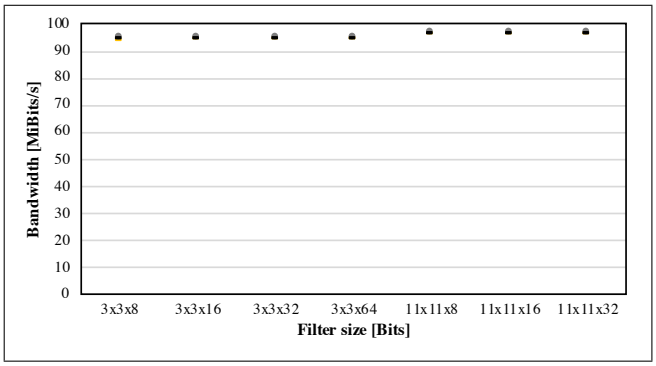
Figure 7: Number of Solutions with a Minimum Number of Row Misses



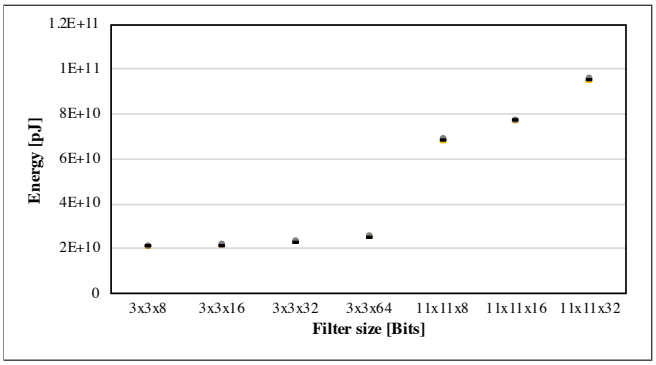
(a) Rotation Bandwidth



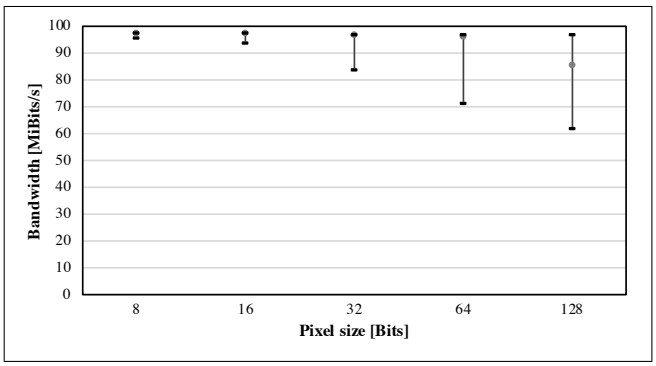
(b) Rotation Energy



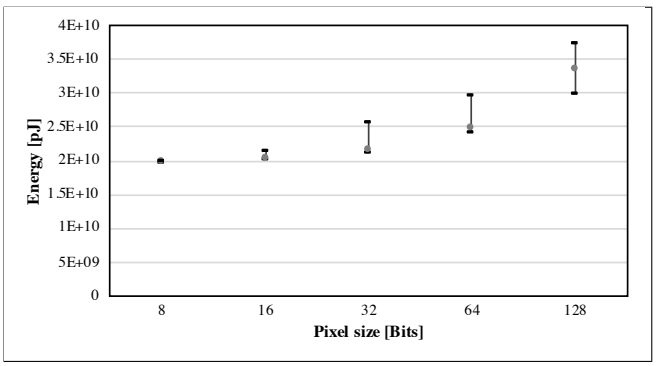
(c) Filter Bandwidth



(d) Filter Energy

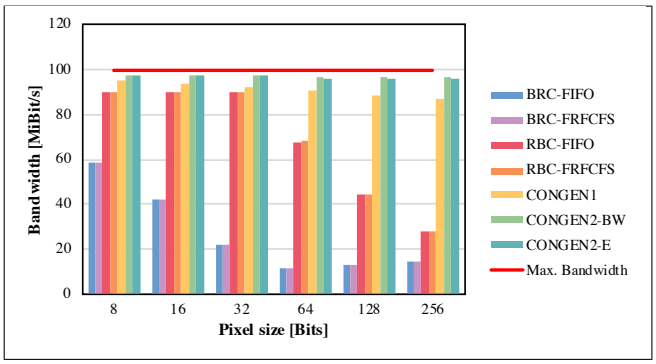


(e) 3D-Rotation Bandwidth

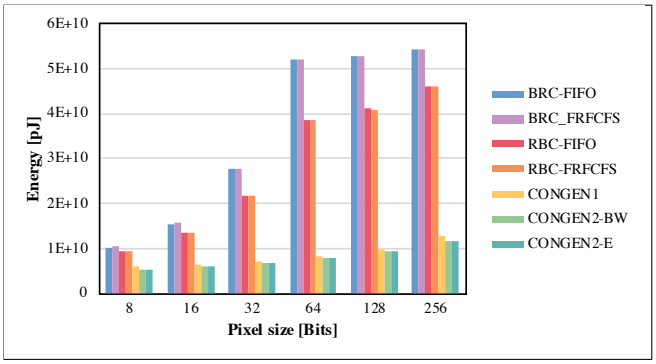


(f) 3D-Rotation Energy

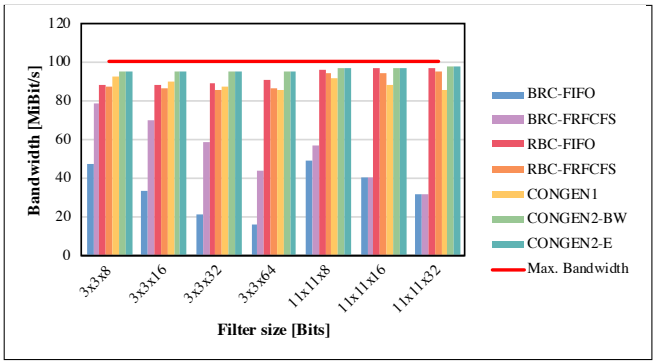
Figure 8: Distributions of Results for Bandwidth and Energy



