ABSTRACT

Title of dissertation: BUFFER-ON-BOARD MEMORY SYSTEM

Elliott Cooper-Balis,
Doctor of Philosophy, 2012

Dissertation directed by: Professor Dr. Bruce Jacob
Dept of Electrical & Computer Engineering

The design and implementation of the commodity memory architecture has resulted in significant limitations in a system's speed and capacity. To circumvent these limitations, designers and vendors have begun to place intermediate logic between the CPU and DRAM. This additional logic has two functions: to control the DRAM and to communicate with the CPU over a fast and narrow bus. The benefit provided by this logic is a reduction in pin-out to the memory system from the CPU and increased signal integrity seen by the DRAM, granting faster clock rates while increasing capacity. This new design is reminiscent of the FB-DIMM memory system yet makes key changes to its architecture including the use of existing DIMMs to reduce cost, a reduction in power (relative to FB-DIMM), and a more stable request latency. The problem is that the few vendors utilizing this design have the same general approach, yet the implementations vary greatly in their non-trivial details.

A hardware verified simulation suite is developed to accurately model and evaluate the behavior of this buffer-on-board memory system. A study of this design space is performed to determine optimal use of the resources involved. This includes DRAM

and bus organization, queue storage, and mapping schemes. Various constraints based on implementation costs are placed on simulated configurations to confirm that these optimizations apply to viable systems. Finally, full system simulations are performed with MARSSx86 to better understand how this memory system interacts with a CPU, cache, and operating system executing an application. Full system simulations uncover behaviors not present in simple limit-case simulations such as the impact of address and channel mapping schemes or the organization of ports and the associated buffers. When applying insights gleaned from these simulations, optimal performance can be achieved while still considering outside constraints (i.e., pin-out, power, and fabrication costs).

BUFFER–ON–BOARD MEMORY SYSTEM

by

Elliott Cooper-Balis

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2012

Advisory Committee:

Professor Dr. Bruce Jacob, Chair/Advisor

Professor Dr. Manoj Franklin

Professor Dr. Gang Qu

Professor Dr. Alan Sussman

Professor Dr. Donald Yeung

This dissertation is dedicated to,

my family, Judith Cooper, Frank Balis, and Will Cooper-Balis

and

Paul Rosenfeld

For their help and support along the way

Contents

1 Background **1**

1.1 Introduction 1

1.2 Past, Present and Future Memory Systems 5

1.2.1 Double Data Rate Synchronous DRAM 5

1.2.2 Registered & Load Reducing DIMM 12

1.2.3 Fully Buffered DIMM 14

1.2.4 IBM Power 7 17

1.2.5 Intel Scalable Memory Interface 19

1.2.6 AMD G3 Memory Extender 21

2 Buffer-On-Board Memory System **23**

2.1 Architecture Overview 24

2.2 Main BOB Controller 26

2.3 Simple Controller 28

2.4 Packets 31

3 BOB Simulation Suite **33**

3.1 Simulation Framework 33

3.2 Hardware Verification 35

4 Simulation Results **38**

4.1 Limit-Case Simulations 39

4.1.1 Simple Controller & DRAM Efficiency 40

4.1.2 Link Bus Configuration 47
4.1.3 Peak Possible Bandwidth 56
4.1.4 Multi-Channel Optimization 58
4.1.5 Cost Constrained Simulations 72
4.1.6 Ports 76
4.2 Full System Simulations 81
4.2.1 System Performance & Power Trade-offs 84
4.2.2 Latency Analysis 93
4.2.3 Address Mapping 101
4.2.4 Read Return Queue 120
4.2.5 Port Parameters & Heuristics 132

5 Conclusion **148**

5.1 Future Work 152

6 Appendix A **154**

List of Figures

1 DRAM Signal Integrity Degradation 3

2 Trends In Commodity DRAM 4

3 Modern DRAM Device 7

4 JEDEC DDR Memory System 10

5 Ganged vs. Unganged Memory 12

6 LR-DIMM 14

7 FB-DIMM Memory Architecture 16

8 IBM Power7 Memory Architecture 19

9 Intel SMI/SMB Architecture 20

10 AMD G3MX 22

11 BOB Memory Architecture 25

12 Main BOB Controller 27

13 Simple Controller 29

14 BOB Packet Definition 31

15 ModelSIM Verification 36

16 ModelSIM In Action 37

17 Varying Rank Depth 41

18 Varying O–O–O Depth 42

19 RRQ Depth and Response Width - DDR3-1066 44

20 RRQ Depth and Response Width - DDR3-1333 44

21 RRQ Depth and Response Width - DDR3-1600 45

22 Varying Response Link Bus 46
23 Impact of tFAW on Data Burst 47
24 Link Bus Configurations - DDR3-1066 50
25 Link Bus Configurations - DDR3-1333 51
26 Link Bus Configurations - DDR3-1600 51
27 Impact of Read–Write Mix on Performance 54
28 Latency Components in Different BOB Configurations 55
29 Varying Number of BOB Channels 57
30 Multi-Channel Arbitration 59
31 4-to-1 Multi-Channel Configuration 61
32 Multi–Channel Bandwidth with DDR3-1066 62
33 Multi–Channel Bandwidth with DDR3-1333 63
34 Multi–Channel Bandwidth with DDR3-1600 64
35 Read Return Queue In Multi-Channel Configurations - DDR3-1066 . 69
36 Read Return Queue In Multi-Channel Configurations - DDR3-1333 . 70
37 Read Return Queue In Multi-Channel Configurations - DDR3-1600 . 71
38 Pareto Frontier Analysis 74
39 Varying Port Width and Depth 78
40 Utilization of Port Resources 80
41 Pareto Frontier Analysis - *mcol* 82
42 Pareto Frontier Analysis - *mg* 83
43 Pareto Frontier Analysis - *sp* 83
44 Pareto Frontier Analysis - *STREAM* 84

45 Full System Simulation - *facesim* 88
46 Full System Simulation - *fluidanimate* 89
47 Full System Simulation - *mcol* 89
48 Full System Simulation - *mg* 90
49 Full System Simulation - *Sandia GUPS* 90
50 Full System Simulation - *sp* 91
51 Full System Simulation - *STREAM* 91
52 Latency Components - *STREAM* 94
53 Latency Components - *mcol* 95
54 Per Channel Latency Components - *facesim* 99
55 Per Channel Latency Components - *fluidanimate* 99
56 Per Channel Latency Components - *mcol* 99
57 Per Channel Latency Components - *mg* 100
58 Per Channel Latency Components - *Sandia GUPS* 100
59 Per Channel Latency Components - *sp* 100
60 Per Channel Latency Components - *STREAM* 101
61 Impact of Channel Mapping - *facesim* 106
62 Impact of Channel Mapping - *fluidanimate* 107
63 Impact of Channel Mapping - *mcol* 108
64 Impact of Channel Mapping - *mg* 109
65 Impact of Channel Mapping - *STREAM* 110
66 Address Mapping - *fluidanimate* 115
67 Address Mapping - *Sandia GUPS* 116

68 Address Mapping - *mcol* 117

69 Address Mapping - *sp* 118

70 Address Mapping - *STREAM* 119

71 Impact of Return Queue Depth - *fluidanimate* 123

72 Impact of Return Queue Depth - *mcol* 124

73 Impact of Return Queue Depth - *mg* 125

74 Impact of Return Queue Depth - *sp* 126

75 Impact of Return Queue Depth - *STREAM* 127

76 Impact of Return Queue on Latency 129

77 Impact of Port Configuration - *STREAM* (A) 136

78 Impact of Port Configuration - *STREAM* (D) 137

79 Impact of Port Configuration - *STREAM* (G) 138

80 Impact of Port Configuration - *STREAM* (I) 139

List of Tables

1 Response Packet Transmission Times 49

2 Multi-Channel Link Bus Utilization - DDR3-1066 62

3 Multi-Channel Link Bus Utilization - DDR3-1333 63

4 Multi-Channel Link Bus Utilization - DDR3-1600 64

5 Configuration Parameters for Cost-Constrained Systems 74

6 MARSSx86 Configuration 81

7 Normalized Power Consumption of Benchmarks 92

8 Bit Field Naming Conventions 103

9 Channel Mapping Schemes 103

10 Channel Mapping - Bandwidth & Execution Time - *facesim* 106

11 Channel Mapping - Bandwidth & Execution Time - *fluidanimate* . . 107

12 Channel Mapping - Bandwidth & Execution Time - *mcol* 108

13 Channel Mapping - Bandwidth & Execution Time - *mg* 109

14 Channel Mapping - Bandwidth & Execution Time - *stream* 110

15 DRAM Mapping Schemes 112

16 DRAM Mapping - Bandwidth & Execution Time - *fluidanimate* . . . 115

17 DRAM Mapping - Bandwidth & Execution Time - *Sandia GUPS* . . 116

18 DRAM Mapping - Bandwidth & Execution Time - *mcol* 117

19 DRAM Mapping - Bandwidth & Execution Time - *sp* 118

20 DRAM Mapping - Bandwidth & Execution Time - *STREAM* 119

21 Impact of Return Queue Capacity on Bandwidth 131

22 Port Configuration Results - A 136

23 Port Configuration Results - D 137

24 Port Configuration Results - G 138

25 Port Configuration Results - I 139

26 Port Buffer Utilization - A 143

27 Port Buffer Utilization - D 144

28 Port Buffer Utilization - G 145

29 Port Buffer Utilization - I 146

1 Background

1.1 Introduction

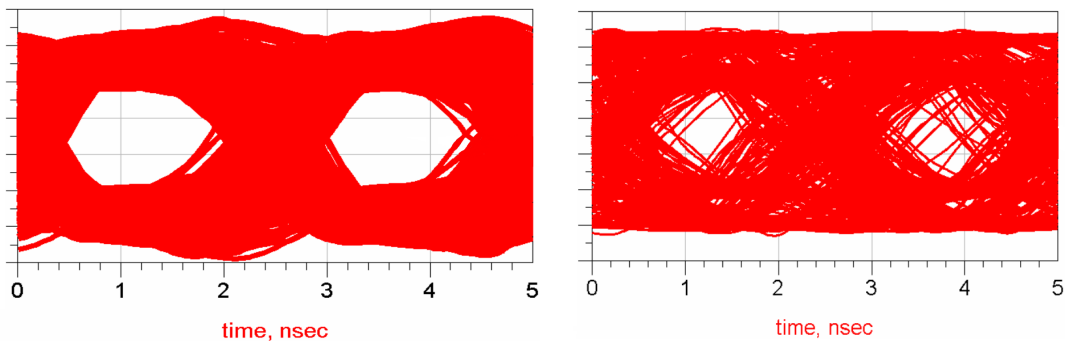
The modern memory system has remained essentially the same for almost 15 years. Decisions made in the past, when the disparity between the CPU and memory clock were not considered to be an issue, are now preventing the memory system from providing the capacity and bandwidth that today's systems and applications demand. Unless modifications are made, the memory system will become an even greater bottleneck than it is now, further impeding performance gains in modern systems.

Unfortunately, alterations to the memory system are met with significant resistance for numerous reasons. First, the fact that the profit margins on memory devices are relatively small (compared to that of a CPU) prevents manufacturers from accepting even modest changes due to the risk of possible failure. Second, any change that is not transparent to the rest of the system also requires support from manufacturers of other system parts. For example, if a new memory module needs to change size or pin-out to support a new feature, this would require cooperation from motherboard manufacturers, CPU and chipset manufacturers, and possibly even operating system and application developers. Lastly, the average consumer is unaware of the importance of the memory system and does not see the benefit when faced with increased costs. In the end, the consumer will typically purchase whatever is cheapest, regardless of the benefit seen by using better products.

To support some level of memory system customization and expandability, commodity DRAM memory is purchased on a PCB, called a dual in-line memory module

(DIMM), which uses physical contact (i.e., pins that plug into a motherboard slot) to provide electrical connectivity with the rest of the system. This physical contact is sufficient for electrical signals that operate at low speeds (<100MHz), but as the memory clock has increased to maintain pace with the CPU, this solution is proving to be less than ideal. The signal integrity at these physical contacts is greatly degraded as the memory clock is increased. This issue is exacerbated as more DIMMs are placed in a channel and the further a particular DIMM is located from the memory controller's signal driver [24, 21].

This can be seen in **Figure 1** where the signal integrity of the DRAM bus is displayed for a system that has two (**Figure 1(a)**) and four (**Figure 1(b)**) DDR2-400 DIMMs. While only two DIMMs are attached to the bus, the data eye is clearly defined. A well defined data eye means that there is a clear differentiation between high and low voltages, and that the signal rise and fall time is short. This makes it easier to interpret what value these signals represent. The data eye is drastically reduced once the DRAM bus has four DIMMs attached, making it more difficult to determine what value is being sent on the bus and increasing the likelihood of errors. The degradation of signal integrity is a result of several factors. The additional DIMMs cause an increase in load seen by the signal drivers, an increase in signal cross-talk, and an increase in signal reflection [40].

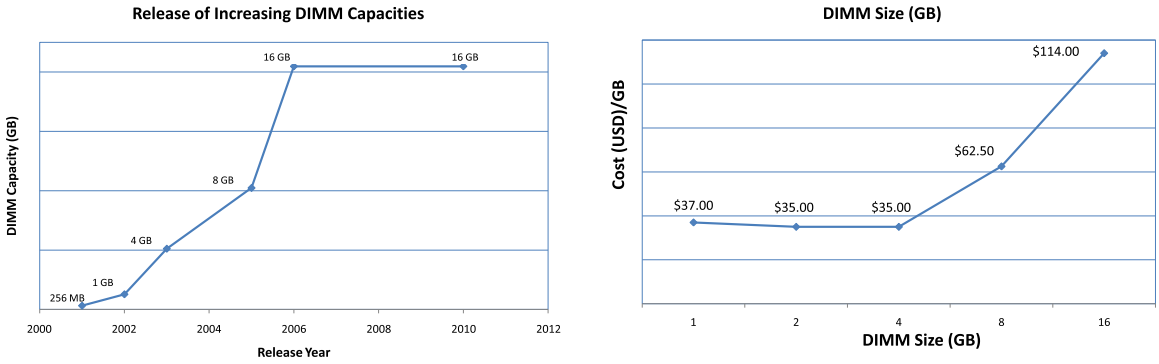


(a) Signal integrity seen by two DDR2 DRAM DIMMs

(b) Signal integrity seen by four DDR2 DRAM DIMMs

Figure 1: Signal integrity degradation seen on DRAM bus when increasing the number of DIMMs (Original images from [40])

As a result of these issues, when manufacturers increase the memory clock, they must reduce the number of DIMMs allowed in a channel to avoid extraneous costs of mitigating these signal integrity issues. This severely limits the total capacity supported in a system. For example, the original DDR SDRAM standard allowed four DIMMs in a channel, while DDR2 allowed two, and the higher-speed DDR3 variants (i.e., DDR3-1600) only allows a single DIMM of depth [21, 22]. While it is possible to place higher capacity DIMMs in the channel, the overall rate of increase in capacity of a DIMM has slowed due to the difficulties in decreasing the size of a DRAM cell's capacitors. **Figure 2(a)** shows indirect evidence of this effect as the size of available DIMMs has remained at a 16 GB ceiling for the last five years. Unfortunately, the cost of these high-capacity DIMMs does not scale proportionally with their capacity, as can be seen in **Figure 2(b)**.



(a) Release dates of various sized DIMMs

(b) Cost per GB of various sized DIMMs

Figure 2: Trends In Commodity DRAM

The FB-DIMM memory system was originally introduced to solve the issues outlined above. Each FB-DIMM uses standard DDRx DRAM devices and has additional logic called the advanced memory buffer (AMB). The AMB allows each memory channel to operate on a fast and narrow bus by interpreting a new packetized protocol and issuing DRAM specific commands to the DRAM devices. Unfortunately, the high speed I/O on each AMB resulted in unacceptable levels of heat and power dissipation [24, 7]. The inclusion of the AMB also resulted in more expensive DIMMs relative to similar capacity DDRx DIMMs. These issues ultimately led to the failure of the standard.

To fill the void left by FB-DIMM, vendors such as Intel, AMD, and IBM have devised new architectures to try and resolve the memory capacity and bandwidth issues. Although similar, these new architectures make key changes that prevent the issues that plagued an FB-DIMM system: while each memory channel still operates on a fast, narrow bus, it contains a single logic chip per channel as opposed to one logic chip per module. This allows the new architecture to use of existing low-cost DIMMs,

prevents excessive power and heat in the logic, and reduces variance in latency.

While this buffer-on-board memory system has already been implemented in a small number of high-end servers, the problem exists that each of these implementations differs in non-trivial details. The contribution of the present work is an examination of this new design space to determine optimal use of the resources involved and the exploration of performance enhancing strategies. This includes proper bus configurations for various types of DRAM, necessary queue depths to reach peak DRAM efficiency, and address and channel mappings to ensure an even request spread in order to reduce resource conflicts.

1.2 Past, Present and Future Memory Systems

The past five years have seen numerous efforts to devise a next-generation memory system. While many ideas have been proposed, no clear solution has been widely adopted.

1.2.1 Double Data Rate Synchronous DRAM

The most ubiquitous form of memory in use today is the JEDEC standardized double data-rate (DDR) synchronous DRAM. For the past 15 years, this has been the dominant form of commodity memory, eventually breaking into mobile and supercomputing markets due to an overwhelming abundance of parts. The widespread success of DRAM is attributed to the standardization of the device packaging, pin-out, and operating protocol.

A modern DDR SDRAM device (**Figure 3**) contains numerous arrays of capac-

itive cells used to store bits of data. Each of these arrays is typically 256Kb each [16, 26]. These arrays are organized into larger *banks* which operate in parallel and independent of one another; a DDR3 SDRAM device has eight banks [11]. Banks are composed of smaller arrays to prevent extraneous control lines which would span the entire length a row or column resulting in an unnecessary load on the control circuitry.

The data stored in each bank is accessed using separate row and column addresses which are sent to the device using separate commands on different cycles. These commands are sent via the device's command bus in conjunction with a row or column address which are sent via the address bus. The row address is sent to the device first with the *ACTIVATE* (ACT) command. Upon receiving the ACT command, the entire row of the bank is sent to the sense amplifier. Each bank has its own sense amplifier which is responsible for interpreting the minuscule charge that is stored in a cell's capacitor.

Once this operation has been completed, data may be read out of or into the sense amplifier with a column access command (*READ* or *WRITE*). The amount of time between an ACT and READ or WRITE is dictated by the tRCD (**R**ow to **C**olumn **D**elay) timing constraint.

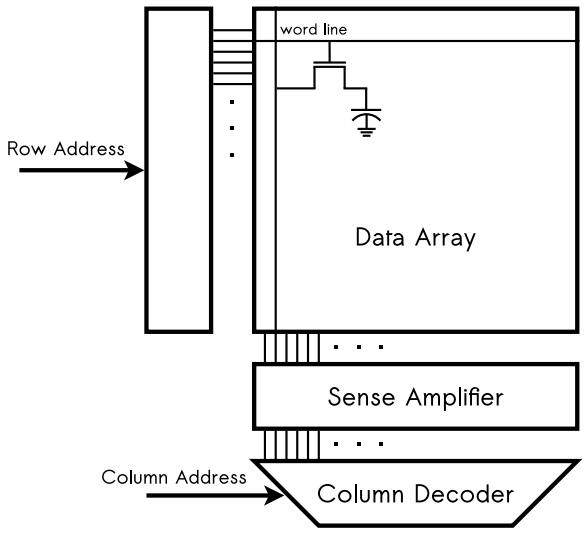


Figure 3: The modern DDR SDRAM Device architecture

Once the data has finished being read from or written to the sense amplifier, the *PRECHARGE* (PRE) command is responsible for resetting the sense amplifier and bit lines to prepare for another row access. The amount of time it takes to perform this action is dictated by the tRP (**R**ow **P**recharge) timing constraint. The point at which this command is issued is dictated by the *row buffer management* policy. If the PRE command is issued immediately after a column access, then the operation is referred to as a *closed page* policy. Conversely, if the precharge command is not issued immediately, the sense amplifier retains the row data, which can be advantageous in the case that subsequent requests access this same data. This will mitigate the power and time costs of precharging and re-activating the same row; such a policy is referred to as *open page* and is used in situations of high address space locality.

Even once the row has been precharged, there are other important timing constraints which must be adhered to before a new row may be activated. For instance the tRC (**R**ow **C**ycle) constraint dictates the amount of time between subsequent

ACT commands to the same bank. The tFAW (**F**our **A**ctivation **W**indow) is a timing constraint used to restrict the current draw from a particular device by dictating the amount of time in which a maximum of four ACT commands may be issued.

Due to the nature of the capacitors used to store individual bits of data, the representative charge leaks over time, and the intended value dissipates beyond recognition. The *REFRESH* (REF) command resolves this issue by reading a row and placing it back into the data array, thereby refreshing the charge in each cell's capacitor. This is done once every 64 ms. Thus, in a device with 8192 rows, a REF command is typically issued every 7.8 microseconds [28]. The amount of time a refresh operation takes is dictated by the tRFC (**R**e**F**resh **C**ommand) timing constraint. This specifies the length of time between a REF and another REF or a subsequent ACT command.

Power consumption within a DRAM device comes from several distinct sources:

- **Background Power -** Regardless of the operation currently taking place, there is always a constant dissipation of background energy which is used to operate the control circuitry. This value can vary depending on whether or not a row is currently activated and being held within the sense amplifiers or whether the device is in low power mode.

- **Activation and Precharge Power -** The power to activate and precharge a row within the data array is one of the main sources of power consumption in a DRAM device. As previously stated, when a row is activated, the entire row of the data array is sent to the sense amplifier regardless of how much data is actually needed. At some point subsequent to the activation, the sense amplifier

must be precharged to prepare for a new request. Under heavy utilization, these operations can account for the majority of the power consumed by the DRAM device [18].

- **Burst Power -** This component of a device's power consumptions accounts for each time data is transmitted on the data bus from a read or write request. The relative size of this component varies with memory system utilization.

- **Refresh Power -** This is the power consumption from refreshing the rows of the DRAM data array. This is normally a constant power draw as the operation is performed at regular periodic intervals.

The ratio of these various sources of power consumption is dependent on how the memory system is being used. An idle memory system's power consumption will be dominated by background power and refresh power. Conversely, in a highly utilized memory system, the activation and precharge power will dominate. The row buffer management policy will also have an impact on the ratio of these sources of power consumption. For example, with an *open page* management policy, it is possible to mitigate some of the activate and precharge power by leaving a row open, therefore causing the read and write burst power to be greater than the activate and precharge power. When all of these sources are accounted for during "high usage" situations of 80% bus utilization, a single DDR SDRAM device consumes approximately 689.5 milliwatts [1], a DDR2 DRAM device consumes approximately 340.1 milliwatts [2], and a DDR3 DRAM device consumes approximately 435.9 milliwatts [4]. These values will vary depending on device capacity, clock rate, and utilization.

The JEDEC standardized DDR SDRAM device described above is used within the standardized memory system seen in **Figure 4**. This memory system contains one or more channels composed of a command and address bus and a 64-bit wide data bus. Each of these channels may have one or more *ranks* of DRAM. A rank of memory is a group of DRAM devices connected in parallel that operate in lockstep by receiving and handling the same requests, at the same time. Each of the device's data buses (which range from 4 to 16 bits) are aggregated together to form the 64-bit data bus. Since DRAM devices do not have a set data bus width, the number of DRAM devices which are used to form a rank can vary. The memory controller, now typically located on the CPU die, is responsible for issuing the DRAM-specific commands detailed above. When reading or writing data out of the device, data is driven on both rising and falling edges of the memory clock, which is where the double data rate nomenclature arises.