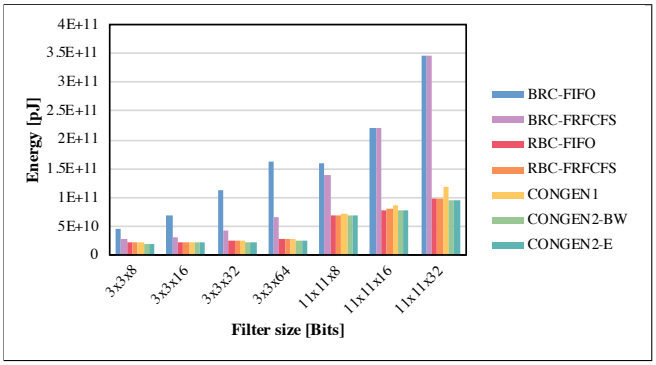
(a) Rotation Bandwidth



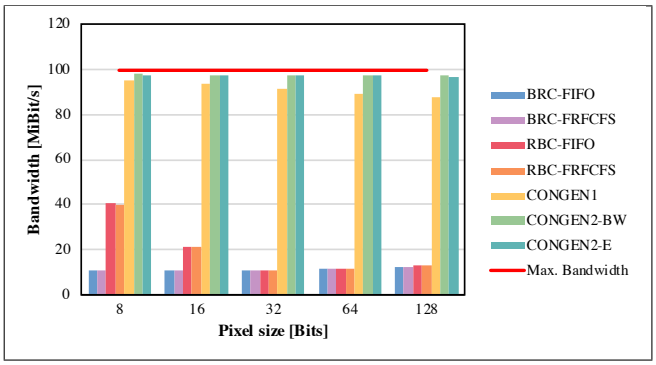
(b) Rotation Energy



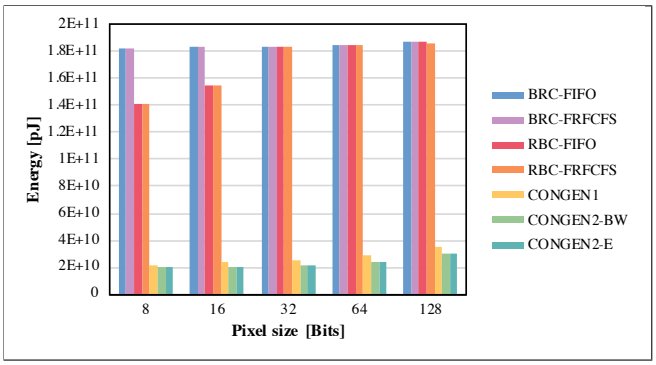
(c) Filter Bandwidth



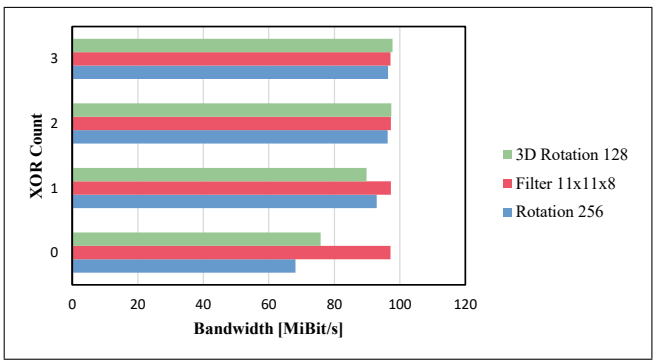
(d) Filter Energy



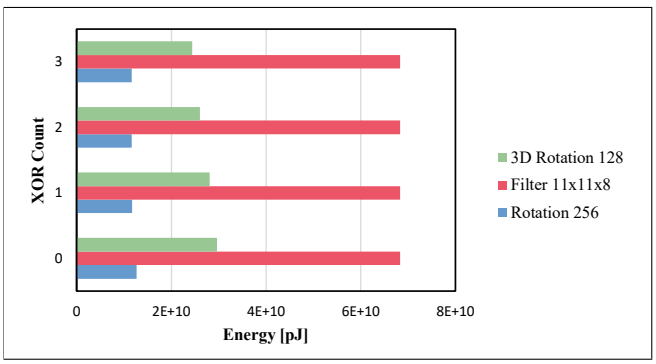
(e) 3D-Rotation Bandwidth



(f) 3D-Rotation Energy



(g) XOR Comparison Bandwidth



(h) XOR Comparison Energy

Figure 9: Results for Bandwidth and Energy and Analysis of the Used Number of XORs

Computing - 11th International Symposium, ARC 2015, Bochum, Germany, April 13-17, 2015, Proceedings. 349–356. https://doi.org/10.1007/978-3-319-16214-0

[8] Mohsen Ghasempour, Aamer Jaleel, Jim D. Garside, and Mikel Luján. 2016. DReAM: Dynamic Re-arrangement of Address Mapping to Improve the Performance of DRAMs. In *Proceedings of the Second International Symposium on Memory Systems (MEMSYS '16)*. ACM, New York, NY, USA, 362–373. https://doi.org/10.1145/2989081.2989102

[9] Ibrahim Hur and Calvin Lin. 2004. Adaptive History-Based Memory Schedulers. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 37)*. IEEE Computer Society, Washington, DC, USA, 343–354. https://doi.org/10.1109/MICRO.2004.4