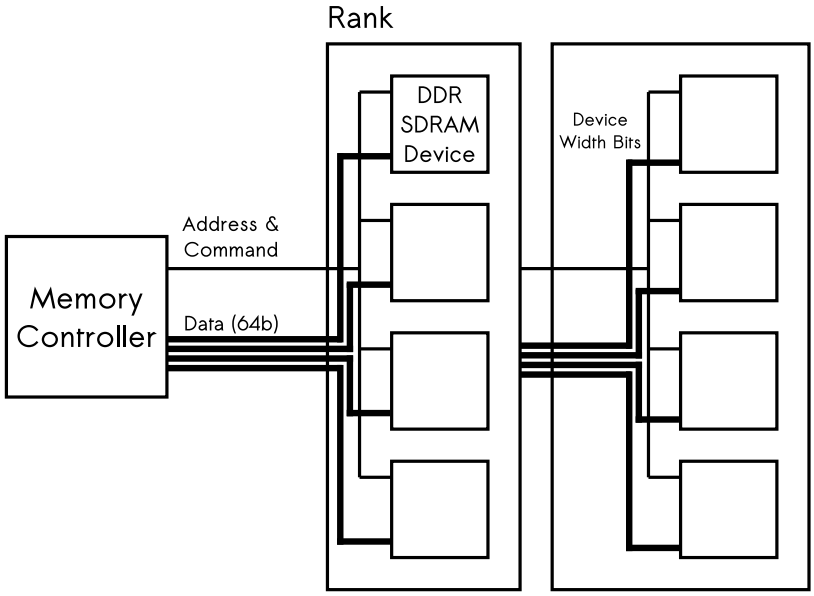
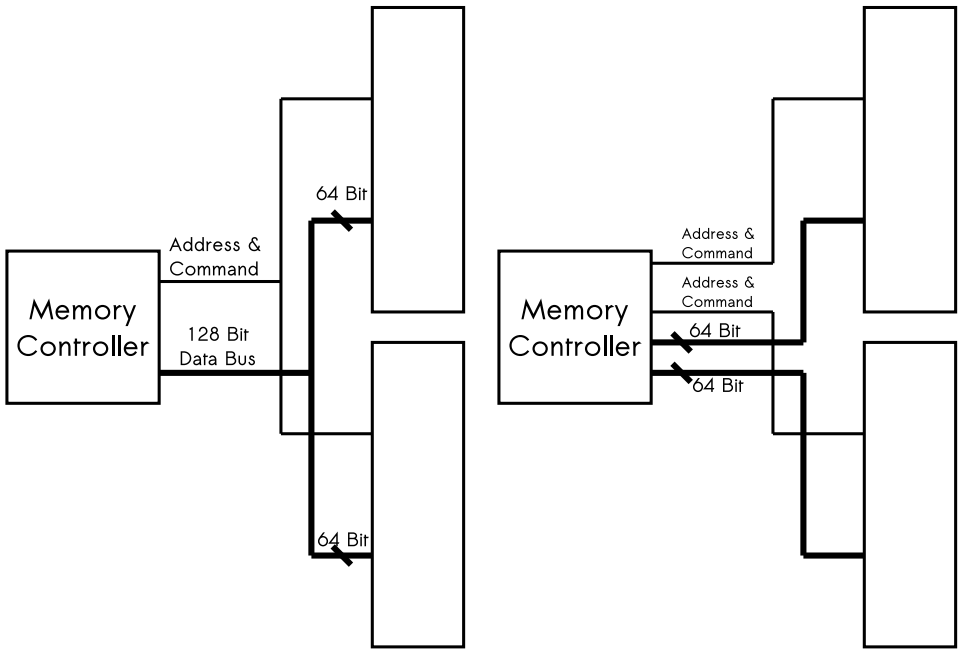


Figure 4: JEDEC standardized memory system architecture

As previously mentioned, increasing the clock rate with each successive generation of the DDR standard has limited the number of DIMMs allowed in each memory channel. To circumvent this limitation, most major manufacturers have modified the architecture and widened the memory bus to increase capacity as opposed to relying on additional rank depth. These architectures, called dual- or triple-channel, have two and three DRAM buses, respectively, which can be *ganged* or *unganged*.

In a *ganged* architecture, each 64-bit rank receives the same request at the same time as if the DRAM data bus is actually double or triple the size of the rank's data bus. In an *unganged* configuration, each 64-bit DRAM channel acts independently of the others. While an unganged architecture will allow for greater concurrency and fewer resource conflicts, it requires duplicate logic within the controller to operate each channel independently. Both of these modified architectures do permit higher capacity at the same time as increasing the clock rate, yet the required pin-out (one of the most significant costs to chip manufacturers) on the processor is incredibly high. A modern Intel i7 processor uses over 350 pins for a triple-channel memory system [14]. The current generation (DDR3) can support transfer rates of up to a theoretical 12.8GB/s while operating at 1600Mbit/s [11]. The next generation of DDR SDRAM (DDR4) is expected to be widely available by 2012 [33] yet still faces the same limitations as previous generations and is simply a temporary solution, in that it will still use a wide, multi-drop bus that will still have the speed and capacity issues explained above.



(a) Ganged ranks (b) Unganged ranks

Figure 5: The differences between ganged and unganged ranks or DRAM

1.2.2 Registered & Load Reducing DIMM

Another modification to the DDR SDRAM architecture is the introduction of the Registered DIMM (RDIMM) and Load Reducing-DIMM (LR-DIMM) [27] standards. The RDIMM standard was introduced first and alleviated some of the signal integrity issues of high-speed memory modules. These DIMMs use standard DDR3 DRAM devices but place a register between the devices and the main memory controller. This register is responsible for latching all control signals, such as the bank address lines, address bits which address the columns and rows, CAS and RAS signals, and power-down control lines [29]. These modules came at a higher cost; this is due to a lower manufacturing volume relative to unbuffered DIMMs and not due to higher part costs, which comprise a simple buffer. A major benefit to this modification is that it

maintains both the interface and protocol of the existing standard, thereby allowing consumers and system manufacturers to retain old hardware while still having the benefit of the increased performance provided by these new DIMMs.

Similar to the RDIMM standard which buffers the control and address signals, an LR-DIMM places a *memory buffer* (MB) [30] on all of the signals between the CPU and DRAM (**Figure 6**). This includes the entire data bus and data strobe lines, along with the control and address lines. This provides an even greater boost in signal integrity and reduction in load on the main controller relative to an RDIMM because there is now only a single load per DIMM as opposed to a load equal to the number of DRAM devices. Again, this allows a faster memory clock with an even greater number of DIMMs possible in a single channel. This is aided even further by the inclusion of a phase-lock loop circuit which re-drives the clock signal, allowing it to be run at a higher frequency without fear of noise or signal degradation.

As with RDIMMs, these modifications have no impact on the existing interface or protocol, making the LR-DIMM attractive to both the memory manufacturers who are wary of costly modifications and to the consumers and vendors who are not required to make modifications to existing systems. The only noticeable difference seen to the system is a one cycle increase in latency to account for the latching of signals and data before it is sent to the DRAM device or memory controller. While this modification will extend the life of the DDR SDRAM architecture, it is still a temporary fix to the underlying issues facing modern memory systems. LR-DIMMs with a capacity of 16GB and 32GB are expected to be released to market at some point in 2011.

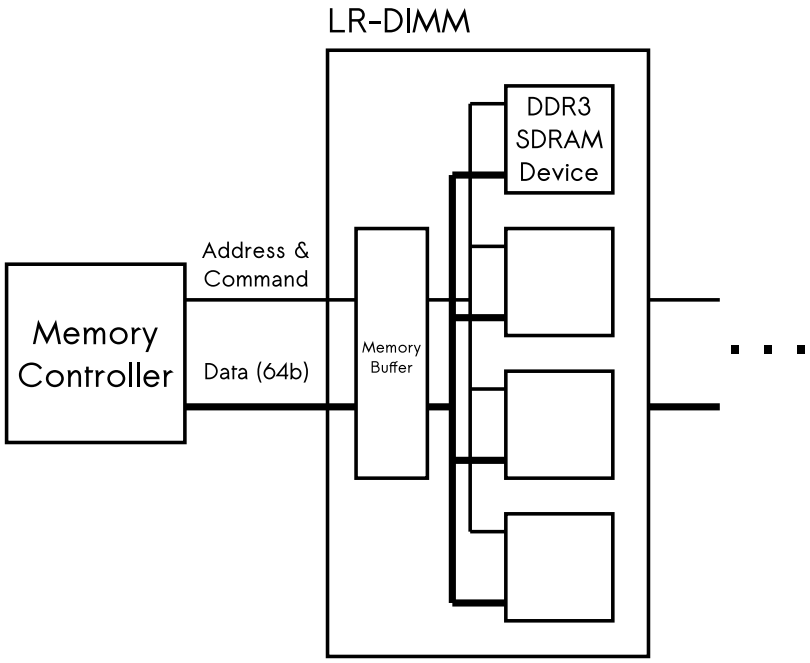


Figure 6: The LR-DIMM architecture

1.2.3 Fully Buffered DIMM

In 2004, an entirely new memory architecture standard was adopted that was intended to alleviate the problems with the current design. The new design, called fully buffered DIMM (FB–DIMM), uses the same DRAM devices as DDR2 and DDR3 SDRAM DIMMs, but operates on a faster and more narrow bus relative to the standard DDR architecture [6]. This was made possible by the inclusion of a small controller on each DIMM called the *advanced memory buffer* (AMB) [10]. The AMB is responsible for interpreting the packetized protocol and controlling the DRAM devices located on each DIMM. The standard defines a point-to-point interface between DIMMs, which causes the memory system to appear as a multi-hop store & forward network [22]. This architecture allows a much higher capacity (up to 768 GB per system) and significantly higher bandwidth per pin due to its increased clock.

An FB–DIMM memory channel operates on two separate logical buses: the northbound channel and the southbound channel [24, 22]. These channels are different widths to account for the disparity between reads and writes during typical operation; the northbound channel (going toward the CPU) is 14 data lanes and the southbound channel (going away from the CPU) is 10 data lanes [9] (**Figure 7**). The resulting peak bandwidth achievable by each bus is dependent on the DRAM devices used on the module. A multiple of the DRAM reference clock is used as the frequency for the northbound and southbound bus and results in the ability to transfer twice as many reads as writes [23]. For example, when an FB-DIMM uses DDR2-667 devices, the channel can support peak bandwidth of 8 GB/s.

To account for such narrow buses, requests and responses are encapsulated in packets or *frames*. As these frames are transmitted on their respective channel, the AMB interprets the contents to determine proper routing or to generate standard DRAM commands for local DRAM devices. To accommodate proper packet transmission, each of these channels operates at speed exactly six times that of the DRAM devices which populate the DIMM (i.e., DDR3-1333 with a 667MHz clock rate operates on FB–DIMM channels of 4GHz).

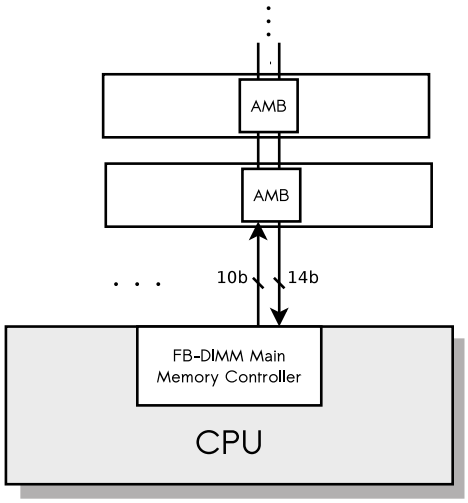


Figure 7: The FB-DIMM memory system

A southbound frame consists of up to three independent commands or a single command and 72 bits of write data (64 data bits and 8 ECC bits). In a frame made up of three separate commands, each command is destined for a different DIMM within the channel. These commands can either be a typical read or write requests (which are then interpreted by the AMB into standard DRAM commands like ACT, READ, WRITE, or REFRESH) or channel commands which are used to manage operating conditions within the FB–DIMM channel [24]. These commands include debug operations, channel syncing operations, reset commands, and control register setting commands. Scheduling at the DRAM level is still handled by the main memory controller with the AMB simply interpreting the frames it receives into DRAM based commands.

A northbound frame consists of 144 data bits retrieved from the DRAM as a result of a previous request (128 data bits and 16 ECC bits). The 128 data bits corresponds to the data retrieved from a single DRAM device cycle where 64 bits are transmitted

on each clock edge.

Unfortunately, an unforeseen consequence of the FB–DIMM architecture's point-to-point nature and the use of high speed I/O in each AMB caused unacceptable levels of heat and power dissipation. Under heavy load, a fully populated FB–DIMM channel of 8 DIMMs (totally 32 GB) requires over 90 watts while under moderate load [8, 3], which is on par with CPUs at the time. Tests have shown that an FB-DIMM system consumes over 800% more power than a comparable DDR2 memory system [17]. Because of this issue, adoption of the standard slowed, and FB-DIMM was eventually removed from all major technology roadmaps. While no clear successor to FB–DIMM has been proposed, major vendors have taken it upon themselves to find solutions to the capacity and bandwidth issues. They have done this by designing new architectures which, like FB–DIMM, use existing DRAM devices attached to intermediate logic, operating on a relatively narrow, high-speed bus. These new architectures are simply organized in a different fashion.

1.2.4 IBM Power 7

IBM's new 8-core Power7 processors (**Figure 8**) have implemented a novel memory system which increases DRAM capacity to up to 256 GB per CPU socket (and up to 8 TB in a system using a Power795 processor) with an access rate of 1066MHz [12]. All Power7 CPUs have two on-die memory controllers each with 8 KBytes of scheduling window [25]. The memory controller communicates with 4 logically independent channels, for a total of 8 memory channels per CPU socket. Unlike a standard DDR3 memory system, a channel is now two logically separate, uni-directional buses which

are faster and more narrow than the standard DRAM bus. Like FB-DIMM, this is possible through the utilization of an *advanced buffer chip* which is placed between the CPU and DIMMs.

Each of the advanced buffer chips communicates with the on-die memory controller via a 6.4 GHz channel which has 8 data lanes towards the DIMM and 16 data lanes towards the CPU. With 8 of these channels, a CPU has a total of 136.44 GB/s of available memory bandwidth [12]. Unlike the FB-DIMM standard where an AMB is responsible for communicating with other AMBs in a channel, the advanced buffer chip used in the Power7 memory architecture is only responsible for communicating with a single DIMM. This alleviates some of the issues with the FB-DIMM design, such as excessive power dissipation within the AMB due to constantly communicating with other DIMMs in the channel and variable latencies caused by a multi-hop store & forward network. The design of the advanced buffer chip is propriety to IBM [25] so the protocol used to communicate with the CPU is unavailable. Due to the widths and operating frequency of each channel, it would be impossible to use the FB-DIMM protocol.

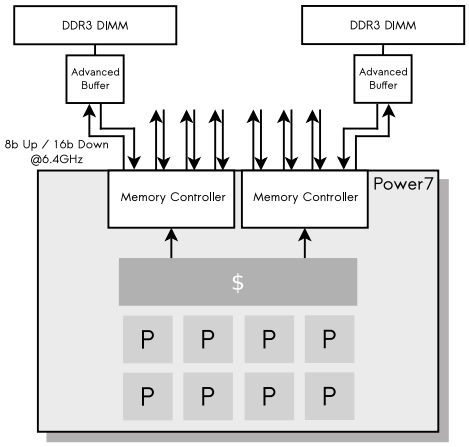


Figure 8: IBM's Power7 memory architecture

1.2.5 Intel Scalable Memory Interface

After it was clear that FB-DIMM had failed, Intel decided to modify the architecture slightly to alleviate the issues with the design instead of abandoning it completely. At one point called FB-DIMM2 [15], Intel's Scalable Memory Interface (SMI) is a memory system architecture for Nehalem EX processors and has recently been implemented into IBM's System X and BladeCenter systems [37]. The block diagram for this system can be seen in **Figure 9**. Similar to both the Power7 memory system and FB-DIMM, this design is made possible by a logic chip attached to each channel. The logic chip used in this system is called the Scalable Memory Buffer (SMB) and is placed between standard DDR3 RDIMMs and the CPU that it communicates with over the SMI buses.

The SMI interface between the Nehalem-EX processor and SMB consists of two uni-directional buses with 9 southbound data lanes (+1 for CRC) for requests and 12 northbound data lanes (+1 for CRC) for responses. The rate at which these buses are operated is dependent on the CPU currently in the socket and includes 4.8 Gb/s,

5.86 Gb/s, or 6.4 Gb/s [13]. The SMB is also responsible for operating two logically independent, JEDEC standardized channels of DDR3 RDIMMs where each channel is allowed up to two DDR3 RDIMMs. Each Nehalem-EX processor has four SMIs, thereby providing a total capacity of up to 256 GB per CPU socket and a total memory channel bandwidth of 67.2 GB/s when clocked at 6.4 Gb/s.

As with IBM's Power7 systems, Intel's architecture is differentiated from the FB-DIMM standard by using the SMB to only communicate with the DIMMs as opposed to other logic. This alleviates many of the issues with the FB-DIMM design such as unacceptable heat and power dissipation and variance in latency. Again, similar to the Power7 memory architecture, the SMI and SMB are both propriety, and therefore the communication which occurs over each SMI bus to the SMB is unknown. A protocol similar to FB-DIMM is likely since Intel developed the original standard, yet they specifically state that the original FB-DIMM protocol is not supported within an SMI/SMB system [13].

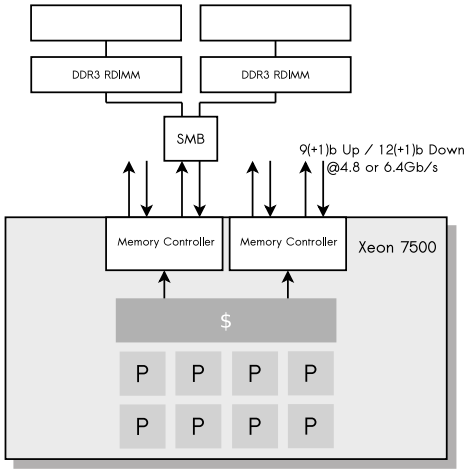


Figure 9: Intel's Xeon 7500 memory architecture

1.2.6 AMD G3 Memory Extender

AMD had also proposed a similar solution to the issues facing current memory system design. Just like IBM and Intel, they proposed placing a piece of logic called the Socket G3 Memory Extender (G3MX) between an Opteron CPU and either DDR3 UDIMMs or RDIMMs. Each Opteron CPU would have an on-die memory controller communicating with four G3MX devices over separate channels. Each of these channels are made up of two logically separate and uni-directional buses with a request bus of 13 data lanes and a response bus of 20 data lanes (**Figure 10**). This leads to a memory channel which only uses 66 pins (when differentially signaled), far less than the previous Opteron DDR memory channel. Therefore, with the same number of pins as a DDR memory channel, the system's memory capacity could be more than doubled.

The G3MX would be responsible for controlling up to 4 standard DDR3 U/RDIMMs giving each processor a total of up to 16 DIMMs per socket (as opposed to the 8 DIMM limitation of previous Opteron based systems). The architecture is similar to IBM and Intel's architectures detailed above; it places logic between the CPU and DIMMs, and communicates over fast and narrow buses of unequal widths. The only discrepancies between all three of these designs are the widths of the respective request and response buses and the amount of DRAM allowed on the far side (relative to the CPU) of the logic. AMD officially canceled the G3MX memory system in 2008, and they have yet to announce a replacement [35].

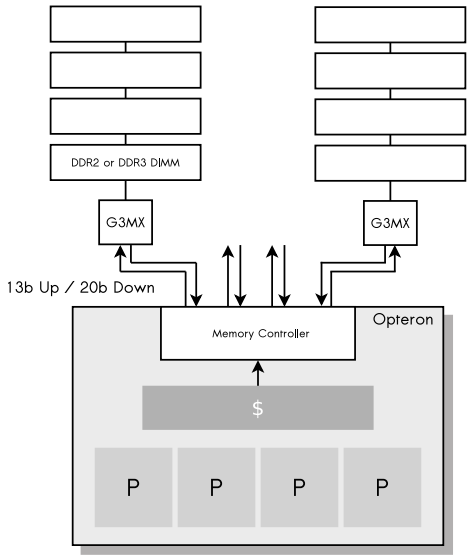


Figure 10: AMD's G3MX memory architecture

2 Buffer-On-Board Memory System

It is clear that with Intel's SMB/SMI systems, AMD's G3MX, IBM's Power7, and even JEDEC's LR-DIMM, near-term memory system design is heading in a similar direction: numerous concurrent channels of DRAM which have logic (either a controller or buffer) located between the DIMMs and the CPU, with communication provided over a narrow and fast bus. The architecture solves all the issues facing commodity memory system design today - it increases capacity, increases aggregate bandwidth, increases speed, and allows for far greater concurrency without increasing pin count. These benefits are a result of the introduced logic which improves signal integrity when faced with physical contact based electrical connections and reduces the required CPU pin-out to the memory system, allowing operation at a higher frequency.

This architecture also alleviates many of the issues that plagued the FB-DIMM memory system. The excessive heat and power dissipation that resulted from communication between chained AMBs is no longer an issue as this new design only utilizes the intermediate logic to communicate with the CPU and DIMMs. There is no longer an issue with large variance in request latency which resulted from chained AMBs and FB-DIMM channels containing a large number of DIMMs. Lastly, the high costs involved with FB-DIMMs can be circumvented as the logic can now placed on the motherboard, allowing standard DIMMs to be used instead.

While a number of influential vendors are currently working on the concept, few systems have actually been implemented, and those which have, all vary in their

specifics, such as bus speed and width or rank depth. Therefore, while the architecture is still young, an examination of the design space is necessary.

2.1 Architecture Overview

The generalized form of this *buffer-on-board* (BOB) memory system architecture can be seen in **Figure 11**. It consists of DRAM channels populated with commodity DIMMs which are composed of standard DDR devices. These DIMMs can be unbuffered, registered, or even load-reducing DIMMs. Each of these *BOB channels* could be considered identical to a regular, JEDEC-standardized memory system. The control and data bus, operating protocol, and timing constraints are the same ones used in a normal memory system. The *simple controller* in each BOB channel operates as the intermediate logic located between the DIMMs and the main, on-die memory controller.

Each simple controller is responsible for controlling the DRAM, as well as receiving requests and returning data back to the main memory controller (as opposed to the DRAM talking directly to the main memory controller). Communication between the simple controller and the CPU occurs over a *link bus* which is narrower and faster than the DRAM bus which communicates with the DIMMs. Unlike the DRAM bus, which has separate control and data signals, the lanes which comprise a link bus are for general purpose communication. The link bus is full–duplex where the request (towards the DRAM) and response (towards the CPU) data lanes may be different widths and operate at some speed faster than the DRAM.

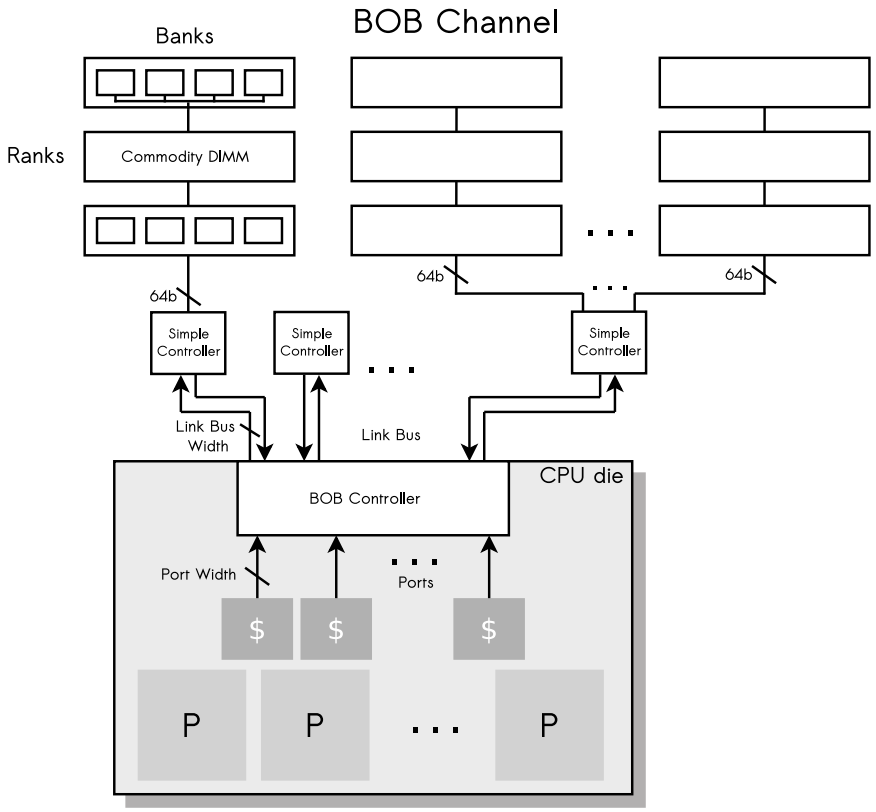


Figure 11: The BOB memory system architecture

The BOB architecture has an interesting characteristic of having three separate clock domains - the CPU clock (which also corresponds to the main BOB controller clock), the link bus clock between the BOB controller and simple controller, and the DRAM clock between the simple controller and the DIMMs it controls. The DRAM clock is defined by the type of DIMM which populates a channel (i.e., if DDR3-1066 DIMMs are used, then the DRAM clock is 533MHz). The link bus clock needs to be faster than the DRAM clock to account for the narrower bus. Obviously, the CPU clock is determined by the CPU. The ratio of each of these clock rates is an important factor in the behavior of each portion of the BOB memory system. Such an architecture provides a chance to optimize in multiple dimensions. Depending

on a system's purpose, the optimal organization of a BOB memory system might vary. Only through an accurate and detailed model and precise simulation can these optimizations be determined.

2.2 Main BOB Controller

The main *BOB controller* (**Figure 12**) that resides on the CPU die is an essential aspect of the architecture. The BOB controller is responsible for the typical functions of a commodity memory controller, such as address mapping and returning data to the cache. The address mapping that takes place within the BOB controller uses the address of a request to determine which BOB channel should receive said request. Like the address mapping in a commodity system, a particular portion of the bits which make up the address are used to determine this mapping.

Communication with the cache and CPU is executed over the main BOB controller's *ports*, which are logically separate, full-duplex lanes. Each port has a corresponding input and output buffer which store requests and responses while awaiting arbitration. The width of each port is on the order of magnitude similar to that of the data bus used to operate the cache. The speed of each port is dictated by the frequency of the CPU. A cross-bar switch is used to route requests and responses to and from port buffers to ensure that a request from any port is capable of being sent to any link bus. The width of this cross-bar switch is the same as each port to ensure an unimpeded flow of requests and data.

Unlike a commodity memory controller, the BOB controller is also responsible for

packetizing requests and interpreting response packets sent to and from the simple controllers over the narrow link bus. Since the link bus is narrower than the DRAM bus, requests and responses must be encapsulated within a packet. These packets are then sent over the link bus during multiple clock cycles. This is accomplished with a serialize-deserialize (*SerDes*) interface and associated buffer for the request and response path of each link bus. These buffers are written into by either the cross-bar switch when issuing requests or the response link bus when returning a response packet. Items are removed from the SerDes buffers when the destination port is free and there is room in that port's buffer, while ensuring that a request is returned to the same port which it was received. When removing items from a SerDes buffer to send via its respective port, a round-robin scheme is used to ensure starvation does not occur.

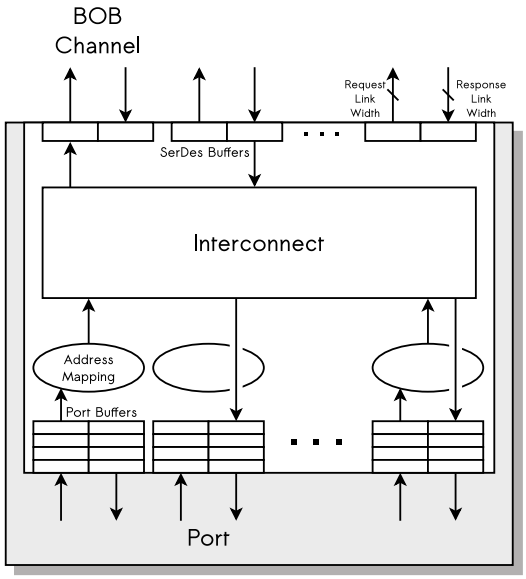


Figure 12: The main BOB controller's block diagram

2.3 Simple Controller

The added logic placed between the main BOB controller and the JEDEC compliant DRAM channels is referred to as the *simple controller* (**Figure 13**). The simple controller has two main functions: controlling the DIMMs using the standard DRAM interface and sending and receiving request and response packets back to the main BOB controller over the request and response link buses, respectively. Upon receiving a request packet, the simple controller must translate the packet into a series of DRAM specific commands such as ACTIVATE, READ, or WRITE. This process also involves address mapping similar to that which takes place in a commodity memory system by a normal memory controller. Particular portions of the address in a request are used to determine the rank, bank, row, and column that will service this request. The address of each resource is then paired with the appropriate command (i.e., row address with ACTIVATE, column address with READ or WRITE, etc.).

Once the address has been mapped to the appropriate resources, and the correct DRAM commands have been created, they are placed in the simple controller's *command queue*. The command queue is searched out of order to find any possible command which may be issued while still respecting all timing constraints imposed by the DRAM devices. The simple controller is also responsible for keeping track of and issuing REFRESH commands to the DRAM, in order to prevent data loss from capacitive leakage.

Upon the completion of a READ's data burst from the DRAM to the simple controller, the data must be stored within the simple controller while it is packetized

and returned to the main BOB controller. This data is stored in the *read return queue*. This is an essential new portion of the architecture and was not necessary in a commodity memory controller. It allows the operation of the DRAM to continue while data is being returned to the main BOB controller on the response link bus. If this queue is full, commands can not be issued to the DRAM. If data were to be returned to the simple controller from the DRAM when there is no space in the queue, the data would be lost. With no assurance as to the point in the future when space will be available in the queue, the DRAM must be immediately stalled until space is available.

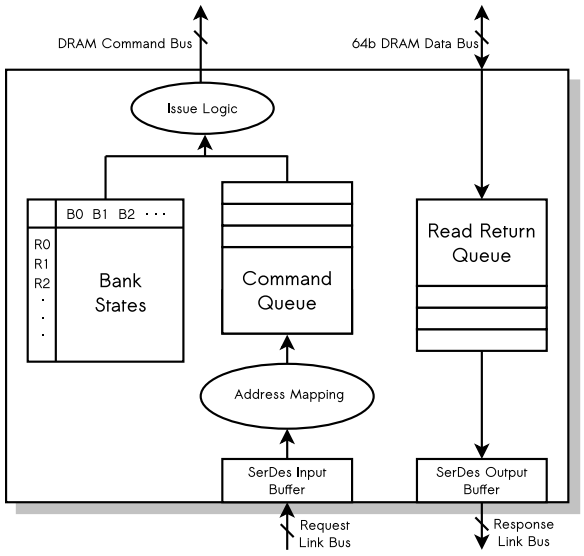


Figure 13: Simple controller block diagram

Similar to a commodity memory controller, the simple controller is responsible for the row buffer management policy. This policy dictates at which point a row of the DRAM should be precharged following a column access. In a BOB memory system, the simple controller uses a closed-page row buffer management policy. With this policy, a PRECHARGE command is issued to the DRAM as soon as possible (while

still adhering to timing constraints) following a column access (either a READ or WRITE). A closed-page policy is used for three important reasons.

First, the closed-page policy favors systems with large processor counts such as servers which typically use this policy by default. With a large number of processors executing numerous threads concurrently, the intermingling of request streams issued to the memory system will tend to negate any address space locality [24]. Therefore, the likelihood of having subsequent column accesses to the same row is greatly reduced making an *open page* policy ineffective. Since the BOB memory system is targeted at server-based systems, the same principles apply.

Secondly, one of the main benefits of the implementation of a BOB memory system is the increased concurrency available within the memory. This reduces the likelihood of a resource conflict. At the same time, using a greater number of logically independent DRAM channels will reduce the likelihood of subsequent requests being mapped to the same row, thus reducing the overall effectiveness of an *open page* policy even further.

Lastly, the logic required to implement an *open page* row buffer management policy is significantly more complex than a closed-page policy. A greater amount of state is necessary to ensure adherence to all timing constraints, and various heuristics are required to prevent request starvation and refresh timing violations. This would therefore make the simple controller more expensive to implement and require a greater amount of power to operate. One of the main reasons FB-DIMM failed was because the introduced logic (AMB), which facilitated the use of narrow buses, also required excessive power and generated excessive heat. Increasing the complexity and

power consumption of the simple controller could result in similar issues in a BOB memory system.

2.4 Packets

Due to the relatively narrow width of each link bus used within the BOB memory system, a packetized interface is required between the main BOB controller on the CPU and each simple controller. Two types of packets are used: a *request packet* and a *response packet*. Each packet is sent on the corresponding link bus (i.e., request packets are sent on the request link bus and response packets are sent on the response link bus) over multiple clock cycles. The format and total size of these packets is important for the generalized model as it determines the amount of time it takes a request or response packet to traverse each link bus. The format for these packets can be seen in **Figure 14**. While certain fields within the packet might not be fully utilized, it is important that the total size of the packet be some even factor of its respective link bus's width to ensure that link bus cycles are not wasted sending only a portion of the packet.

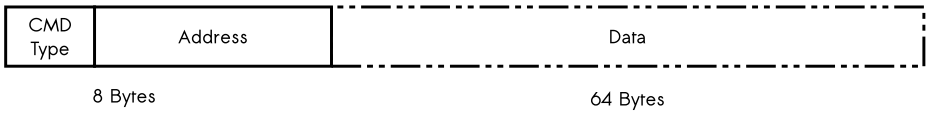


Figure 14: BOB packet definition

A *request packet* being sent on the request link bus must contain the request's address, the request type, and the data if said packet is a write request. The type of request can either be a read or write. The amount of data included in a write

request packet is always 64 bytes since the DRAM attached to each simple controller are commodity devices which expect that amount of data. When a write request is received, this data is stored in a queue while awaiting the WRITE command to be issued to a particular rank of DRAM.

A *response packet* will contain data requested by a READ command as well as the address of the initial request for identification purposes. This is necessary due to out-of-order issuing of requests both within the BOB controller and simple controller, and may be completed at different times. The order of requests to the same address is always maintained.

3 BOB Simulation Suite

To properly evaluate this new architecture, a simulation suite is developed with a strong focus on hardware verification and comprehensive, detailed system modeling. Two separate simulators are used in this suite: a BOB memory system simulator developed at the University of Maryland, and MARSSx86 [31], a multi-core x86 simulator developed at SUNY-Binghamton. Together, they create an accurate model of a processor which boots an operating system, launches an application, and interacts with the cache and memory system.

3.1 Simulation Framework

The BOB memory system model is a cycle-based simulator written in C++ that encapsulates the main BOB controller, each BOB channel, and their associated link bus and simple controller. Each of the major logical portions of the design have a corresponding software object and associated parameters that give total control over all aspects of the system's configuration and behavior. Some simple examples include the type of DIMMs and number of ranks within an individual BOB channel, the total number of BOB channels, or speed and width of each link bus. The BOB simulator may be run in one of two modes – a stand-alone mode where requests from a parameterizable, random address generator are issued directly to the memory system or a full-system simulation mode where the BOB simulator receives requests from MARSSx86. A google-perftools analysis and call-graph can be seen in **Appendix A**.

Simulating a BOB memory system in stand-alone mode may provide many insights

into a system's behavior, yet a full-system simulation is the most ideal situation as it will show important interactions and behaviors which might not have been obvious otherwise. In order to perform a full system simulation, an accurate CPU and cache simulator must be selected and integrated. After considering several CPU simulators, MARSSx86 is selected. MARSSx86 merges the highly detailed, out-of-order x86 pipeline models from PTLSim [41] with the QEMU emulator. MARSSx86 augments the original PTLSim models with multi-core simulation capability and a configurable coherent cache hierarchy.

The ability to simulate multi-core environments is critical since multithreaded workloads are quickly becoming the rule rather than the exception. Additionally, it is difficult to imagine a single threaded application being able to take full advantage of the tremendous bandwidth provided by a BOB memory system. MARSSx86 provides a full system simulation capabilities that allow the simulator to capture the effects of the cache, virtual memory, and kernel interaction. These things are key factors in the efficient operation of the memory system. The CPU models are highly configurable, and it is possible to change the internals of the CPU or behavior of caches to take advantage of new features (for example, replacing a traditional memory bus with a number of ports).

To attach the BOB memory system model to the MARSSx86 simulator, the memory controller class in MARSSx86 is modified to reroute requests and responses. As requests arrive at the memory controller within MARSSx86, they are sent to the main BOB controller's ports. The heuristic for assigning requests to specific ports can be altered – for example, requests could round robin over all ports, or a specific

set of cores might always use a specific port. The implications of this concept is fully explored in the full-system simulation section (4.2). If there are no available ports, or all port buffers are full, the CPU will be stalled until requests can be issued.

The MARSSx86 memory hierarchy contains a clock signal which is used to drive the clock of the BOB simulator. Since a BOB memory system contains multiple clock domains, the clock provided by the MARSSx86 simulator is multiplied or divided to create the correct frequency for each portion of the architecture. Once a memory request is complete and finished being sent out of the main BOB controller's port, the data is returned to the MARSSx86 memory controller using a callback function. The memory controller then sends the completed requests back up the cache hierarchy to the CPU.

Just as in a real CPU, thread execution is stalled while waiting for a memory request to complete. This interaction between the CPU and memory system is key because it shows the impact of an optimally configured memory system and the impact the memory system can have on a program's execution. DRAMSim2 uses a similar method to connect to MARSSx86 [32], which makes it easy to run the same workloads in both simulators to compare the system level benefits of a BOB memory system over a traditional DDR2/3 memory system.

3.2 Hardware Verification

An important aspect of this simulation framework is its ability to validate simulated behavior against that of actual hardware. Since the DIMMs used in a BOB memory

system utilize the same DRAM devices, same interface, and same timing constraints as those in a commodity system, validating this portion of the simulator can be achieved in a manner similar to that of DRAMSim2 [32]. Micron Technology publicly provides Verilog HDL models for each of the DRAM devices that it produces. These models determine whether or not a timing constraint has been violated based on a series of inputs from a hardware behavioral simulator like ModelSim. Therefore, with simple Python scripts to massage simulator output, the validity of the simulation can be confirmed.

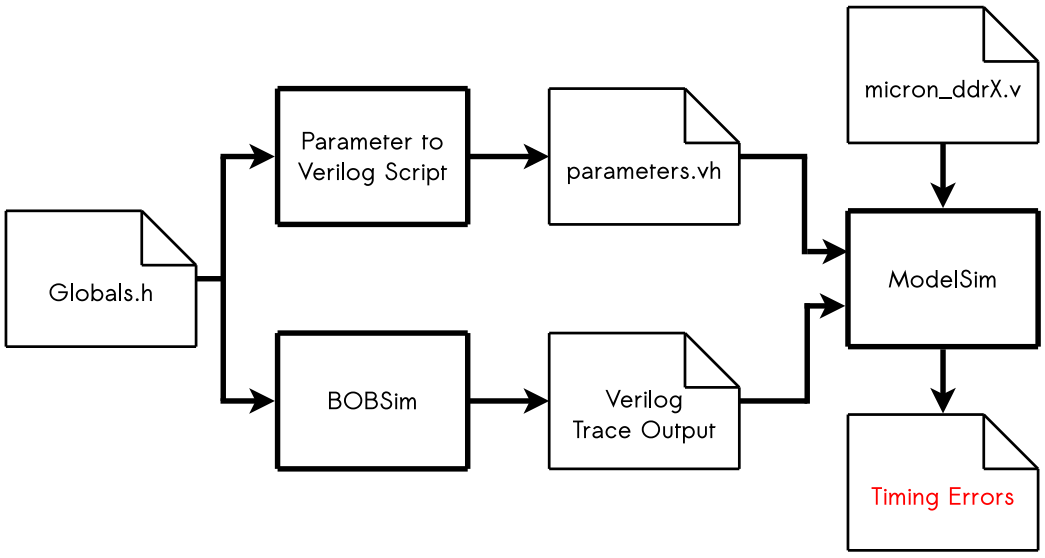


Figure 15: The verification process using ModelSIM

During a BOB memory system simulation, each DRAM channel produces a bus trace file consisting of a command (i.e., ACTIVATE, READ, WRITE, PRE) or data and the cycle on which it was issued. This file is post-processed by a separate Python script that generates a series of Verilog compatible commands. These Verilog commands are used in conjunction with ModelSim and the Micron HDL models to ensure the timing of both commands and data issued on the bus are cycle accurate at the

DRAM level. The parameters used in each simulation are parsed out of the global BOBSim header file (which defines all parameters in a simulation) and are placed in a Verilog header file that tells ModelSIM specifics about the device it is simulating. A block diagram of this process can be seen in **Figure 15**. The BOB simulator uses timing and device parameters from a Micron DDR3-1066 device (MT41J512M4-187E), a DDR3-1333 device (MT41J1G4-15E), and a DDR3-1600 device (MT41J256M4-125E), yet any JEDEC standard device will work.

The specific signals which are used in this verification are the signals which make up the command bus (RAS, CAS, WE), the address bus (for bank, row and column), and the data bus. For all simulations, the first one million DRAM cycles are verified with the above technique; all are always successful. While the overall purpose of utilizing ModelSIM's behavioral modeling abilities is to ensure timing constraints are not violated within BOBSim, it also provides the ability to visualize the operation of the DRAM. In **Figure 16**, a read and write are issued and the corresponding signals are shown accordingly.

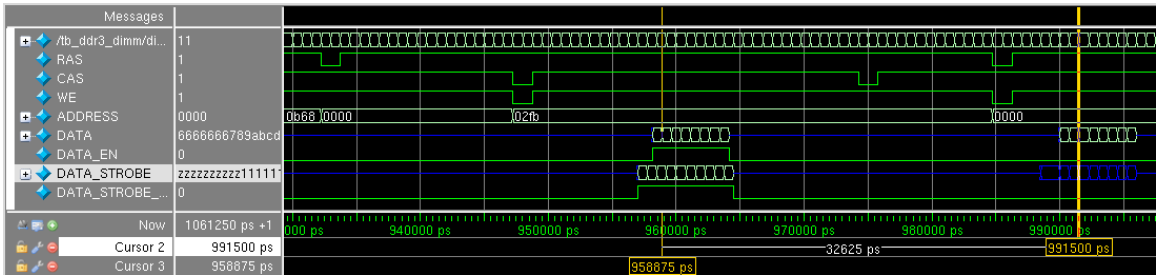


Figure 16: A read and write cycle shown in ModelSIM

4 Simulation Results

When evaluating the characteristics and behavior of this new architecture, we performed two experiments: a limit-case simulation where a random address stream is issued into a BOB memory system as fast as possible and a full system simulation where an operating system is booted on an x86 processor and applications are executed. The limit-case study is useful for identifying the achievable maximum sustained bandwidth and the behavior of the system in extreme situations. Many server and HPC applications generate address streams that have little locality (temporal or spatial) and appear random. In contrast, a full system simulation gives a much more realistic picture of the new memory system's interaction with the cache and processor, operating system, and actual applications.

A host of benchmarks are selected with an emphasis on multi-threaded workloads to demonstrate the types of request streams the BOB memory system is likely to encounter. It is necessary to have a wide variety of benchmarks because variations in the memory request stream cause vastly different behaviors and subsequent performance. The list of benchmarks is:

- The PARSEC benchmark suite is a set of multi-threaded benchmarks designed by Princeton aimed at testing shared-memory CMPs. The memory intensive benchmarks of the suite include *fluidanimate*, *facesim*, and *bodytrack*.

- The STREAM benchmark is a memory bandwidth test designed to demonstrate real world achievable performance via software. This benchmark is run for 10 iterations and uses a two million element array.

- GUPS (Giga-Updates Per Second) is a benchmark designed to mimic the computational and memory behavior of sparse matrix updates, graph traversals, and cryptographic algorithms. This benchmark performs 256M updates on a 1GB array with a 64-byte element size.

- Sandia National Lab's implementation of GUPS. This benchmark uses a 4GB array size and performs 2 million updates.

- The NAS parallel benchmark suite is a set of fluid dynamics computations released by NASA.

- MCOL is a benchmark which scans over a matrix in memory. It uses eight threads to perform 25 iterations over a 64MB array.

When considering the cost of a particular system's implementation, design tradeoffs that are considered include total pin count required by the CPU, power dissipation of both DIMMs and simple controllers, and the physical space required (or total DIMM count).

4.1 Limit-Case Simulations

For the limit-case simulations, the BOB simulator is run in a stand-alone mode where memory transactions are added directly to the BOB controller via an execution wrapper. This wrapper is also responsible for collecting and analyzing useful statistics. The generated request stream can be tailored to issue at a specific frequency or read-write ratio. For these simulations, requests are issued as soon as possible with a mix

of 2/3 reads and 1/3 writes.

4.1.1 Simple Controller & DRAM Efficiency

Commodity memory system design has been examined and analyzed extensively [20, 24, 39, 19]. Because each BOB channel uses commodity DIMMs, operates on the same data and command buses, and requires the same operating protocol and timing constraints, it stands to reason that the previous insights, optimizations and analysis targeting commodity systems should apply here as well. When observing an individual BOB channel's behavior, the simulations confirm and reinforce these previous insights. For example, the impact of an increasing number of ranks of DDR3-1066, DDR3-1333, and DDR3-1600 on the DRAM bus efficiency can be seen in **Figure 17**. Efficiency begins to drop after two and four ranks due to the increased necessity of idling the data bus for arbitration when switching between the ranks. While the use of 8 and 16 ranks of DRAM is uncommon due to electrical constraints, this shows that even the logical behavior of the system displays a decrease in performance when faced with an increasing number of ranks. The peak efficiency achieved is also verified by manufacturers results [5] and prior research [36].

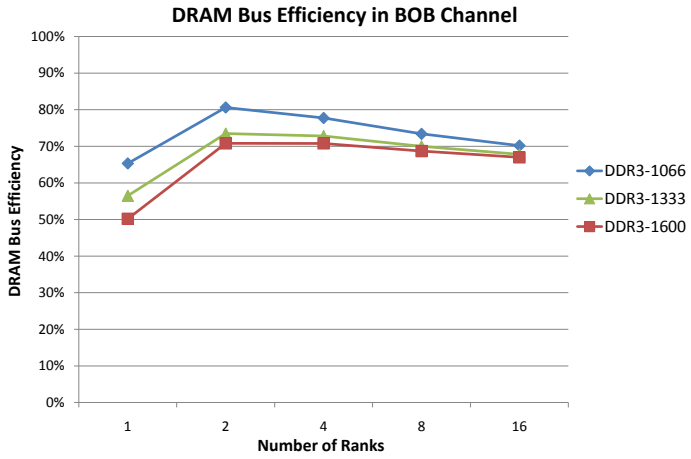


Figure 17: Increasing number of ranks in memory system has a negative impact on DRAM bus efficiency

In a similar vein, the simple controller in each BOB channel is comparable to a typical memory controller in a standard memory system. Therefore, characteristics and optimizations which have been previously identified in commodity memory controllers should apply here as well. Parameters such as address mapping and queue depths have shown to have a significant impact on performance [39], which is also the case in BOB systems. Increasing the out-of-order depth and command queue size allows the simple controller to more easily find commands to issue (while still adhering to the timing constraints), thereby increasing the DRAM bus efficiency (**Figure 18**). The increased queue depth and search facility will increase costs such as die size and power consumption. Therefore, an appropriate out-of-order queue depth can be determined either by the required performance or outside constraints, such as target power consumption or transistor count. Given this, there are still diminishing returns after a depth of eight; doubling the queue to 16 only increases DRAM efficiency by approximately 5% in all cases.

Changing some parameters within the simple controller will have no impact on the performance during a limit-case simulation due to the random nature of the address stream. Parameters such as the address mapping scheme, which normally have a significant impact on performance, will not have any effect due to the equal likeliness of all bit combinations within the address. Other parameters, such as the return queue depth will have different behaviors under the limit-case compared to that of a full-system simulation, since the random address stream ensures an equal spread of requests both spatially and temporally. These parameters are explored further when simulations are run in full system mode with MARSSx86.

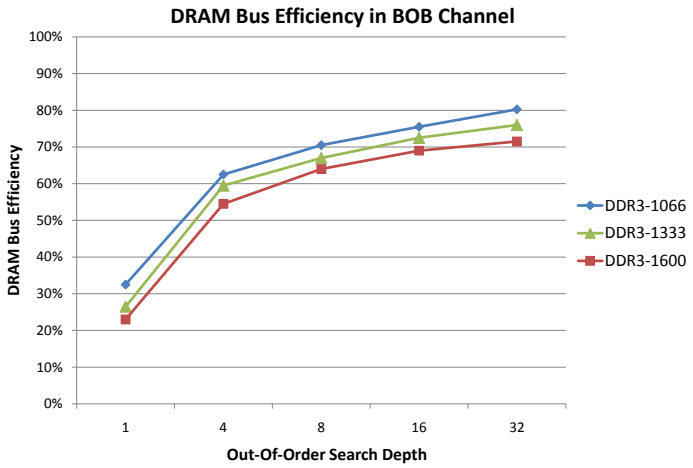


Figure 18: Increasing the out-of-order search depth increases DRAM bus efficiency

The simple controller must also have features that differentiate it from a commodity memory controller as a result of serializing communication on the link bus. The read return queue within each simple controller is responsible for storing requested data before it is packetized and transferred out on the response link bus back to the main BOB controller. If this queue is full, no further read or write commands will be issued to the DRAM until there is space within this queue. While write commands

do not require space in this queue, they are still stalled to maintain proper request ordering. It is possible to continue issuing ACT commands to prepare rows for access, although this will increase background power consumption since rows will be left open for a longer amount of time (within limit-case simulations, the ACT commands are continually issued when possible).

The rate at which items are removed from this queue is determined by both the width and speed of the response link bus. A parameter sweep is performed on both the depth of the read return queue and the configuration of the response link bus to detail the impact these decisions have on the achievable efficiency of different speeds of DRAM. The results can be seen in can be seen in **Figures 19, 20, & 21**.