[10] J. Y. Hur, S. W. Rhim, B. H. Lee, and W. Jang. 2019. Adaptive Linear Address Map for Bank Interleaving in DRAMs. *IEEE Access* 7 (2019), 129604–129616.

[11] Bruce Jacob, S. Ng, and D. Wang. 2010. *Memory Systems: Cache, DRAM, Disk*. Elsevier Science.

[12] Matthias Jung, Irene Heinrich, Marco Natale, Deepak M. Mathew, Christian Weis, Sven Krumke, and Norbert Wehn. 2016. ConGen: An Application Specific DRAM Memory Controller Generator. In *Proceedings of the Second International Symposium on Memory Systems (MEMSYS '16)*. ACM, New York, NY, USA, 257–267. https://doi.org/10.1145/2989081.2989131

[13] Matthias Jung, Christian Weis, Patrick Bertram, and Norbert Wehn. 2013. Power Modelling of 3D-Stacked Memories with TLM2.0 based Virtual Platforms. In *Synopsys User Group Conference (SNUG), May, 2013, Munich, Germany*.

[14] Matthias Jung, Christian Weis, and Norbert Wehn. 2015. DRAMSys: A flexible DRAM Subsystem Design Space Exploration Framework. *IPSJ Transactions on System LSI Design Methodology (T-SLDM)* (August 2015). https://doi.org/10.2197/ipsjtsldm.8.63

[15] Matthias Jung, Éder Zulian, Deepak Mathew, Matthias Herrmann, Christian Brugger, Christian Weis, and Norbert Wehn. 2015. Omitting Refresh - A Case Study for Commodity and Wide I/O DRAMs. In *1st International Symposium on Memory Systems (MEMSYS 2015)*. Washington, DC, USA.