The efficiency of all DRAM speeds is greatly impeded when the read return queue only provides storage capacity for a single response. To prevent overflowing the queue, space in the queue is reserved for the incoming data upon issuing the READ command to the DRAM. Therefore, when the queue depth is one, all subsequent requests are stalled, not only while the response is being sent over the response link bus, but while the data is being retrieved from the DRAM as well. Therefore, as soon as a single READ command is issued, all requests are stalled until the data has been retrieved from the DRAM and it has finished being sent back to the main BOB controller. This situation is entirely too restrictive and performance can be more than doubled by simply having a queue depth of two.

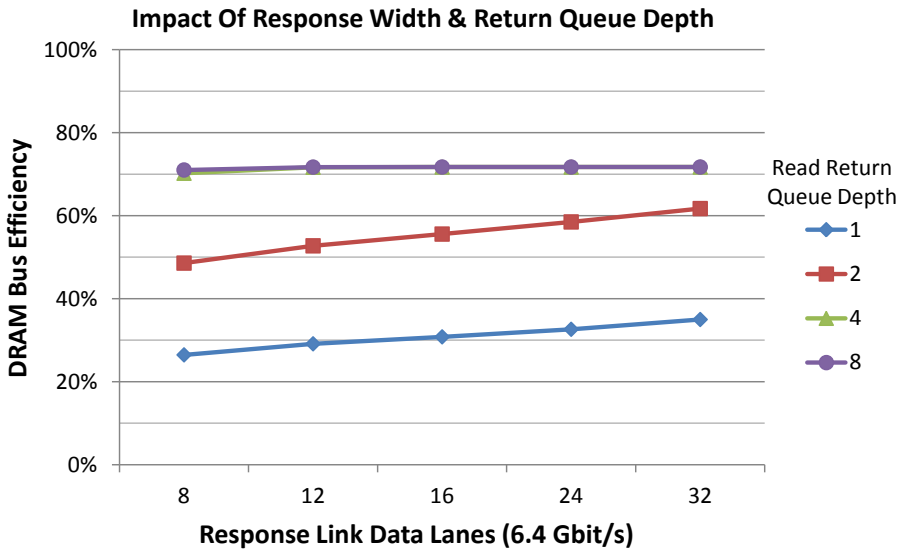


Figure 19: Response link and return queue depth impact on DDR3-1066 DRAM bus efficiency

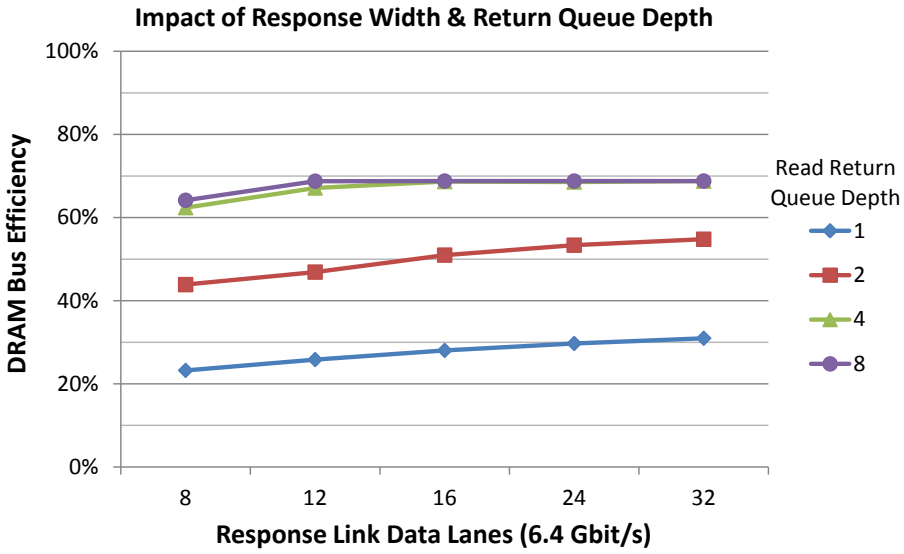


Figure 20: Response link and return queue depth impact on DDR3-1333 DRAM bus efficiency

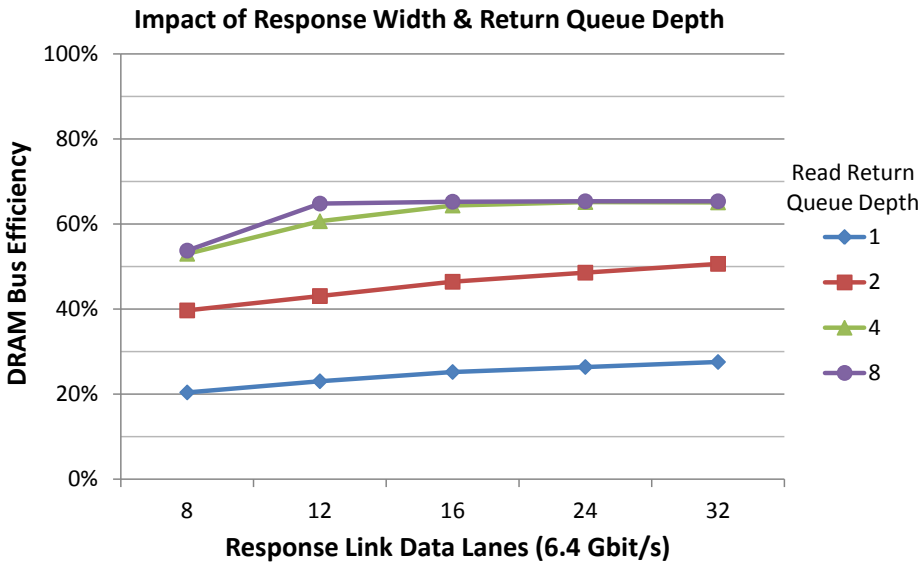


Figure 21: Response link and return queue depth impact on DDR3-1600 DRAM bus efficiency

With a queue depth of two and four, the efficiency is significantly increased. The increase in storage capacity allows a better utilization of the parallelism available within modern DRAM devices. As the response link bus is widened, response packets are removed more quickly thereby clearing room in the queue for subsequent requests. With a depth of four, the DRAM is rarely stalled as a result of the read return queue reaching maximum capacity. The gains seen by increasing the response link bus width eventually taper off as the DRAM has already achieved maximum attainable efficiency.

In **Figures 19, 20, & 21**, the link bus is clocked at 3.2 GHz and utilizes double data rate transferring of packets (6.4 Gbit/s). If the link bus clock is increased, the width of the bus becomes less of a determining factor in the efficiency of the DRAM. When the link bus is clocked at 6.4 GHz (12.8 Gbit/s), 8 data lanes is already sufficient to reach peak efficiency (**Figure 22**). While a 9.6 GHz link bus might not be feasible

now, if future technology enables this to be possible, the width of the link buses can be reduced even further without any negative impact on performance. This would greatly save on CPU pin-out costs, an important factor in chip fabrication.

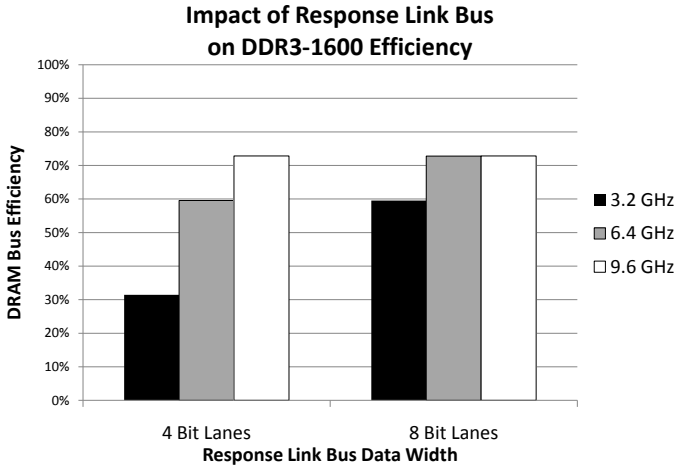


Figure 22: Response link bus configuration's impact on DRAM efficiency

It is possible to use parameters and behaviors of other parts of the system to help quantify decisions about the read return queue and response link bus. The DRAM timing parameter tFAW was introduced to prevent large current draw while performing numerous concurrent row-activation operations. The timing parameter dictates a sliding-window of time in which at most four ACTIVATE commands may be issued. Indirectly, this also determines the longest possible period of uninterrupted data being retrieved from the data bus (**Figure 23**). This situation can be used as a lower bound for the read return queue capacity. Evidence of this can be seen in **Figures 19, 20, & 21**. When the queue depth is four, the system is capable of reaching peak efficiency once the response link bus bandwidth is capable of removing the four requests fast enough. This is not the case for more shallow queue depths.

From the DRAM perspective, this is the worst case scenario for the read return queue but will be explored in the context of the whole system in later sections.

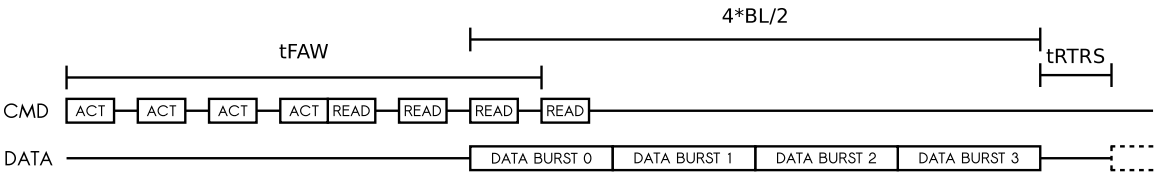


Figure 23: The impact of tFAW on the DRAM operation and the longest period of uninterrupted data retrieval

4.1.2 Link Bus Configuration

The overall performance of a BOB memory system is inherently linked with the efficiency of each DRAM channel. Therefore, optimal system configurations are ones in which the request link bus and response link bus do not negatively impact the DRAM efficiency. The width and speed of these buses should be configured such that request and response packets can be sent at a rate that does not stall the DRAM, either due to a lack of available requests issuable to the DRAM or due to an inability to clear the read return queue quickly enough.

As previously mentioned, the response link bus is responsible for removing response packets from the read return queue. The fastest rate at which data from the DRAM can be added to this queue is determined by the burst length (BL) and tFAW DRAM timing constraints (**Figure 23**). As described above, it is possible to calculate a first-order approximation for the necessary response link bus bandwidth when taking into account the DRAM's tFAW (37.5ns for DDR3-1066 and 30ns for DDR3-1333 and DDR3-1600) and BL values (7.5ns for DDR3-1066, 6ns for DDR3-1333, 5ns for DDR3-1600). This calculation can be seen in **Equation 1** and dictates

how quickly the response link bus must transmit four requests in order to prevent stalling the DRAM due to lack of space in the read return queue.

$$TimeToTransmit = \frac{BL}{2} \times 4 \times tCK + tRTRS \times tCK \quad (1)$$

This equation represents the length of time between the start of the longest possible continuous data burst and the next possible chance for new data to arrive from the DRAM (situation depicted in **Figure 23**). The first term ($BL/2 \times 4 \times tCK$) is the total time (in nanoseconds) that these four continuous data bursts take to be retrieved from the DRAM. The second term ($tRTRS \times tCK$) is the time (in nanoseconds) that it takes to switch between transmission sources – in this case, separate ranks attached to the DRAM bus. When these terms are summed, the result represents the period of time in which the greatest quantity of continuous data can return in the shortest amount of time. This is solely a result of the tFAW timing constraint.

If the response link bus bandwidth is capable of transmitting four response packets within this computed time, then, from the DRAM's perspective, it would be impossible to stall as a result of the return queue being full. Other parts of the system may have an impact on this and will be explored in later sections. The length of time to send a single 72B response packet for numerous response link bus configurations can be seen in **Table 1**. When relating these values to the times computed with **Equation 1**, it is clear that certain response link bus configurations would not be able to remove response packets from the read return queue fast enough to prevent it from reaching maximum capacity and thus causing the DRAM to stall.

Data Lanes	3.2 GHz	6.4 GHz	9.2 GHz
4	22.5 ns	11.25 ns	7.5 ns
8	11.25 ns	5.625 ns	3.75 ns
12	7.5 ns	3.75 ns	2.5 ns
16	5.625 ns	2.8125 ns	1.875 ns

Table 1: Time to transmit response packet over response link bus

The request link bus width and speed will have an impact on the DRAM efficiency as well. In a BOB memory system, the request link bus is responsible for issuing read request packets (8 bytes) and write request packets (8 bytes of overhead and 64 bytes of data) to their respective simple controllers. The request link bus must be able to send these packets at a rate which keeps the DRAM as busy as possible. When a simple controller's work queue is full, the issuing logic is more likely to find a command which can be issued within the timing constraints imposed by the DRAM.

An accepted rule-of-thumb is that a typical request stream will have a read-to-write request ratio of approximately 2-to-1. Implemented systems have accounted for this fact by weighting response paths more than requests paths. This can be seen starting from the FB-DIMM standard, which had the northbound bus (for responses) 40% larger than the southbound bus (for requests) [9]. The new architectures detailed above adopt this convention as well with Intel's SMI response bus 33% larger [13] than the request bus, and IBM's Power7 system whose response bus is twice as wide as the request bus [12].

Limit-case simulations are performed with eight DRAM channels of various speed grades that are attached to different request and response link bus configurations (operating at 3.2 GHz). The results can be seen in **Figures 24, 25, & 26**. The

simulation results show a clear peak bandwidth where additional link bus bandwidth has no impact on the overall performance. This is due to the link bus bandwidth exceeding that which is required by the DRAM to reach its maximum attainable efficiency.

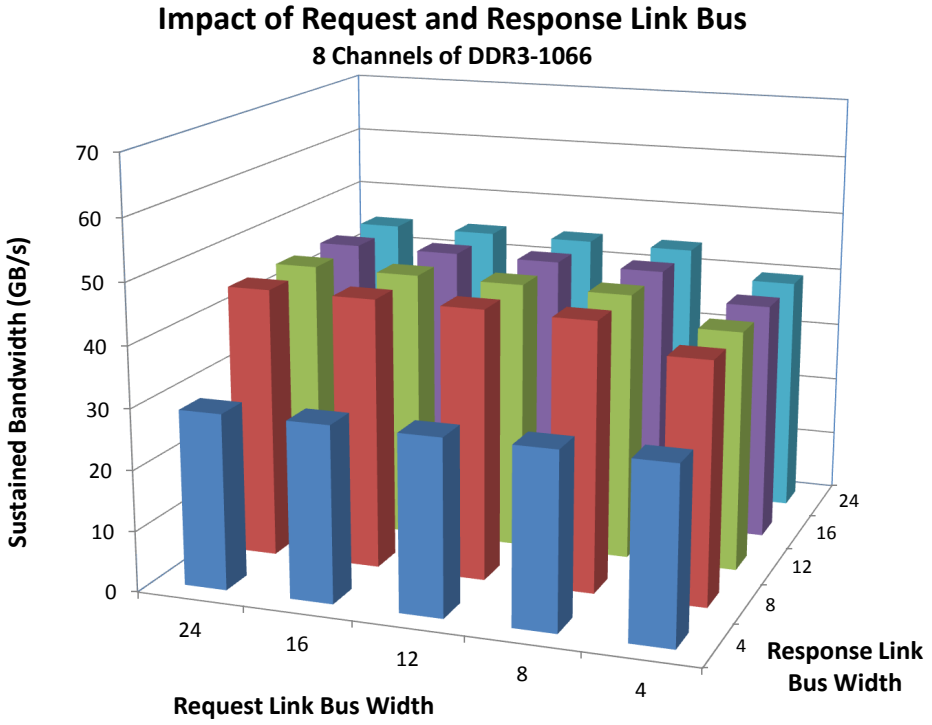


Figure 24: Sustained bandwidth of eight DDR3-1066 DRAM channels using various link bus configurations

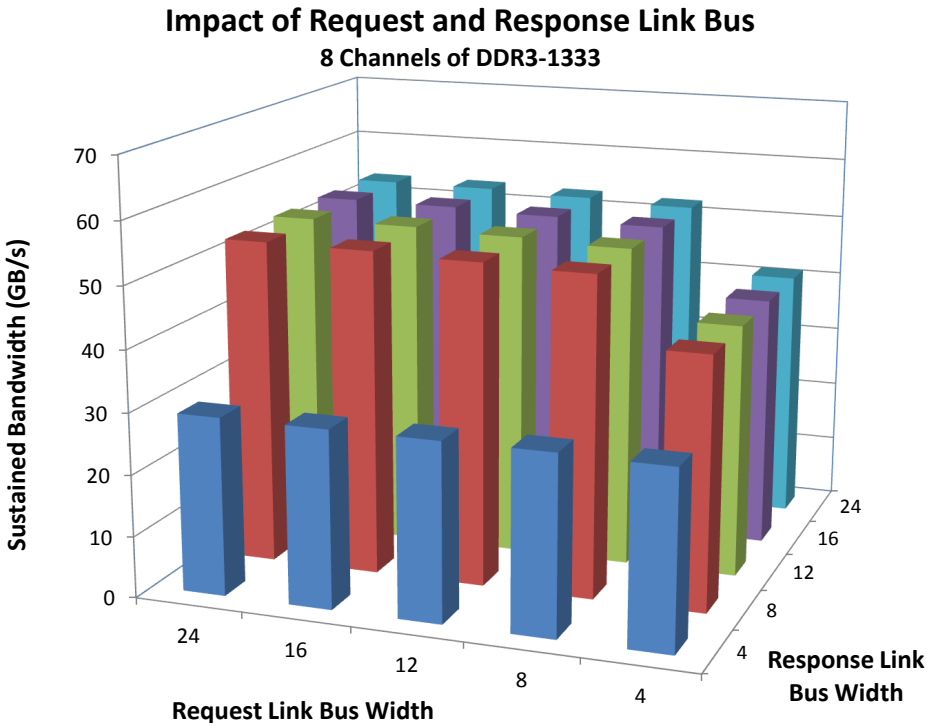


Figure 25: Sustained bandwidth of eight DDR3-1333 DRAM channels using various link bus configurations

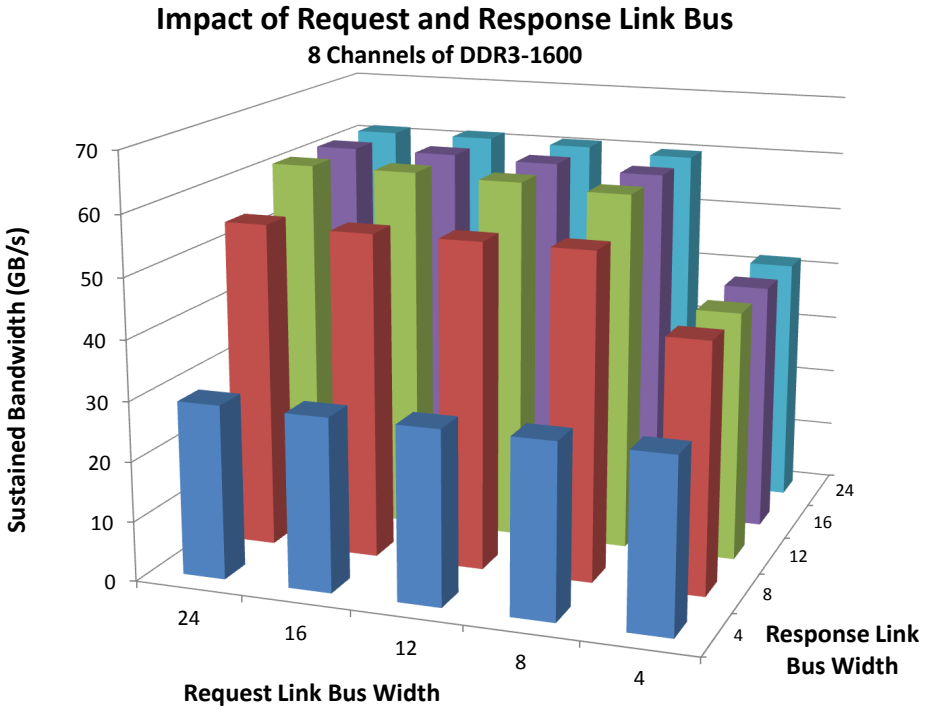


Figure 26: Sustained bandwidth of eight DDR3-1600 DRAM channels using various link bus configurations

To prevent the request and response link bus from having a negative impact on DRAM performance, each must be capable of meeting bandwidth requirements dictated by the DRAM and request stream. These requirements have several factors. First, the achievable DRAM bandwidth determined by the speed grade and the expected efficiency of that device is clearly a main factor in how each link bus should be configured. Second, like FB-DIMM [22], the read-write ratio has a significant impact on the utilization of each link bus (explored in detail later). Lastly, because the link buses are responsible for transmitting packets and packet overhead as well as data, this must be accounted for as well.

Incorporating all of the factors listed above leads to **Equations 2 & 3**; these equations dictate the bandwidth required by each link bus to prevent them from negatively impacting the efficiency of each channel. BW_{DRAM} represents the peak bandwidth of the DRAM devices in use and $Efficiency$ is the peak expected efficiency achievable by these same devices. The product of these two terms is the peak bandwidth the DRAM is capable of achieving. The $\%Reads$ and $\%Writes$ terms represent the ratio of read and write requests within the request stream. These factors are multiplied by the amount of data which must be moved to make these requests, including the size of read request packets ($ReadPacketSize$), the overhead of a write packet ($WritePacketSize$), the size of each request ($RequestSize$), and the overhead for each response packet ($ResponsePacketSize$).

$$BW_{Request} = (BW_{DRAM} \times Efficiency) \times \left[\%Writes \times \left(1 + \frac{WritePacketSize}{RequestSize}\right) + \%Reads \times \frac{ReadPacketSize}{RequestSize}\right] \quad (2)$$

$$BW_{Response} = (BW_{DRAM} \times Efficiency) \times \left[(\%Reads \times \left(1 + \frac{ResponsePacketSize}{RequestSize}\right)\right] \quad (3)$$

In **Figures 24, 25, & 26**, link bus configurations which have a bandwidth equal to or greater than the values dictated by these equations are capable of achieving the peak possible bandwidth for the simulated system. These results also confirm the predictions made about the link bus in **Equation 1**; when the response link bus is capable of moving requests comparable to this rate, maximum attainable DRAM efficiency is achievable.

The unidirectional nature of each link bus causes sensitivity to the read-write request mix similar to that seen in FB-DIMM [22]. While weighting the response link bus more than the request link bus might be ideal for many application request streams, performance will be significantly different as soon as the request mix changes. **Figure 27** shows the impact of different read-write request ratios on weighted and unweighted link bus configurations during limit-case simulations. Intuitively, when the request mix is weighted in the same fashion as the link buses, the DRAM can reach maximum attainable efficiency. Unfortunately, this behavior is an unavoidable side-effect of serializing the communication on unidirectional buses and a decision should be made based on the most likely workload the memory system will see.

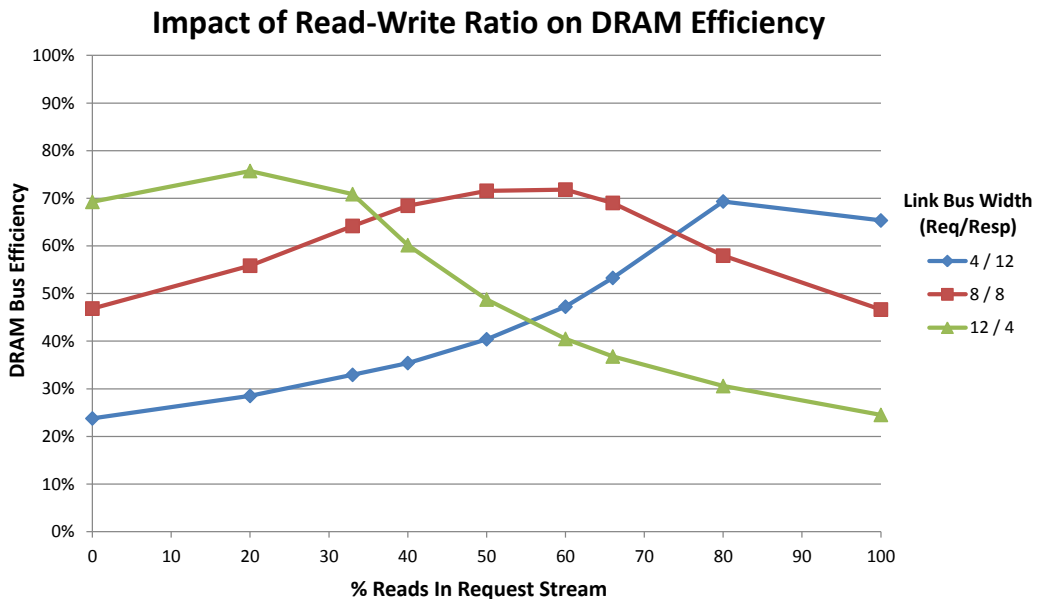


Figure 27: The effect of the request stream mix on weighted and unweighted link bus widths

While there are many benefits afforded by using a BOB memory system, such as reduced pin-out of the CPU, higher possible bandwidth, and increased concurrency, there is a drawback which has not yet been discussed : the latency penalty incurred by serializing requests and responses and the transmission time over the request and response link bus. Regardless of how fast or wide each of the link buses are, the overall latency of a single request will always be longer than in a commodity memory system. This phenomenon is similar to that seen in an FB-DIMM memory system, which must also serialize requests and responses to account for the new protocol and narrow buses. At lower utilization, requests in an FB-DIMM memory system experience a 25% degradation in overall latency [22].

To analyze this latency penalty, the total latency of a request is divided into components which correspond to the various parts of the system and the time a

request spends there (**Note:** In this analysis, only read requests are examined as the latency of a write request is typically meaningless and difficult to measure). Aside from the latency seen from retrieving the data from the DRAM which is dictated by the standard DRAM timing constraints, other latency components now include time spent on both the input and output ports, time spent on the request and response link buses, and the time spent in the work queue and read return queue. To measure each of these components, a single read request is issued to an empty memory system using DDR3-1333 and the time spent at each point is recorded. The resulting latency seen by a read request when being issued to BOB memory system with various request and response link bus widths can be seen in **Figure 28**.

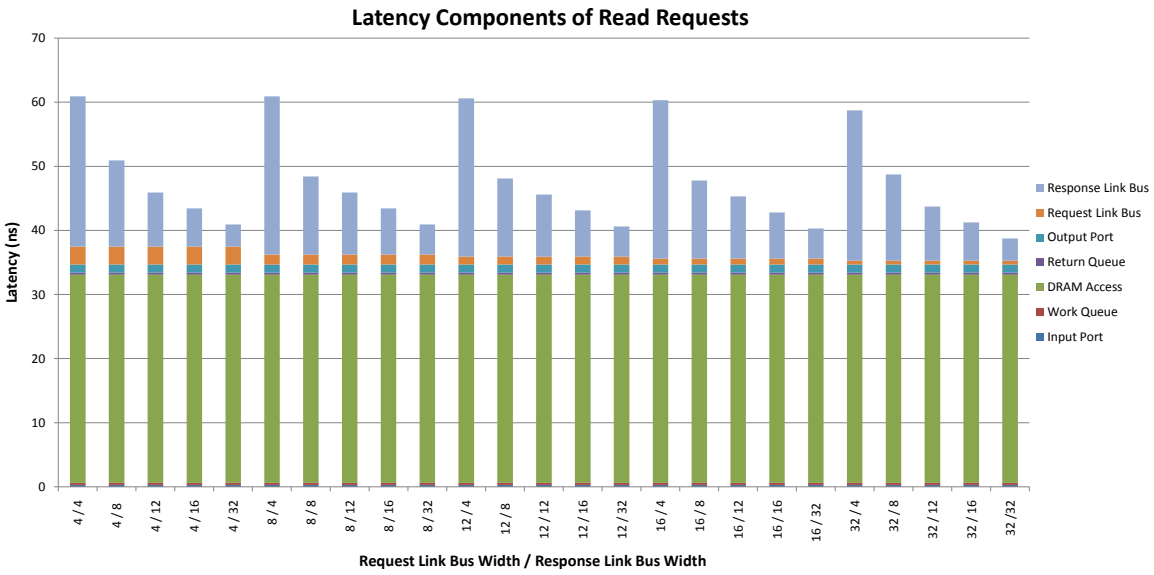


Figure 28: Latency components of a single read request with various BOB system configurations

As expected, the DRAM access time is uniform across all configurations. This latency consists of the time between when the ACT command is issued and the end of the resulting data burst from the column access. This latency will only change with

a different speed DRAM device or when the memory system is being heavily utilized and column access commands are delayed to account for other timing constraints. The latency component which has the largest impact on the overall latency is the time spent on the response link bus. This is a result of the 72 byte response packet generated by a read request that must be returned on the response link bus. As the response link bus widens and provides additional bandwidth, a response packet can be transferred back to the main BOB controller in less time. The request link bus latency is not as much of a factor as the response bus latency but still has an impact; in some cases it accounts for approximately 7% of the request latency. A read request must only send an 8 byte request packet on the request link bus and can be sent in far less time than the 72-byte response packet.

Due to the way these latencies were measured, the time spent in both the work queue and read return queue does not provide any insight into a request's overall latency. The depth of these queues and the activity of the memory system will have a significant impact on a request's latency, but these times are meaningless in a limit-case simulation. These values will be further explored with full-system simulations when the request stream is the result of an actual application issuing requests to the memory system.

4.1.3 Peak Possible Bandwidth

While other aspects of a BOB memory system have an impact on the overall performance, the peak sustainable bandwidth is determined by the total number of concurrent and logically independent channels of DRAM. When other parameters are

configured in a way to ensure maximum DRAM efficiency, this peak bandwidth is simply a product of the sustained DRAM bandwidths and the total number of channels. For example, if parameters are chosen such that the DRAM is not impeded in any way, increasing the number of BOB channels makes it is clear that the sustained bandwidth is simply the aggregate of the maximum sustained bandwidths across all the DRAM channels (**Figure 29**).

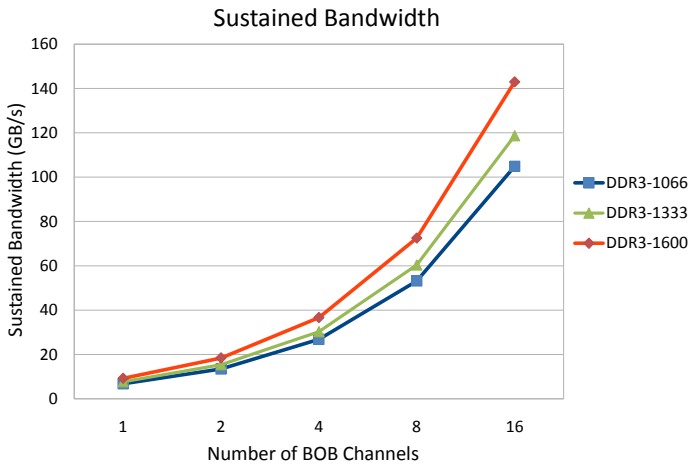


Figure 29: Sustained bandwidth with various numbers of BOB Channels

Unfortunately, increasing the total number of BOB channels to improve performance in this fashion is expensive for multiple reasons. The most obvious constraint is the physical space required to have numerous, concurrent DRAM channels, each of which requires the wide, standardized DDR3 bus and attached DIMM slots. There is also the cost of the pin-out required by the CPU to communicate over each of the channel's request and response link buses. While the link bus widths are insignificant relative to the costs in pin-out of a standard DDR3 bus, the high-speed nature of the lanes which comprise each link bus make it essential to differentially signal, and thus double, the total pin out (not including additional power, ground, and CRC lanes).

Finally there are the costs involved with requiring a simple controller for each channel. These costs include the fabrication cost for each controller, the power involved to operate them, and the physical space needed to place them on the motherboard. The following section describes an optimization which can significantly reduce these costs while maintaining acceptable performance and storage capacity.

4.1.4 Multi-Channel Optimization

If the link bus configurations provide bandwidth that can not be fully utilized by a single logical channel of DRAM (and that exceeds the calculated values from **Equations 2 & 3**), it is possible for multiple, logically independent channels of DRAM to share the same link bus and simple controller without negatively impacting performance. This will reduce the costs detailed above. The pin-out of the CPU can be reduced for an equivalent number of DRAM channels since fewer link buses are required. This will also reduce the number of simple controllers, which will reduce fabrication costs and the physical space necessary to place them on the motherboard.

While reducing these costs, it is important to note that the complexity of the simple controller will increase at the same time. The pin-out of the simple controller must be increased to support multiple DRAM channels, and the logic within must be replicated for each of the logically independent channels (which will in turn increase the power requirements of each controller). In order to implement a configuration where multiple DRAM channels share a single link bus, arbitration is required within the simple controller to route requests and determine who can send responses. To reduce the overall complexity of the simple controller design, a simple round-robin

scheduling mechanism was implemented. This can be seen in **Figure 30**.

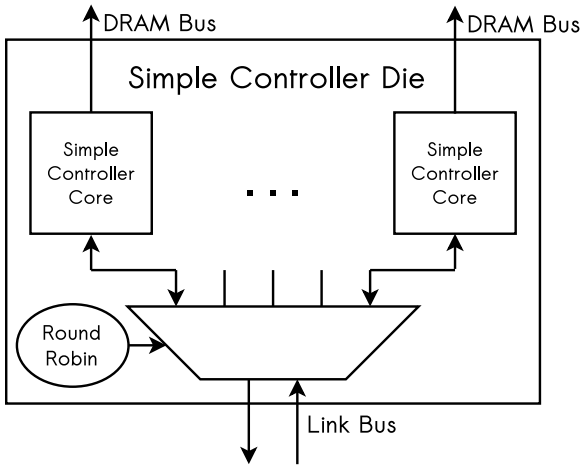


Figure 30: Arbitration between simple controller cores

The link bus bandwidth requirements defined by **Equations 2 & 3** can easily be modified to account for this optimization. Each request and response link bus must now be capable of meeting the bandwidth requirements of all the DRAM channels which are attached. In **Equations 4 & 5**, $NumDRAMChannels$ is the integer value corresponding to the number of independent DRAM channels which are now attached to the link buses and $BW_{Request}$ and $BW_{Response}$ are the values computed in **Equations 2 & 3**. Because **Equations 2 & 3** represent the bandwidth requirements of a single DRAM channel, it stands to reason that with the multi-channel optimization, a link bus must capable of handling all the bandwidth required by all channels; therefore the product of these two values dictates the necessary bandwidth for each request and response link bus in a system with the multi-channel optimization.

$$BW_{Request}{}' = (NumDRAMChannels) \times BW_{Request} \tag{4}$$

$$BW_{Response}{}' = (NumDRAMChannels) \times BW_{Response} \tag{5}$$

An example of the multi-channel optimization can be seen in Intel's SMI/SMB

architecture. Each SMB supports two separate channels of DDR3. Intel's architectural equivalent of the request link bus and response link bus (referred to as the southbound and northbound bus, respectively) are 9 data lanes (7.2 GB/s) and 12 data lanes (9.6 GB/s), respectively [13]. If **Equations 4 & 5** are used to calculate the bandwidth requirements for two channels of DDR3-1066 (as in Intel's SMI/SMB system), the request link bus would require 5.8 GB/s, and the response link bus would require 9.5 GB/s (assuming 66% reads, 33% writes, 75% DRAM efficiency, and 8 byte packet overhead). This can be seen in detail in **Equations 6 & 7**. While the packet overhead is unknown for the SMI, the computed values are similar enough to the implemented values to show that **Equations 4 & 5** accurately predict the bandwidth requirements of each link bus.

$$5.8GB/s = 2 \times ((8.533GB/s \times .75) \times \left[.333 \times \left(1 + \frac{8B}{64B}\right) + .666 \times \frac{8B}{64B}\right]) \tag{6}$$

$$9.5GB/s = 2 \times ((8.533GB/s \times .75) \times \left[(.666 \times \left(1 + \frac{8B}{64B}\right)\right]) \tag{7}$$

Limit-case simulations are performed on systems which utilize the multi-channel optimization to varying degrees in order to demonstrate the impact that this optimization has on system performance. Each configuration has eight DRAM channels of either DDR3-1066, DDR3-1333, or DDR3-1600 and uses a link bus clock of 6.4 GHz. The multi-channel utilization is increased from two to eight. Increasing the degree of multi-channel utilization in this fashion results in fewer request and response link buses in that system. For example, with a multi-channel utilization degree of two, the system has four link buses and simple controllers; with a multi-channel utilization degree of four, the system has two link buses (displayed in **Figure 31** for

demonstration purposes), and so on. **Figure 32**, **Figure 33**, and **Figure 34** show the sustained aggregate bandwidth of each of these systems during a limit-case simulation and **Table 2**, **Table 3**, and **Table 4** show the utilization of both the request link bus and response link bus in the format of (*RequestUtilization*%, *ResponseUtilization*%). The results also provide evidence of **Equations 4 & 5** accurately predicting optimal configurations.

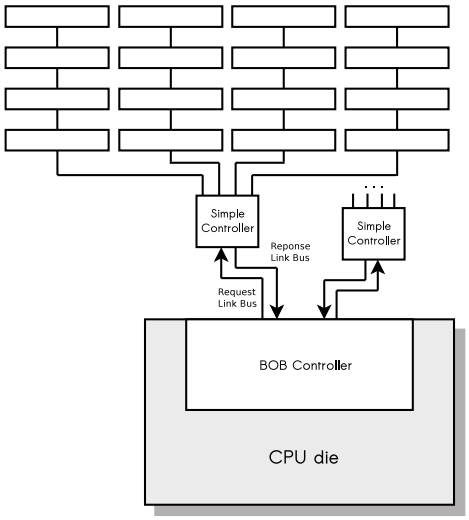
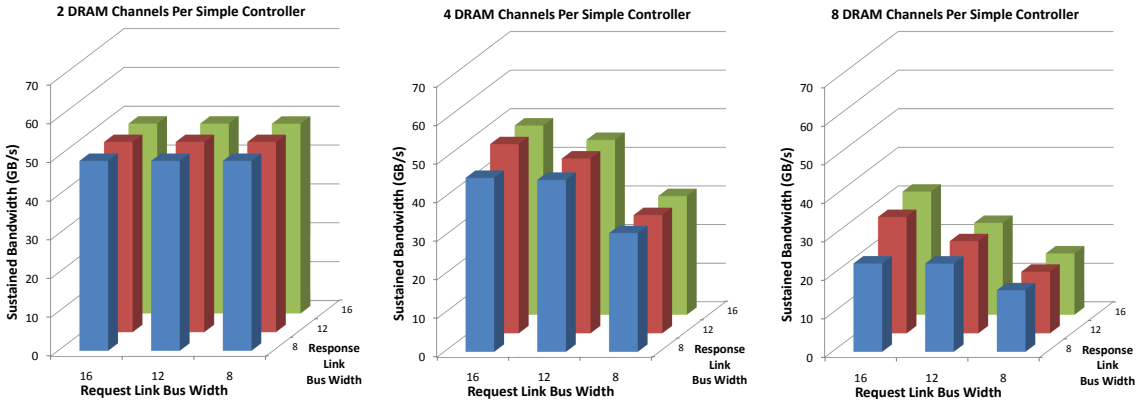


Figure 31: An example of a 4-to-1 multi-channel configuration

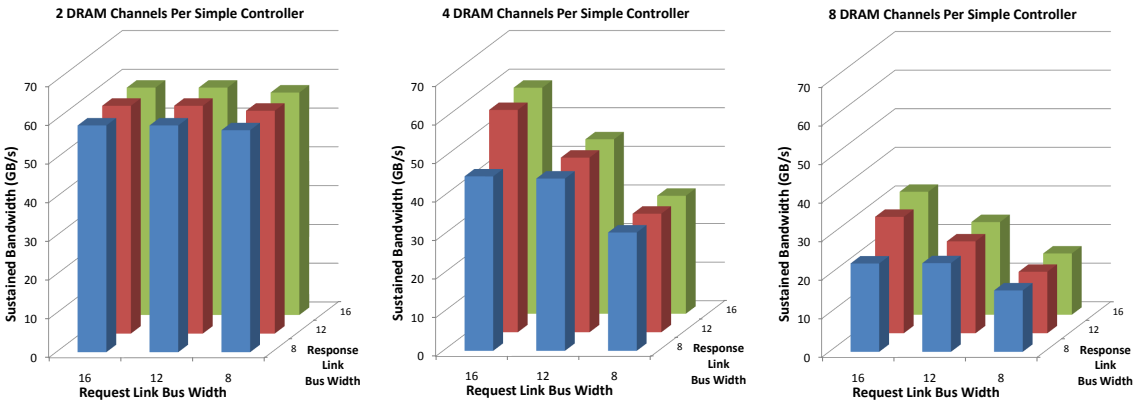


(a) 2-to-1 Ratio - DDR3-1066 (b) 4-to-1 Ratio - DDR3-1066 (c) 8-to-1 Ratio - DDR3-1066

Figure 32: Sustained bandwidth of 8 DRAM channels of DDR3-1066 with varying degrees of *multi-channel* utilization

DDR3-1066 - 2-to-1			
	Response Width		
Request Width	**8**	**12**	**16**
8	47.03%, 76.72%	47.03%, 51.15%	47.03%, 38.36%
12	34.17%, 76.95%	34.17%, 51.30%	34.17%, 34.48%
16	23.48%, 76.80%	23.48%, 51.20%	23.48%, 38.40%
DDR3-1066 - 4-to-1			
	Response Width		
Request Width	**8**	**12**	**16**
8	59.45%, 96.25%	85.94%, 93.13%	86.72%, 70.70%
12	42.86%, 96.09%	63.18%, 95.00%	68.65%, 76.84%
16	29.73%, 96.09%	43.59%, 94.79%	47.34%, 76.80%
DDR3-1066 - 8-to-1			
	Response Width		
Request Width	**8**	**12**	**16**
8	61.56%, 99.92%	87.97%, 95.73%	87.97%, 71.95%
12	44.74%, 99.92%	66.72%, 99.95%	84.17%, 94.77%
16	30.55%, 99.92%	45.47%, 99.99%	61.64%, 99.99%

Table 2: The utilization of the request link bus and response link bus (*Request Utilization*%, *Response Utilization*%) when eight channels of DDR3-1066 are used in systems with various multi-channel utilization

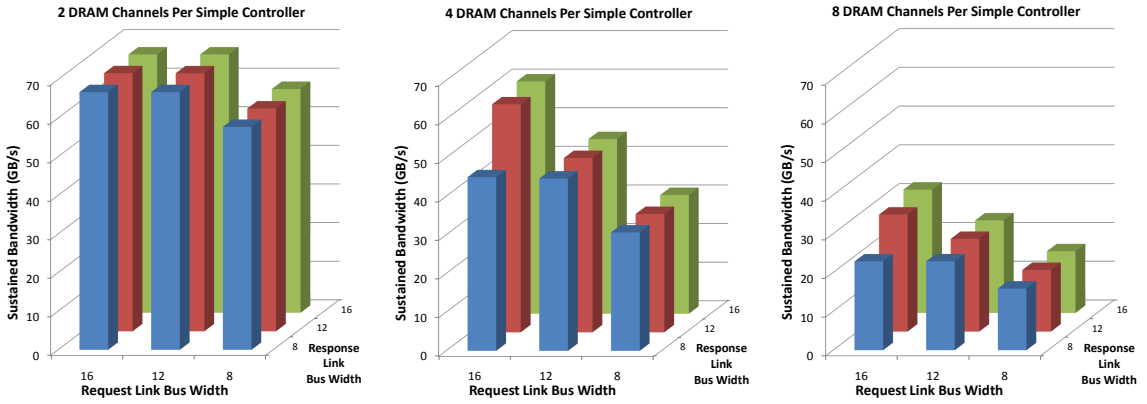


(a) 2-to-1 Ratio - DDR3-1333 (b) 4-to-1 Ratio - DDR3-1333 (c) 8-to-1 Ratio - DDR3-1333

Figure 33: Sustained bandwidth of 8 DRAM channels of DDR3-1333 with varying degrees of *multi-channel* utilization

DDR3-1333 - 2-to-1			
	Response Width		
Request Width	**8**	**12**	**16**
8	55.08%, 89.92%	56.34%, 61.22%	56.34%, 45.91%
12	40.21%, 90.25%	41.04%, 61.56%	41.04%, 46.20%
16	27.62%, 90.07%	28.20%, 61.43%	28.20%, 46.07%
DDR3-1333 - 4-to-1			
	Response Width		
Request Width	**8**	**12**	**16**
8	59.14%, 96.09%	85.86%, 93.49%	86.80%, 71.05%
12	42.97%, 96.48%	63.39%, 94.69%	80.83%, 90.27%
16	29.38%, 96.02%	43.48%, 94.69%	56.56%, 91.76%
DDR3-1333 - 8-to-1			
	Response Width		
Request Width	**8**	**12**	**16**
8	61.25%, 99.98%	87.89%, 96.51%	88.00%, 71.97%
12	44.69%, 99.99%	66.48%, 99.98%	84.17%, 95.00%
16	30.45%, 99.99%	46.60%, 99.99%	61.45%, 99.99%

Table 3: The utilization of the request link bus and response link bus (*Request Utilization*%, *Response Utilization*%) when eight channels of DDR3-1333 are used in systems with various multi-channel utilization



(a) 2-to-1 Ratio - DDR3-1600 (b) 4-to-1 Ratio - DDR3-1600 (c) 8-to-1 Ratio - DDR3-1600

Figure 34: Sustained bandwidth of 8 DRAM channels of DDR3-1600 with varying degrees of *multi-channel* utilization

DDR3-1600 - 2-to-1			
	Response Width		
Request Width	**8**	**12**	**16**
8	55.47%, 90.78%	64.30%, 69.79%	64.30%, 52.34%
12	40.36%, 90.63%	46.77%, 69.95%	46.80%, 52.50%
16	27.99%, 90.88%	32.30%, 69.90%	32.30%, 52.42%
DDR3-1600 - 4-to-1			
	Response Width		
Request Width	**8**	**12**	**16**
8	59.14%, 96.09%	86.02%, 93.39%	86.48%, 70.66%
12	34.04%, 96.06%	63.07%, 94.67%	82.81%, 92.63%
16	29.65%, 96.14%	43.67%, 94.60%	57.99%, 94.48%
DDR3-1600 - 8-to-1			
	Response Width		
Request Width	**8**	**12**	**16**
8	60.95%, 99.98%	87.94%, 96.17%	88.00%, 71.97%
12	44.46%, 99.99%	67.05%, 99.99%	84.18%, 95.05%
16	30.62%, 99.99%	45.99%, 99.99%	61.23%, 99.99%

Table 4: The utilization of the request link bus and response link bus (*Request Utilization*%, *Response Utilization*%) when eight channels of DDR3-1600 are used in systems with various multi-channel utilization

When two DRAM channels share a link bus and simple controller (multi-channel degree of two), there is little impact on performance relative to a system which does not employ this optimization. Slower speeds of DRAM can easily reach peak per-

formance, demonstrating that this optimization can significantly reduce system costs without any degradation of performance. This configuration uses half the number of simple controllers and CPU pins relative to a system without the multi-channel optimization, making it attractive to system and chip manufacturers who must consider these costs. Systems that use DDR3-1600 have their performance reduced by approximately 14% with the narrowest response link bus simulated (**Figure 34(a)**). This is a result of the read-write ratio used in limit-case simulations weighting the request stream towards a greater number of reads. When using **Equation 5** to determine what response link bus bandwidth is necessary for DDR3-1600 in this scenario, a value of 13.4 GB/s is found, which is greater than what configurations with 8 bit lanes of response width can provide (12.8 GB/s), thereby reducing performance. The simulations show that once the response link bus has been widened to 12 bit lanes, the available bandwidth of the response link bus exceeds that of the value computed by **Equation 5** and therefore peak possible performance is achieved.

Once four DRAM channels share the same simple controller and link bus (multi-channel degree of four as pictured in **Figure 31**), the lack of available link bus bandwidth becomes a hindrance to system performance. Slower speed grades of DRAM are capable of achieving peak performance in configurations with wider link buses, but channels of DDR3-1600 are unable to reach maximum attainable efficiency (70%) in any configuration simulated. When using **Equations 4 & 5** to compute the requirements of four DRAM channels of DDR3-1600, a link bus needs approximately 26.8 GB/s response bandwidth and 16.4 GB/s request bandwidth in order to operate at maximum efficiency. This is not provided even by even widest link buses tested,

which provide only 25.6 GB/s. When simulating this system, each channel of DDR3-1600 is only capable of achieving approximately 60% efficiency, which is less than the maximum efficiency for that speed grade.

As can be seen in **Table 2**, **Table 3**, and **Table 4**, a multi-channel degree of 8 leads to the response link bus that is utilized over 99% of the time in a majority of the configurations simulated, thereby causing a significant drop in performance. While such a setup will make the design of the simple controller significantly more expensive and does not perform nearly as well as lesser degrees of multi-channel utilization, the pin-out of the CPU would decrease dramatically while still achieving both performance and capacity gains relative to a standard DDR memory system.