[16] H. S. Kim, N. Vijaykrishnan, M. Kandemir, E. Brockmeyer, F. Catthoor, and M. J. Irwin. 2003. Estimating influence of data layout optimizations on SDRAM energy consumption. In *Low Power Electronics and Design, 2003. ISLPED '03. Proceedings of the 2003 International Symposium on*. 40–43. https://doi.org/10.1109/LPE.2003.1231832

[17] Tim Kogel. 2016. Optimizing DDR Memory Subsystem Efficiency - The Unpredictable Memory Bottleneck. *Synopsys Inc.* (January 2016).

[18] S. Langemeyer, P. Pirsch, and H. Blume. 2011. Using SDRAMs for two-dimensional accesses of long 2n x 2m-point FFTs and transposing. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*. 242–248. https://doi.org/10.1109/SAMOS.2011.6045467

[19] Wei-Fen Lin, S.K. Reinhardt, and D. Burger. 2001. Reducing DRAM latencies with an integrated memory hierarchy design. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*. 301–312. https://doi.org/10.1109/HPCA.2001.903272

[20] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu. 2013. An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms. *SIGARCH Comput. Archit. News* 41, 3 (June 2013), 60–71. https://doi.org/10.1145/2508148.2485928

[21] A. C. I. Malossi, M. Schaffner, A. Molnos, L. Gammaitoni, G. Tagliavini, A. Emerson, A. TomÃąs, D. S. Nikolopoulos, E. Flamand, and N. Wehn. 2018. The transprecision computing paradigm: Concept, design, and applications. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1105–1110. https://doi.org/10.23919/DATE.2018.8342176

[22] Cadence Inc. 2014, last access 18.02.2015. Cadence Denali DDR Memory IP. http://ip.cadence.com/ipportfolio/ip-portfolio-overview/memory-ip/ddr-lpddr. (October 2014, last access 18.02.2015).

[23] Micron Technology Inc. 2006. 1Gb: x4, x8, x16 DDR3 SDRAM. (July 2006).

[24] Synopsys, Inc. 2015, Last Access: 18.02.2015. DesignWare DDR IP. http://www.synopsys.com/IP/InterfaceIP/DDRn/Pages/. (2015, Last Access: 18.02.2015).

[25] Xilinx, Inc. 2015, Last Access: 18.02.2015. Memory Interface Generator (MIG). http://www.xilinx.com/products/intellectual-property/mig.html. (2015, Last Access: 18.02.2015).

[26] Wei Mi, Xiaobing Feng, Jingling Xue, and Yaocang Jia. 2010. Software-hardware Cooperative DRAM Bank Partitioning for Chip Multiprocessors. In *Proceedings of the 2010 IFIP International Conference on Network and Parallel Computing (NPC'10)*. Springer-Verlag, Berlin, Heidelberg, 329–343. http://dl.acm.org/citation.cfm?id=1882011.1882045

[27] Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-Aware Batch-Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *35th International Symposium on Computer Architecture (ISCA)*. Association for Computing Machinery, Inc. http://research.microsoft.com/apps/pubs/default.aspx?id=79626

[28] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. 2000. Memory Access Scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*. ACM, New York, NY, USA, 128–138. https://doi.org/10.1145/339647.339668

[29] Tomas Rockicki. 1996. Indexing memory banks to maximize page mode hit percentage and minimize memory latency. *Hewlett-Packard Laboratories Technical Report, HPL-96-95* (1996).

[30] Vivek Seshadri, Thomas Mullins, Amirali Boroumand, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2015. Gather-scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 267–280. https://doi.org/10.1145/2830772.2830820

[31] Jun Shao and Brian T. Davis. 2005. The Bit-reversal SDRAM Address Mapping. In *Proceedings of the 2005 Workshop on Software and Compilers for Embedded Systems (SCOPES '05)*. ACM, New York, NY, USA, 62–71. https://doi.org/10.1145/1140389.1140396

[32] Chirag Sudarshan, Jan Lappas, Christian Weis, Deepak M. Mathew, Matthias Jung, and Norbert Wehn. 2019. A Lean, Low Power, Low Latency DRAM Memory Controller for Transprecision Computing. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Dionisios N. Pnevmatikatos, Maxime Pelcat, and Matthias Jung (Eds.). Springer International Publishing, Cham, 429–441.

[33] S.A. Vinterbo. 2002. *Maximum k-Intersection, Edge Labeled Multigraph Max Capacity k-Path, and Max Factor k-gcd are all NP-hard*. Technical Report. Decision Systems Group, Harvard Medical School.

[34] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. 2000. A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 33)*. ACM, New York, NY, USA, 32–41. https://doi.org/10.1145/360128.360134