Through simulation of numerous configurations, **Equations 4 & 5** have been shown to provide an accurate way of determining the optimal link bus bandwidth necessary to reach peak DRAM efficiency given a certain set of parameters and behaviors such as DRAM speed, packet overhead, and read-write ratio. When the request link bus bandwidth is less than computed values, requests are incapable of reaching the DRAM at a rate which allows it to operate at maximum attainable efficiency. When the response link bus is less than the computed valued, responses are incapable of evacuating the read return queue fast enough to prevent it from reaching maximum capacity and forcing the DRAM to stall. These equations provide system manufacturers with the ability to determine the proper link bus configuration so as to optimally utilize the available resources.

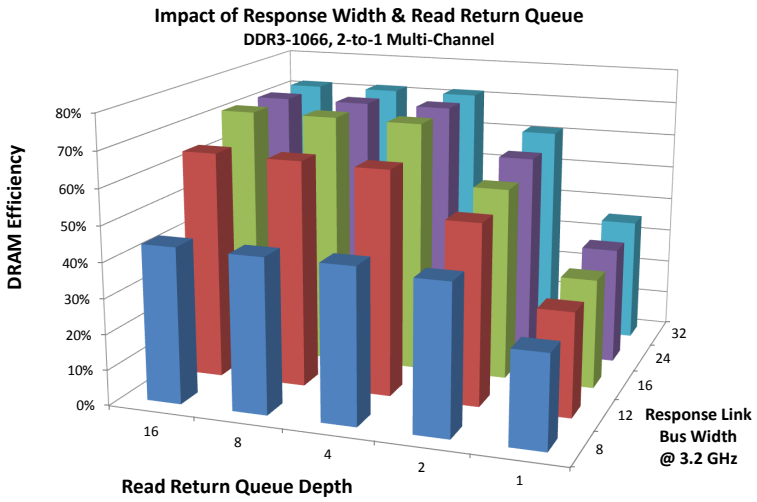
The implementation of the multi-channel optimization has implications in the design decisions of other parts of the system as well. The read return queue was

previously examined in configurations with only a single DRAM channel attached to each link bus. Aspects of this queue must now be explored with various degrees of the multi-channel optimization. When multiple DRAM channels use the same response link bus, the available bandwidth must be shared and will result in different behavior when removing response packets from this queue. As discussed previously (**Figure 30**), while the address determines which channel gets a request packet, a simple round-robin policy determines which read return queue will send a response packet to the response link bus. If a channel's queue is empty, the next channel is selected instead.

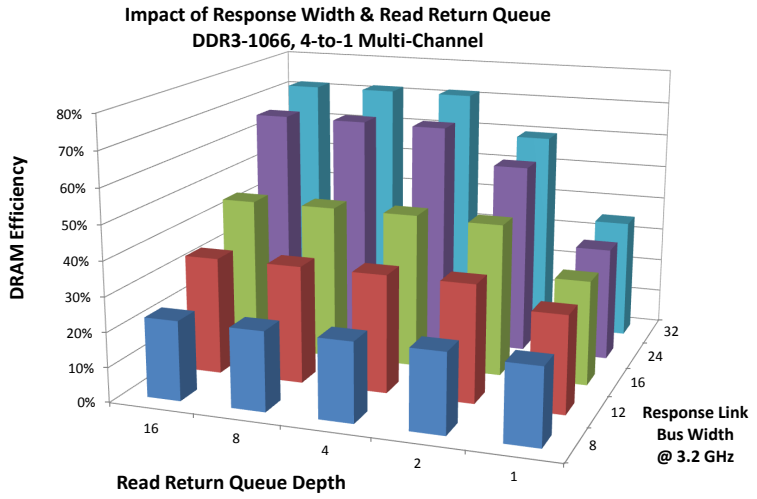
Similar to the examinations of the return queue depth seen in **Figures 19, 20, & 21**, the response link bus width and read return queue depth are varied during limit-case simulations of multi-channel configurations which use various speeds of DRAM. The results can be seen in **Figures 35, 36, & 37**. In systems that do not use the multi-channel optimization, a queue depth of four is required to reach maximum attainable efficiency; this is the case with multi-channel configurations as well. While slower speeds of DRAM do not require as wide of a response bus to reach the maximum attainable efficiency, a queue depth of four is always required. Further increasing the queue depth beyond four provides only marginal gains to DRAM efficiency. At most an increase of 2% in efficiency was seen when increasing the queue depth from four to eight. There are two possible reasons: either the DRAM has already achieved maximum attainable efficiency and can not be increased any further, or the response link bus utilization is so great that increasing it further becomes difficult. A high response link bus utilization is the result of the bus being too narrow to support that

level of multi-channel utilization. In many situations, the response link bus is utilized over 95% of the time and over 99.9% of the time in 8-to-1 configurations. **Figures 35(c), 36(c), & 37(c)** show this; a queue depth of two is sufficient to reach that system's peak possible efficiency since the response link bus is the bottleneck for that configuration.

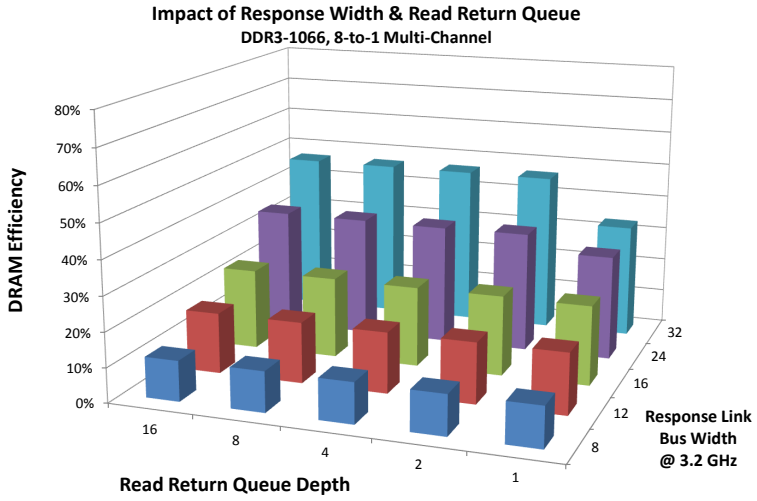
At this point, the conclusion can be made that a queue depth of four for the read return queue appears to represent the best trade off between performance and cost. The read return queue will be explored further during full-system simulations to ensure this is the case when a system handles a real workload. After the above examinations of the multi-channel optimization, another conclusion that can be made is that an 8-to-1 multi-channel configuration is too restrictive and not a viable solution. The DRAM is wholly under utilized, while other parts of the system (such as queues and buses) are over utilized. The complexity of the simple controller in an 8-to-1 configuration is a factor in this conclusion as well as well. These configurations are ruled out in further studies.



(a) DDR3-1066 efficiency in a 2-to-1 multi-channel configuration

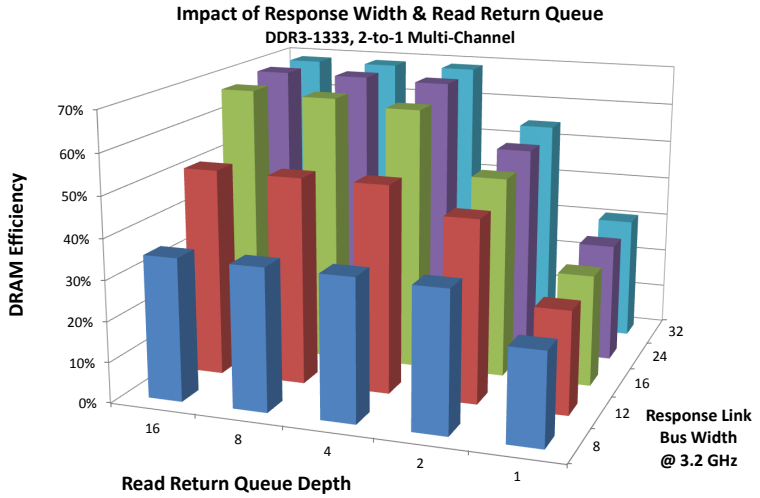


(b) DDR3-1066 efficiency in a 4-to-1 multi-channel configuration

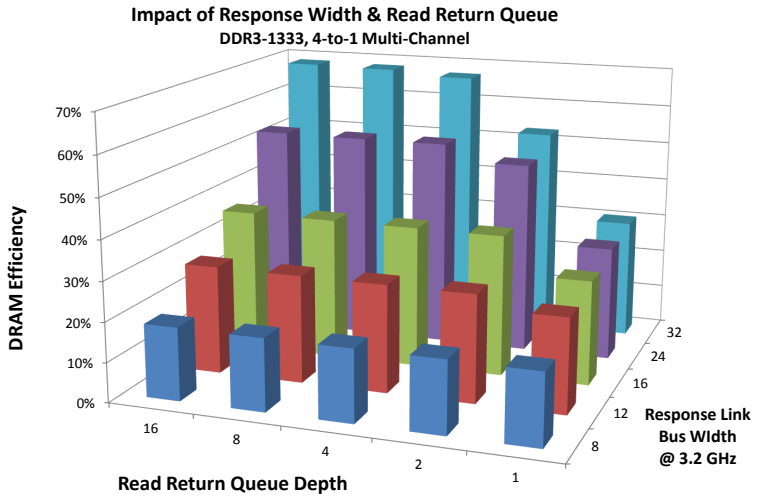


(c) DDR3-1066 efficiency in a 8-to-1 multi-channel configuration

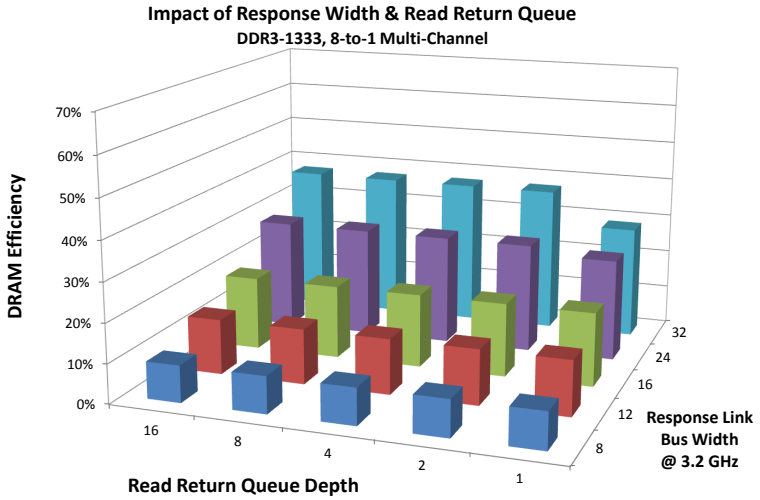
Figure 35: Impact of response link bus width and read return queue on the efficiency of eight channels of DDR3-1066 DRAM in multi-channel configurations



(a) DDR3-1333 efficiency in a 2-to-1 multi-channel configuration

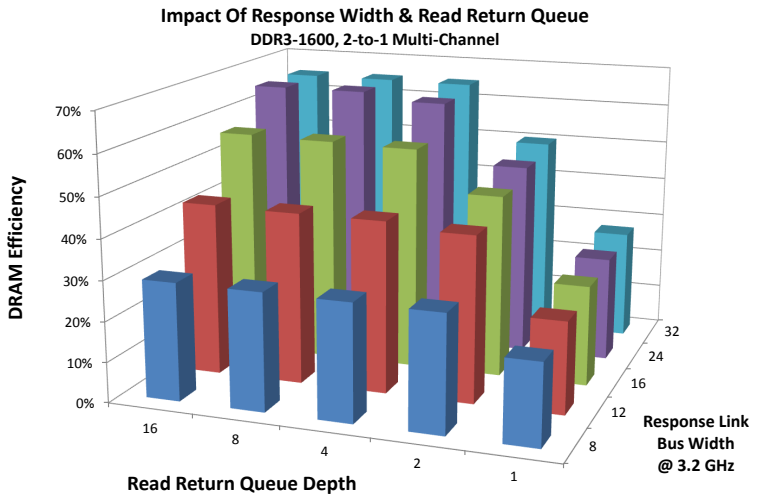


(b) DDR3-1333 efficiency in a 4-to-1 multi-channel configuration

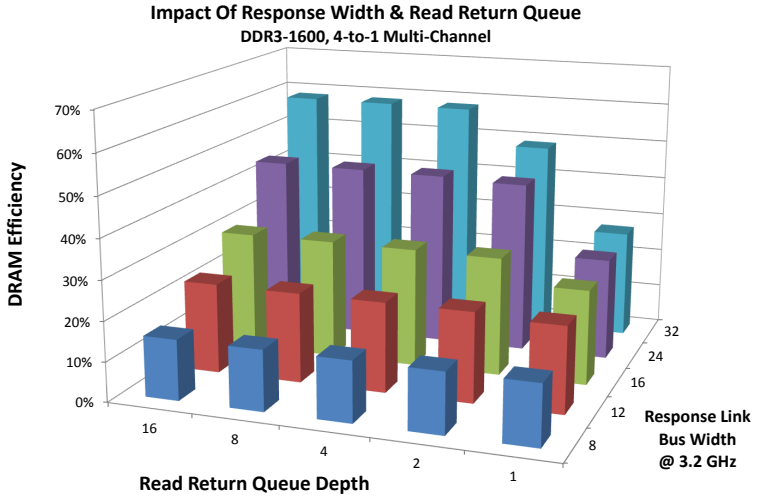


(c) DDR3-1333 efficiency in a 8-to-1 multi-channel configuration

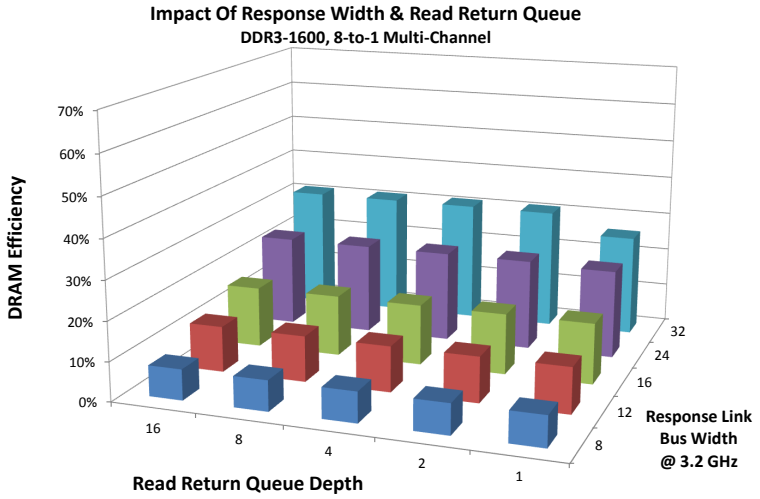
Figure 36: Impact of response link bus width and read return queue on the efficiency of eight channels of DDR3-1333 DRAM in multi-channel configurations



(a) DDR3-1600 efficiency in a 2-to-1 multi-channel configuration



(b) DDR3-1600 efficiency in a 4-to-1 multi-channel configuration



(c) DDR3-1600 efficiency in a 8-to-1 multi-channel configuration

Figure 37: Impact of response link bus width and read return queue on the efficiency of eight channels of DDR3-1600 DRAM in multi-channel configurations

4.1.5 Cost Constrained Simulations

When implementing an actual system, costs such as the required CPU pin-out, power consumption, physical space, and the monetary cost of the DIMMs are all important aspects that need to be considered. So far simulations have not taken these constraints into account, instead aiming for a general overview of the system's behavior by exploring the design space both inside and outside of what might actually be feasible. To ensure that the principles learned from these simulations actually apply to real–world situations, constraints should be placed on various dimensions of the design space.

For example, a constraint could be the total number of CPU pins allotted to communicate with the memory system. The total number of CPU pins is a significant portion of the fabrication cost, so optimal use of the ones available is critical. A fixed number of pins can be configured as link buses in numerous ways, from a few, wide buses to numerous narrow ones. Another example constraint could be a maximum number of DIMMs allowed in a single system, either due to physical space, monetary, or power limitations.

Intel's SMB is used to determine some of the other costs involved with this architecture. In an Intel-based system (**Section 1.2.5**), the SMB dissipates 7 watts while idle and up to 14 watts under load [13]. Within the simulator, these values are used when determining system power. Each simple controller consumes 7 watts of background power and an additional 3.5 watts for each DRAM channel it controls (i.e., a simple controller which controls four DRAM channels will consume 7 watts

background power and 14 watts for the channels it operates, totaling 21 watts under load). The pin-out of the SMB is also used to determine the appropriate number of pins required to control a channel of DRAM. Of the 655 pins on each SMB package, 147 are used to control a single channel of DRAM. Therefore, when determining the pin cost of a simple controller in the simulator, a multiple of this value is used depending on the number of DRAM channels.

Further limit-case simulations are performed with such outside constraints placed on aspects of the BOB system. In these systems, eight DRAM channels, each with four ranks (32 DIMMs making 256 GB total) are allowed, while the CPU has up to 128 pins which can be used for data lanes to comprise various link buses operating at 3.2 GHz. The theoretical peak of this system is 85.333 GB/s (eight channels of DDR3-1333 whose theoretical peak bandwidth is 10.666 GB/s each). Even with these constraints, there are still numerous ways to configure a BOB memory system from numerous channels with narrow link buses with no multi-channel utilization to only a few wider link buses that utilize a high degree of the multi-channel optimization. Each of these types of configurations can be optimized for bandwidth performance, latency, power, or monetary cost. Some of these possibilities (**Table 5**) are simulated, and the results can be seen in **Figure 38**. **Note** : Y-Axis in **Figure 38** is inverted.

Config Name	Request Bus Width	Response Bus Width	DRAM : Simp. Controller	Simp. Controller Pin-Out	Num Simp. Controller	CPU Data Lanes
A	8	8	1:1	147	8	128
B	12	12	2:1	294	4	96
C	16	16	2:1	294	4	128
D	16	16	4:1	588	2	64
E	32	32	4:1	588	2	128
F	4	8	1:1	147	8	96
G	8	12	2:1	294	4	80
H	8	16	2:1	294	4	96
I	8	32	4:1	588	2	80
J	12	16	2:1	294	4	112
K	12	32	4:1	588	2	88
L	16	32	4:1	588	2	96

Table 5: Configuration parameters for various tested systems. Optimal configurations determined in **Figure 38** are highlighted in yellow.

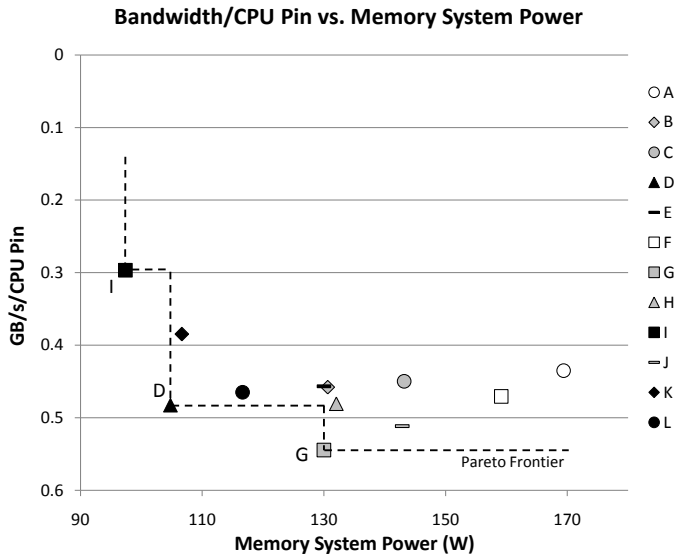


Figure 38: Pareto frontier analysis plot for configurations in **Table 5**. **Note:** Shade of data point corresponds to simple controller complexity

To perform a fair Pareto frontier analysis, relevant costs must be incorporated into the data. In order to do this, the sustained bandwidth is normalized against the total number of CPU pins that are utilized because some configurations do not use all 128

pins which are allotted. The color of each data point in **Figure 38** corresponds to the relative complexity of the simple controller. Black data points are configurations that have a simple controller which requires 588 pins for four DRAM channels; the gray points require 294 pins for two DRAM channels; the white points require 147 for one DRAM channel.

It is clear from **Figure 38** that some configurations of the available resources (DRAM and pins) are more desirable than others. The Pareto frontier analysis dictates that configurations D, G, and I are Pareto equivalent and the most optimal for the parameters tested. Because the simple controller complexity is not accounted for in this analysis, there is still a decision to be made about which configuration is best suited for a particular situation. If raw performance or a less complex simple controller is more desirable, than configuration G is better suited, yet if system power consumption is a concern, configurations D and I are better. This analysis also clearly shows the downside to configurations where a simple controller drives only a single channel of DRAM (white points). In this situation, the power dissipation from having 8 separate simple controllers far exceeds that of the other configurations and yields no benefit in performance.

Configurations D, G, and I represent a range of likely and viable configurations of a BOB memory system. For subsequent examinations of the BOB memory system, these optimally configured systems will be used.

4.1.6 Ports

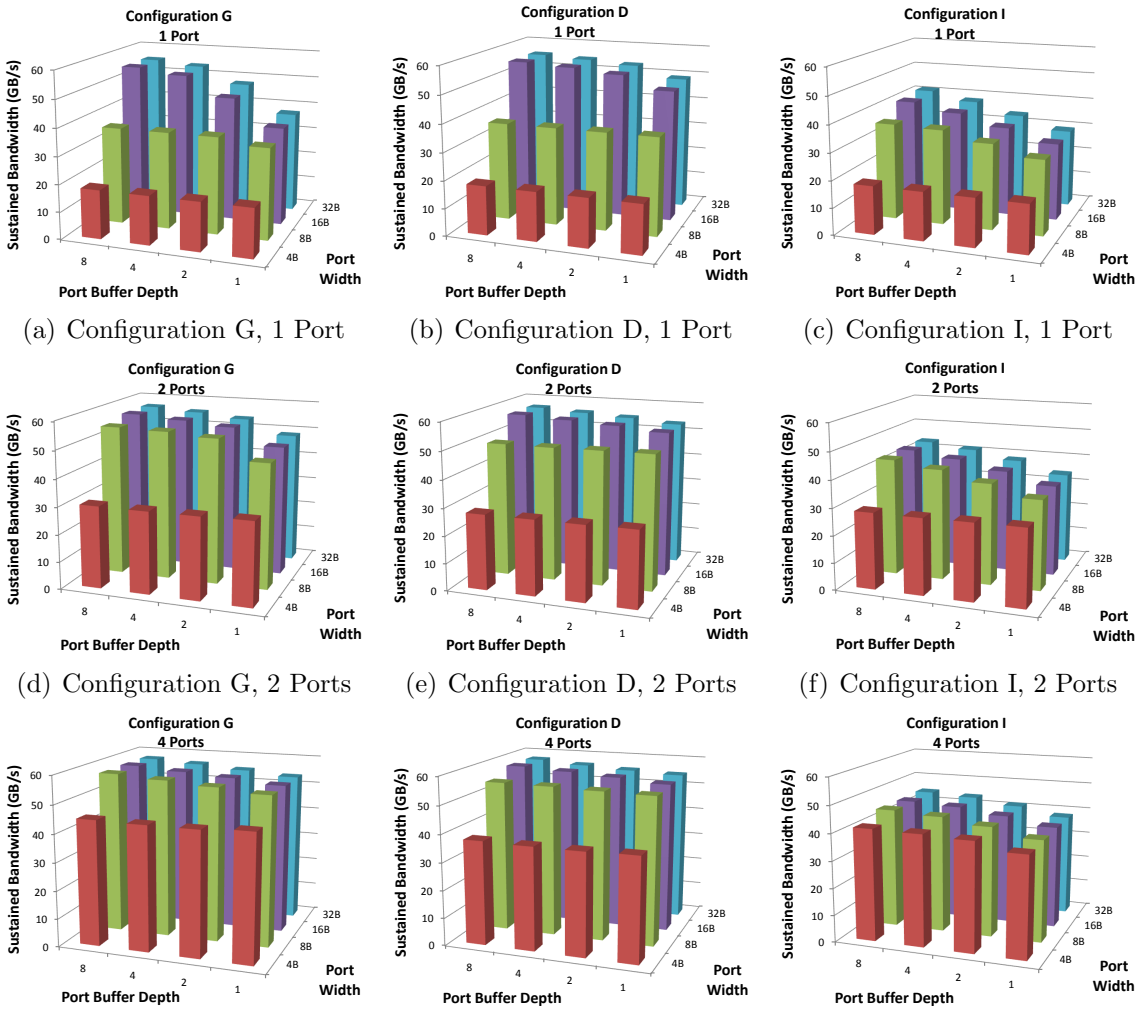
An aspect of the BOB memory system that has not yet been explored are the *ports* in the main BOB controller. These ports are used as the means of communication between the CPU and main BOB controller. Each port is a full–duplex bus which can receive requests and return read data to and from the cache at the same time. With a greater number of CPU cores now operating in parallel on modern processors, the likelihood of simultaneous issuing of memory requests is growing as well, dictating the need for concurrent issuing of requests and responses to and from the memory system. With multiple ports on the main BOB controller, this becomes possible. As ports are an integral part of a BOB system, details about how they are organized must be investigated. Such details include the width and speed of each port, total number of ports, and how requests are added to each port.

Since the main BOB controller and corresponding ports reside on the CPU die, the frequency that these ports are operated is dictated by the CPU clock. During limit-case simulations this is 3.2 GHz. The width of a port determines the number of bytes that can be moved in a single CPU cycle. This can be the result of a single cycle on a wide bus or a more narrow bus utilizing double-data rate transfer. Since the bus is full–duplex, this number of bytes may move in both directions on each cycle.

Each port has corresponding buffers that store request and response packets. Since each bus is full-duplex, there is a buffer for both input and output. Packets are stored in the input buffer while waiting for arbitration on the link buses and in the output

buffer while waiting for the port to become available in order to return data back to the cache. These port buffers are searched out of order to find the first possible item that is capable of being issued to the respective link bus. If the request is destined for a link bus that is currently in use (or corresponding Ser–Des buffer is full) or the destined simple controller work queue is at its maximum capacity, then that request will not be removed from the port buffer.

While performing limit-case simulations on the Pareto optimal configurations determined above (configurations D, G, and I from **Table 5**), the number of ports, the width of each port, and the buffer space given to each port are varied and the sustained bandwidth of these systems can be seen in **Figure 39**. In all configurations, when the port is only capable of transferring 4 bytes on each CPU cycle, the performance of the rest of the system is significantly impeded. This is mainly the result of the outgoing port requiring 18 cycles to return a 72-byte data packet to the cache. This inordinate amount of time results in a back-up in the rest of the system; data responses are stalled within the simple controller's read return queue while waiting for the port to become free, which results in stalling the operation of the DRAM. In these cases the output port is being utilized over 99% of the entire lifetime of the simulation. The conclusion can be made that the depth of the port buffer when the bus is so narrow has practically no impact on the performance, as the width of the bus is the overall limiting factor.



(a) Configuration G, 1 Port (b) Configuration D, 1 Port (c) Configuration I, 1 Port

(d) Configuration G, 2 Ports (e) Configuration D, 2 Ports (f) Configuration I, 2 Ports

(g) Configuration G, 4 Port (h) Configuration D, 4 Port (i) Configuration I, 4 Port

Figure 39: The impact the number of ports, port depth, and port width have on configurations G, D, and I

The data in **Figure 39** also shows no significant difference in performance when increasing the port width from 16 bytes to 32 bytes. This is in large part due to a port width of 16 bytes being able to send and receive data faster than other constraints within the system. A port with a width of 16 bytes can return a data packet to the cache in 5 CPU cycles (1.5625 ns) while the time a response packet spends on the link bus is at least 2.8125 ns (in configuration I which has the widest response bus). Therefore, the response packet can be sent out the port before a new one arrives from

the link bus, and the extra bandwidth available when doubling the port to 32 bytes would not improve performance and thus would be a waste of resources.

When increasing the total number or ports, the performance is almost universally improved, especially in configurations of greater constraint, such as with a shallow port buffer or narrow port bus. For example when the port is only 4 bytes wide with a depth of one entry, increasing the number of ports from one to four increases the sustained bandwidth more than 250%. Unfortunately, when increasing the number of ports in this fashion, a comparison of the end result is not entirely fair as the overall buffer space and total width across all ports is not equivalent. As with the previous cost-constrained simulations seen in section 4.1.5, a constraint should be placed on the overall buffer space and total width given to ports, and the organization of these resources should be explored.

When the main BOB controller is allotted 16 entries worth of port buffer storage and a total of 32 bytes of width to be used for a various number of ports, the possible organizations and the resulting bandwidth for each configuration can be seen in **Figure 40**. Again, even with eight total ports operating in parallel, a width of 4 bytes is universally the least ideal as the width is too restrictive when returning data to the CPU. Configuration D sees the greatest reduction in performance when using such a narrow port because the configuration only has two link buses to return responses from each simple controller (compared to configuration G which has four). The impact of the port organization on configuration I is minimal due to the restrictive link buses in that configuration causing the performance bottleneck. The two request link buses of 8 data lanes can not keep the DRAM busy enough to reach the performance

of other configurations. There is little difference in many of the other configurations, but this is likely due to the nature of limit-case simulations. Port organizations will be explored later with full-system simulations when the parallelism of issuing requests and responses will have a greater impact on the overall system performance.

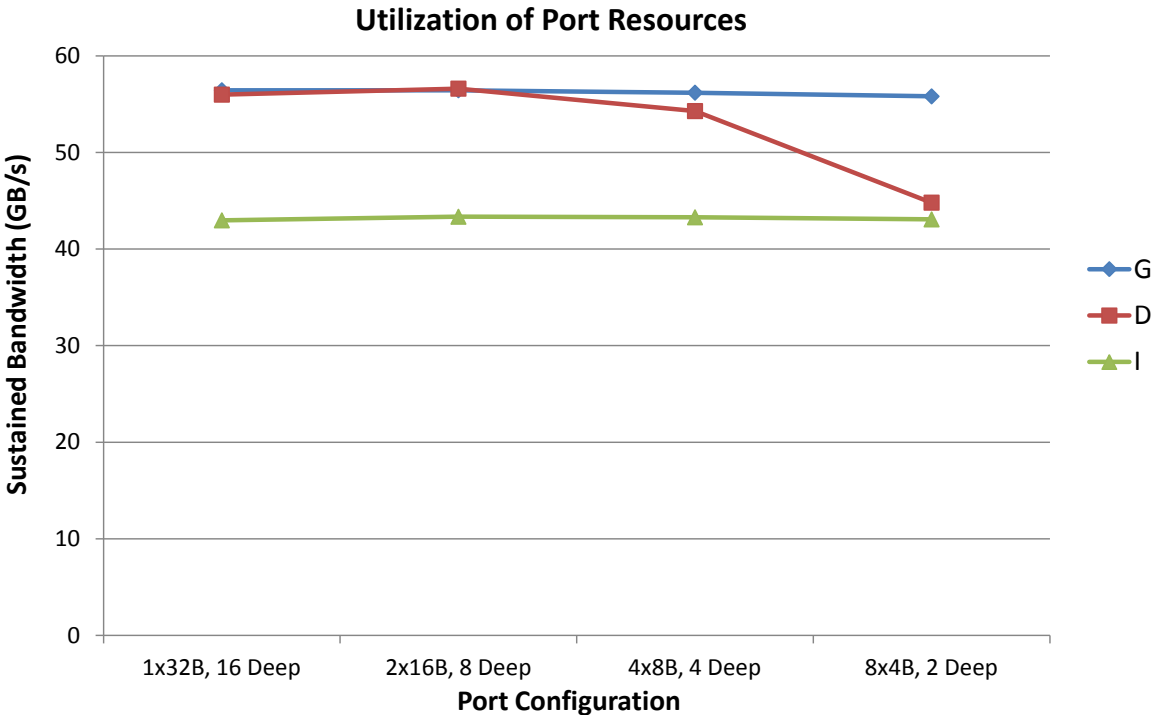


Figure 40: Different port configurations of 32 bytes worth of width and 16 entries worth of depth

Another aspect of the ports that can not be explored under a limit-case simulation is the heuristic used to add new requests. The nature of a limit-case simulation dictates that requests are added to a port as soon as it becomes available. While executing a full-system simulation, the request rate and address stream are dictated by the application. Various heuristics regarding how to add new requests to available ports are explored in the sections below.

4.2 Full System Simulations

While the limit-case simulations help characterize the basic behavior of a BOB memory system, a time-independent stream of random addresses may not always be representative of the workload the system might encounter. With MARSSx86 and numerous multi-threaded benchmarks, the memory system can be observed and analyzed while handling a cycle-accurate and meaningful request stream that includes interaction with the operating system, cache, and virtual memory.

This type of simulation will exercise different aspects of the memory system due to the higher likelihood of conflicts resulting from address stream locality or non-uniform request rates which could flood various parts of the system. These types of interactions are not present in the limit-case workload and are essential to developing a complete picture of a BOB memory system. The memory system's impact on the overall execution time will also be visible with a full system simulation. This is possible because the MARSSx86 CPU model will stall thread execution when waiting for pending memory transactions. The relevant MARSSx86 configuration parameters can be seen in **Table 6**. All figures and tables which display the performance of a full-system simulation use data points which are collected in *epochs*. For these simulations an epoch is one million DRAM cycles.

CPU Speed	3.2 GHz
Num. Cores	8
L1 D/I $	512 KB
L2 Shared $	4 MB
DRAM	256 GB DDR3-1333
OS/Kernel	Ubuntu 9.10 / 2.6.31

Table 6: Configuration parameters for MARSSx86

While BOB configurations D, G, and I were determined to be Pareto optimal configurations during a limit-case simulation, several benchmarks are executed on every configuration listed in **Table 5** to ensure that the conclusions made are applicable to real workloads. **Figures 41** through **44** show the Pareto frontier analysis of all configurations running *mcol*, *mg*, *sp*, and *STREAM*. Each Pareto plot has a slightly different outcome due to different read-write ratios within the request stream which favor some configurations over others. Configurations D and I are consistently along the Pareto frontier, although in different positions on each plot, and other configurations change relative positions in each benchmark. Given this, the overall outcome of the analysis shows a consistent placement of configurations when using performance, pin count, and power consumption as metrics.

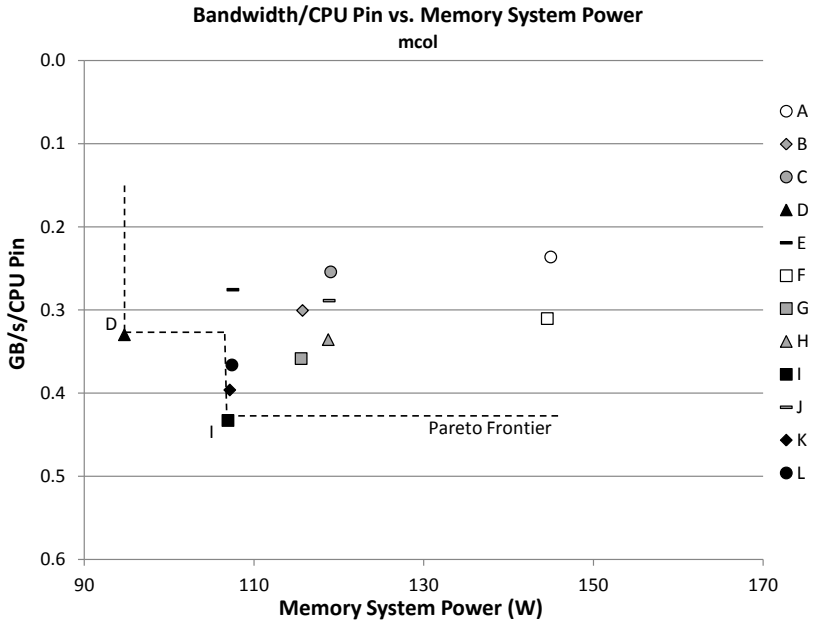


Figure 41: Pareto frontier analysis of all configurations executing *mcol* benchmark

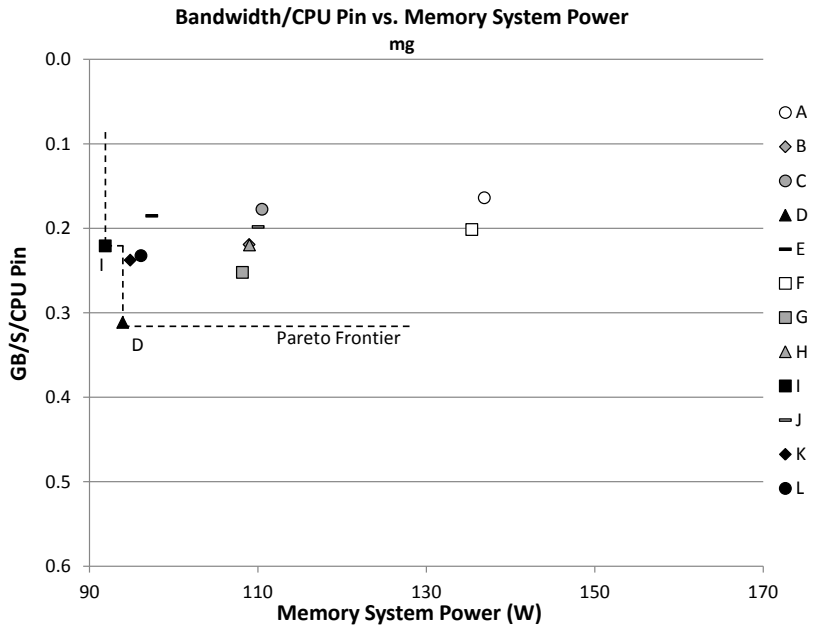


Figure 42: Pareto frontier analysis of all configurations executing *mg* benchmark

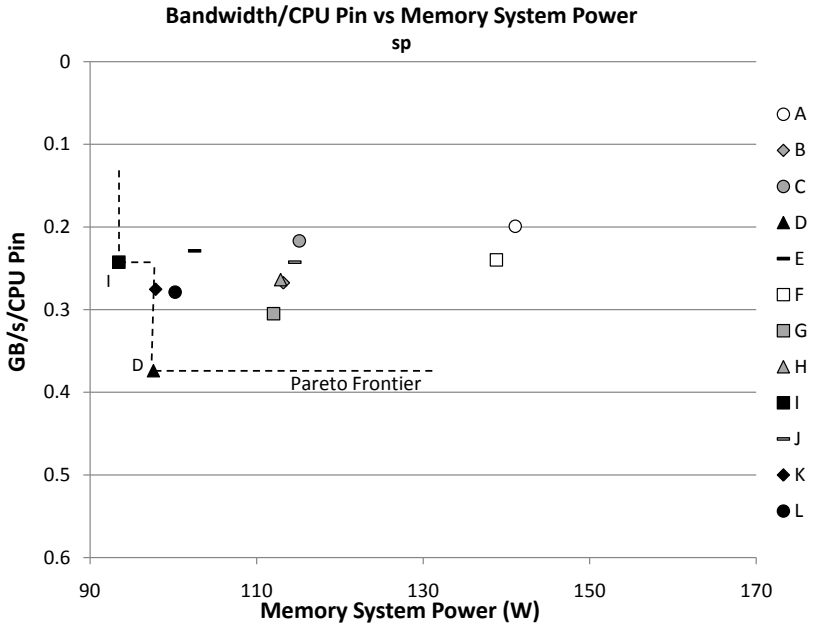


Figure 43: Pareto frontier analysis of all configurations executing *sp* benchmark

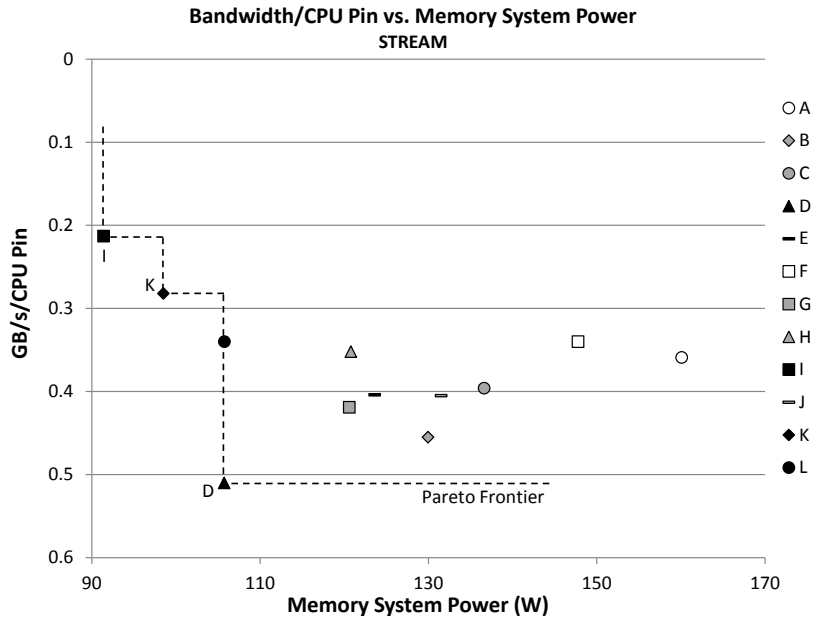


Figure 44: Pareto frontier analysis of all configurations executing *STREAM* benchmark

The Pareto plots confirm the conclusions made during the limit-case simulations and configurations D, G, and I are used for all further studies. While other configurations lie along the Pareto frontier during some benchmarks above, D, G, and I are still chosen because the read-to-write ratio during a limit-case simulation (2/3 reads, 1/3 writes) is generally accepted as the most common case. These configurations also encompass a wide range of possibilities that the available resources can be configured into a memory system, and they will give a broad overview of the behaviors resulting from how these resources are used.

4.2.1 System Performance & Power Trade-offs

Figures 45 through **51** display a number of benchmarks and the impact that each BOB configuration has on the achievable bandwidth, latency, and total execution

time. Each graph displays a "region of interest" within the benchmark; this is the relevant portion of the execution where the core computation is performed. The wide variety of benchmarks results in significantly different behaviors in the memory system. Each BOB configuration achieves the best relative performance during at least one benchmark, showing that the application is the ultimate determining factor in the performance of the memory system.

Table 7 displays the average bandwidth and power consumption of each of the benchmarks, as well as the energy per bit and total energy. These values provide a clearer picture of benefits and drawbacks of each system and are necessary to account for the achieved performance and execution time of each benchmark when comparing configurations. The power consumption accounts for both the DRAM power and the power necessary to operate all simple controllers within that BOB configuration. As previously explained, the power consumption of a single simple controller is based on the Intel SMB [13]; the background power for a single chip is 7 watts, and the operational power for each simple controller core is 3.5 watts. The power consumption of the DRAM is computed using IDD values from the respective device data-sheets and the methodology detailed in Micron technical note TN-47-04 [2].

STREAM (**Figure 51**) and *mcol* (**Figure 47**) generate the greatest average bandwidth among the benchmarks which are executed, yet the behavior of the memory system during these benchmarks is drastically different. While these benchmarks experience significantly different performance from each of the BOB configurations studied, the best performing configuration is different. This is due to the request mix generated during the region of interest; the *STREAM* benchmark generates a request

stream of approximately 46% reads and 54% writes while the *mcol* benchmark issues 98.9% reads.

The relatively balanced request ratio of *STREAM* favors the parallelism and relatively balanced link-bus widths of configuration G, whose execution time is 2.9% less than D and 49% less than I. Configuration I's performance is significantly worse due to the inability of the request link bus to provide the DRAM channels with requests at a sufficient rate. The request link buses encounter periods of over 95% utilization. This issue also results in a higher latency, as incoming requests spend more time waiting in a port's input buffer for arbitration onto the link bus. Conversely, during *mcol*, configuration I performs significantly better than both configurations D and G; this is a result of the wide response link bus in configuration I, which can easily handle the inordinate amount of read requests during this benchmark. The execution time when using configuration I is 15.6% less than G and 36.6% less than D.

While the instantaneous power dissipation provides some insight into the characteristics of each BOB configuration, incorporating the execution time and achieved performance will generate a complete view of relative benefits and drawbacks of each configuration. For example, the instantaneous power of configuration G is 14% greater than configuration D, yet the increased performance and reduced execution time results in only 10.8% more energy actually used. Conversely, the instantaneous power dissipation of configuration I is the least of the configurations, but its increased execution time and poor performance results in over 64% more total energy consumption over the lifetime of the execution. This is also the case during *mcol*; configuration D has the smallest instantaneous power dissipation, but its increased execution time

leads to the greatest total energy consumption.

For benchmarks where the memory system is largely idle (i.e., *fluidanimate* (**Figure 46**), *facesim* (**Figure 45**), and *SandiaGUPS* (**Figure 49**)), the average bandwidth achieved by each configuration and therefore the total execution time as well, are relatively similar; therefore, comparing performance provides little insight. When the memory system is not heavily utilized, the power dissipation and energy play a much larger factor in determining the best configuration. For example, the execution time of *fluidanimate* differs by less than 6% across all systems, yet the energy per bit consumed within configuration G and is 16% greater than D (which has the least). Even with the shortest execution time, configuration G still consumes the most energy. The increased energy consumption is a result of a greater number of simple controllers – four, whereas configuration D and I only have two. A greater number of simple controllers is a benefit during some benchmarks and can increased performance, but when the memory system is mostly idle, the increased power consumption becomes a detrimental factor.

In conclusion, the full system simulations have provided insights into behaviors which were not apparent during the limit-case simulations, such as the impact that the performance and resulting execution time has on the energy consumption of the system. The similarities between D and G's performance in a majority of the cases also makes it clear that significantly different configurations can achieve similar performance and that outside constraints and costs involved with the implementation of the system can still be considered without sacrificing performance. As predicted by the limit-case simulations above, configuration I only performs well in situations of

inordinate numbers of reads and typically is limited by the lack of request link bus bandwidth. With a general idea of how these systems behave and perform under a full system simulation, specific characteristics and features can now be explored.

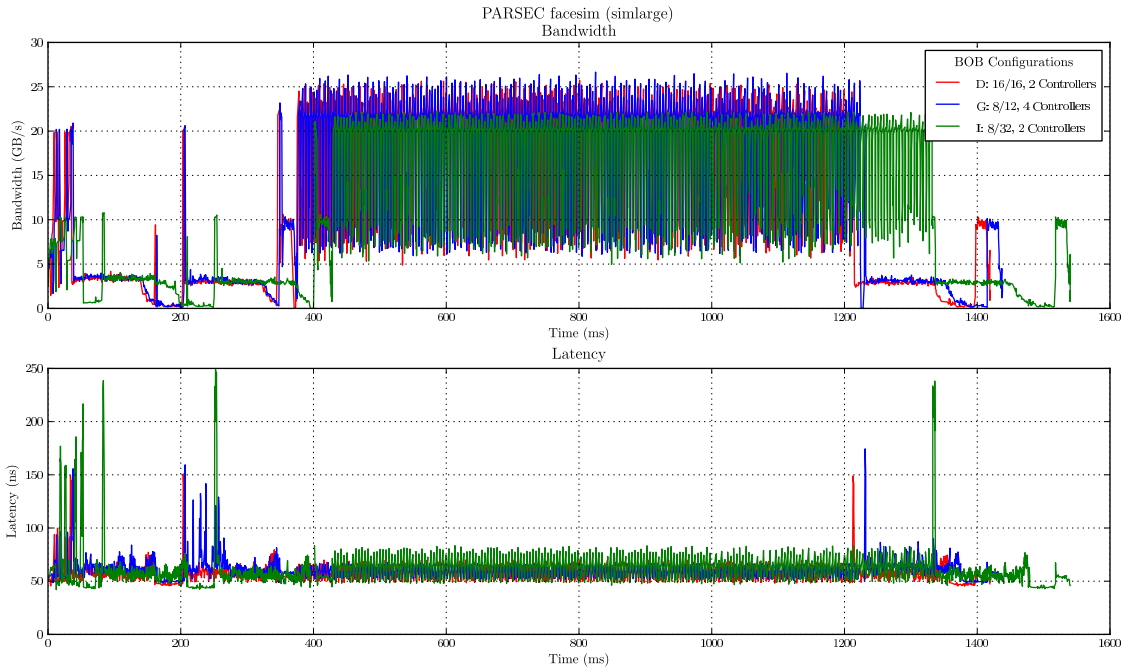


Figure 45: Full system simulations running the *facesim* benchmark

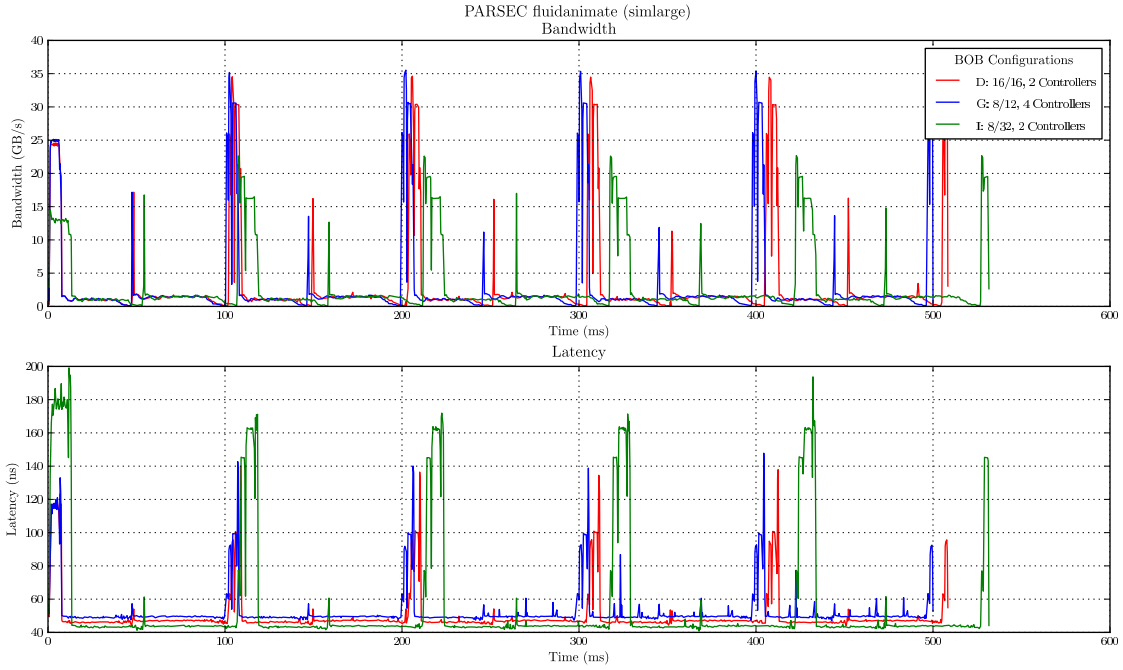


Figure 46: Full system simulations running the *fluidanimate* benchmark from the PARSEC benchmark suite

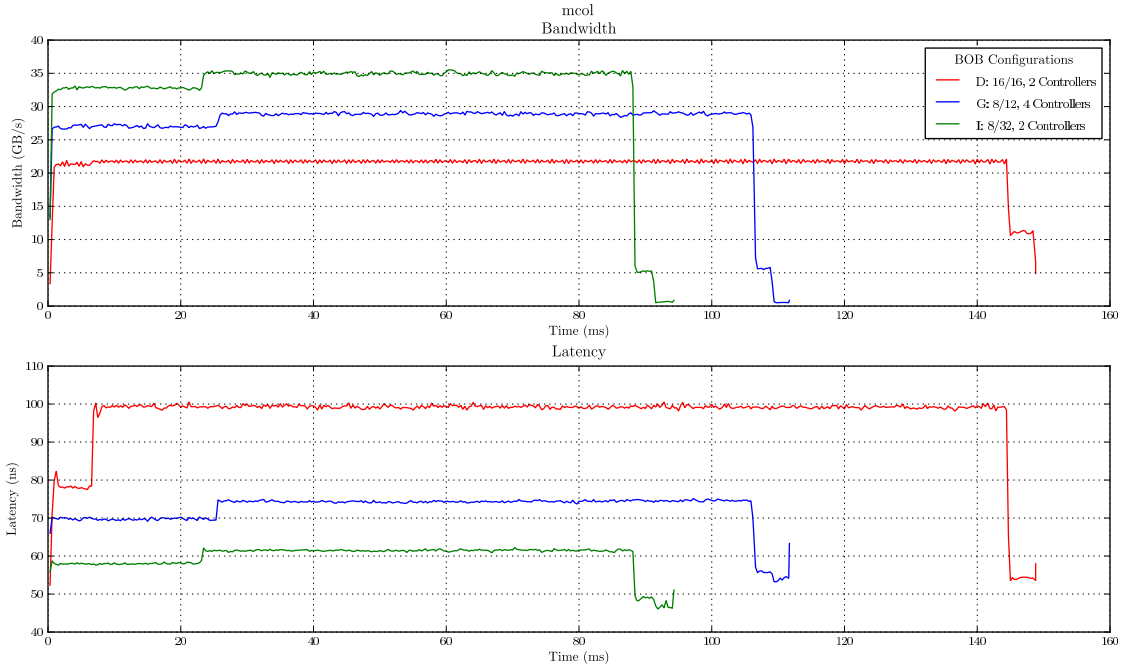


Figure 47: Full system simulations running the *mcol* benchmark

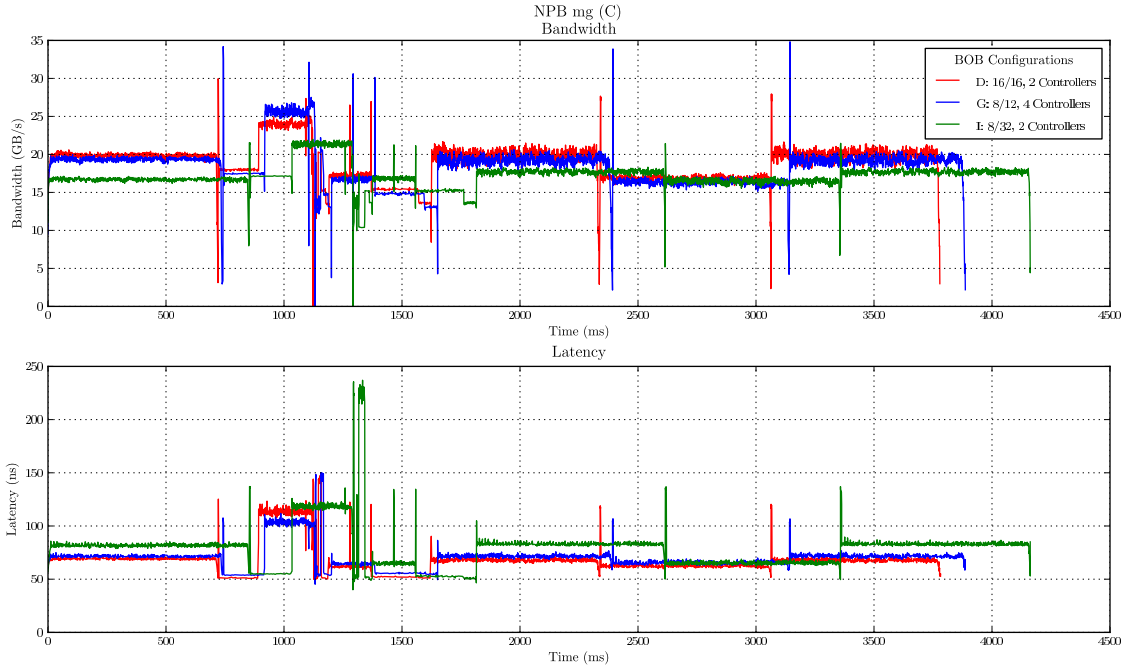


Figure 48: Full system simulations running the *mg* benchmark from the NAS parallel benchmark suite

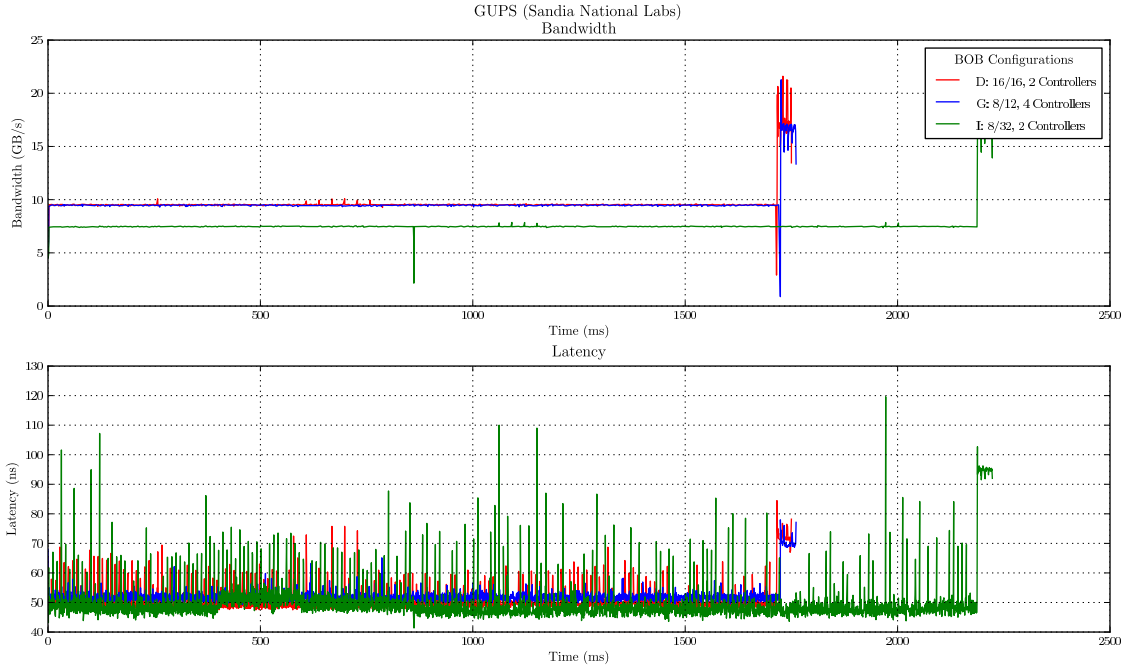


Figure 49: Full system simulations running the *SandiaGUPS* benchmark

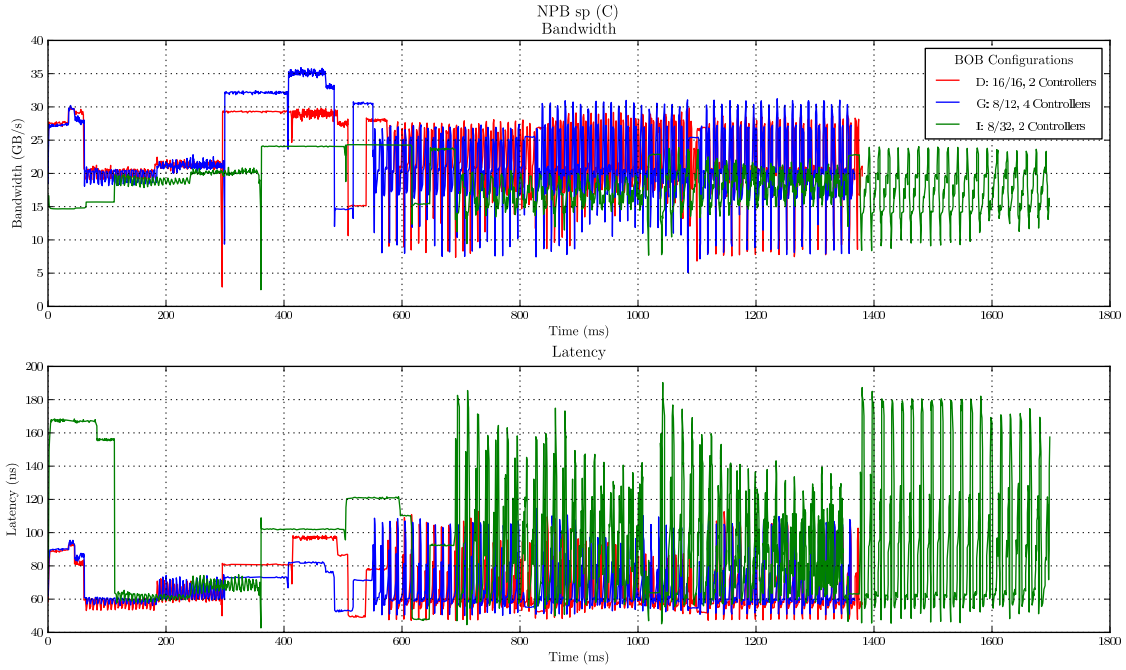


Figure 50: Full system simulations running the *sp* benchmark from the NAS parallel benchmark suite

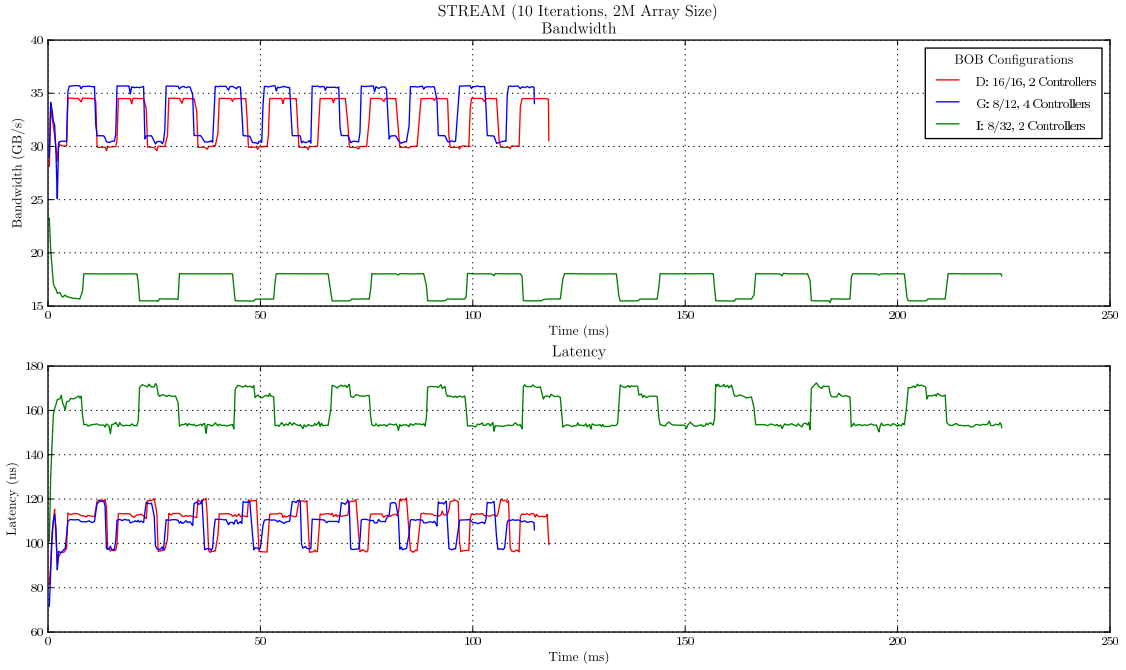


Figure 51: Full system simulations running the *STREAM* benchmark

facesim					
Config	Power (W)	Avg BW (GB/s)	W per GB/s	Energy (J)	Time (ms)
D	87.62	13.02	**6.72**	**124.33**	1419
G	101.53	12.93	**7.85**	**146.0**	1438
I	86.72	12.04	**7.20**	**133.54**	1540
fluidanimate					
Config	Power (W)	Avg BW (GB/s)	W per GB/s	Energy (J)	Time (ms)
D	78.5	3.06	**25.65**	**40**	508.4
G	92.5	3.11	**29.74**	**46.3**	500
I	78.4	2.92	**26.84**	**41.7**	531.6
mcol					
Config	Power (W)	Avg BW (GB/s)	W per GB/s	Energy (J)	Time (ms)
D	94.95	21.33	**4.45**	**14.1**	148.8
G	114.16	27.13	**4.20**	**12.8**	111.7
I	104.77	32.34	**3.23**	**9.9**	94.3
mg					
Config	Power (W)	Avg BW (GB/s)	W per GB/s	Energy (J)	Time (ms)
D	93.16	18.87	**4.93**	**352.5**	3782
G	101.83	18.35	**5.55**	**395.8**	3887
I	92.97	17.13	**5.43**	**377.6**	4162
Sandia GUPS					
Config	Power (W)	Avg BW (GB/s)	W per GB/s	Energy (J)	Time (ms)
D	84.76	9.67	**8.76**	**148.3**	1750
G	98.70	9.61	**10.27**	**173.8**	1761
I	82.84	7.62	**10.87**	**184.2**	2223
sp					
Config	Power (W)	Avg BW (GB/s)	W per GB/s	Energy (J)	Time (ms)
D	97.02	23.27	**4.17**	**133.89**	1380
G	111.17	23.43	**4.74**	**151.97**	1367
I	92.97	18.86	**4.93**	**157.86**	1698
STREAM					
Config	Power (W)	Avg BW (GB/s)	W per GB/s	Energy (J)	Time (ms)
D	105.7	32.61	**3.24**	**12.5**	117.9
G	120.5	33.54	**3.59**	**13.8**	114.5
I	91.44	17.10	**5.34**	**20.5**	224.5

Table 7: Power consumption of each configuration running the aforementioned benchmarks normalized against their average performance

4.2.2 Latency Analysis

During limit-case simulations, requests are added to the memory system as soon as resources permit (i.e., as fast as possible). This results in the system quickly reaching a steady-state where queues reach their maximum capacity without the opportunity of emptying. In such a situation, the latency of a request provides no insight into how the system might behave during typical program execution. While performing a full system simulation, the frequency of requests will vary depending on the current point of the application, as well as system level events like operating system intervention. This will grant a clearer picture of how each BOB configuration will impact the latency of requests.

The bottlenecks that cause poor performance in a configuration can be easily identified when the latency of a request is separated into the various components described earlier (i.e., time spent in command queue or port buffer, access time). For example, the latency components of *STREAM* in **Figure 52** clearly shows that the request link bus is the limiting factor in configuration I. The majority of a request's latency is from stalling in the port input buffer while waiting for arbitration on the request link bus. Consequently, latencies within the DRAM, work queue, and read return queue are far less than the other configurations simply because new requests are unable to reach the simple controllers fast enough.

Another observation is that while the latency seen by configurations D and G are within 3% of each other, the individual latency components contribute different proportions to the overall latency, thereby uncovering relative bottlenecks. For instance,

in configuration D, the response link bus latency is less due to a wider response link bus, yet with fewer of these buses (2 compared to 4 in configuration G) the time spent waiting in the return queue awaiting arbitration is longer. As stated previously, the DRAM access time is relatively similar between all configurations. Any variation seen in this latency component is due to increased DRAM activity which is confirmed by configuration G producing the greatest average bandwidth and having the highest DRAM access time.

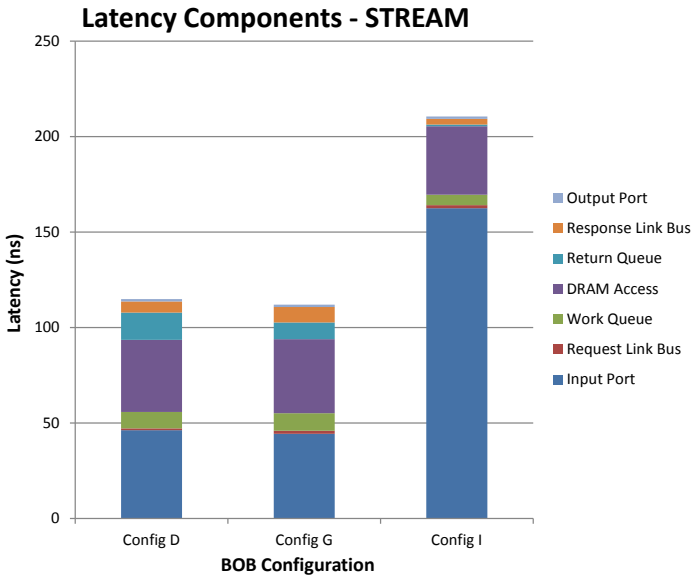


Figure 52: Latency components of read requests during *STREAM* benchmark in **Figure 51**

The *mcol* benchmark (**Figure 47**) has an atypical request stream which is composed of approximately 98% read requests. Such an extreme situation will stress different parts of the system compared to normal operation. The latency components for each configuration can be seen in **Figure 53**; this clearly shows the read-dominated request stream stressing both the response link bus and read return queue. Read requests to configuration D spend the most time stalled in the read return queue

awaiting arbitration onto the response link bus. This is because configuration D does not have the same degree of parallelism as configuration G or the bandwidth of configuration I, both of which perform better. Even though configuration I has the smallest request latency, the poor request link bus bandwidth does cause certain latency components to be greater than the other configurations; while insignificant to the total, requests to configuration I spend more than twice as much time in the port buffers than the other configurations

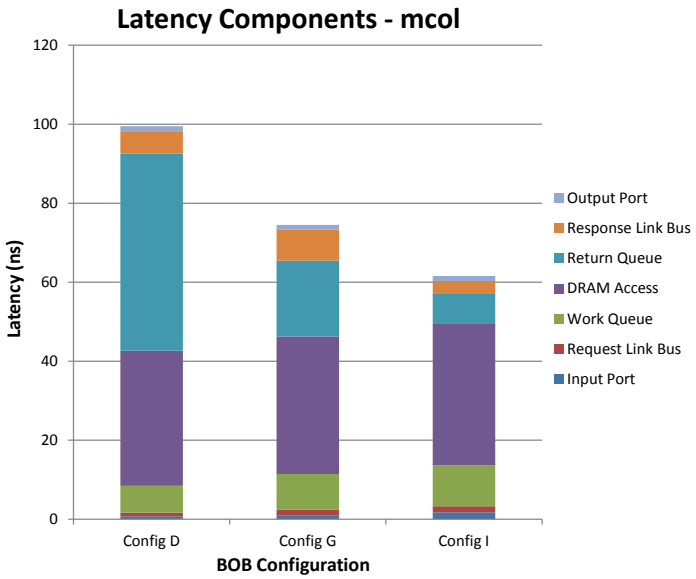


Figure 53: Latency components of read requests during *mcol* benchmark

Figures 54 through **60** display the latency components and number of requests sent to each channel during the benchmarks above (**Figures 45** through **51**). These uncover another important behavior which would not have been visible during a limit-case simulation – the impact that the multi-channel optimization has on request latency to DRAM channels that share resources such as the link bus or SerDes buffer. When a group of DRAM channels share the link bus and SerDes buffer, an uneven dis-

tribution of requests across channels will not only negatively impact the performance of the system as a whole but will also hurt the performance of the lightly-loaded channels individually. This is a result of the lightly-loaded channels being forced to wait for the shared resources to become free. The degree of multi-channel utilization is the determining factor in how many DRAM channels are negatively impacted by this behavior. With a multi-channel degree of two (two DRAM channels for each simple controller and link bus), a heavily utilized DRAM channel can only impact one other channel. With a multi-channel degree of four and eight, the number of DRAM channels impacted by one is increased to 3 and 7, respectively.

Configuration G (multi-channel degree of two) has greater link bus concurrency compared to configurations D and I. This behavior occurs with this configuration, but the available concurrency causes it to be less pronounced than configurations D and I, because requests have a greater opportunity of being issued and will not be stalled in the port input buffer. For example, while configuration G is executing benchmarks like *fluidanimate* and *sp*, there are clear examples of DRAM channels sharing resources having similarly poor performance relative to the others. DRAM channels 0 & 1 experience a 50% higher latency relative to the others during *fluidanimate*, and channels 6 & 7 experience a 37% higher latency during the NAS benchmark *sp*. In the other benchmarks this does not occur, because the link buses are capable of transmitting requests and responses at a rate that does not stall packets and does not cause a back-up.

The higher degree of multi-channel utilization in configurations D and I causes this phenomenon to become more visible as a greater number of DRAM channels have the

chance to be effected by a single, heavily-utilized channel. Examples of this can be easily seen in *facesim* (**Figure 54**), *fluidanimate* (**Figure 55**), *sp* (**Figure 59**), and *STREAM* (**Figure 60**). In these situations, half of the DRAM channels in the system experience significantly higher latency than others. The relatively narrow request link bus in configuration I causes this phenomenon to be worse than configuration D, and in some extreme cases (*fluidanimate* and *STREAM*), some channels have double the latency of the other channels.

It is important to note that, during these situations of uneven request latency, the request stream is composed mainly of write requests. Write requests packets, which are 72 bytes, take significantly longer to serially transmit on a link bus relative to read request packets, which are 8 bytes. Therefore, when the memory system experiences a greater number of writes than reads, read requests are forced to stall within a port input buffer while write requests occupy the request link bus. This also shows the importance of how requests are added to the main BOB controller's ports. For these simulations, requests were simply added to the first available port, by index. Therefore, the underlying cause of this behavior is the result of two factors: an imbalance in requests to a particular channel (mainly writes) and requests being stalled in an input port while awaiting arbitration on the link bus. This can also be indirectly confirmed during *mcol*, which is 99% reads and has an even latency (**Figure 56**).

The latency components for *Sandia GUPS* (**Figure 58**) are radically different from all the other benchmarks. A large work queue latency indicates that there are significant conflicts at the DRAM level, and requests are stalled within the work queue

while waiting for a rank, bank, row, or column to become available. This is typically the sole result of the request stream and can only be mitigated through faster DRAM parts or greater rank or bank parallelism within that DRAM channel.

In conclusion, the request latencies during full system simulations show how the request stream and various bottlenecks in different parts of the system will impact overall latency. For example, insufficient request link bus parallelism or bandwidth causes requests to stall within the port buffer, which increases latency while requests await arbitration. This can occur on the response path as well where requests stall within the read return queue while awaiting arbitration onto the response link bus, thus increasing latency. Another important issue which has arisen through full system simulations is the negative impact that the multi-channel optimization can have on request latencies. If a particular channel receives an inordinate number of requests, other channels can be negatively effected as well. The underlying cause of these issues is an uneven request loading on resources within the memory system and the mechanism used to add requests to the ports. The best way to mitigate these issues is with optimal address and channel mapping schemes and port-adding heuristics – these are explored below.

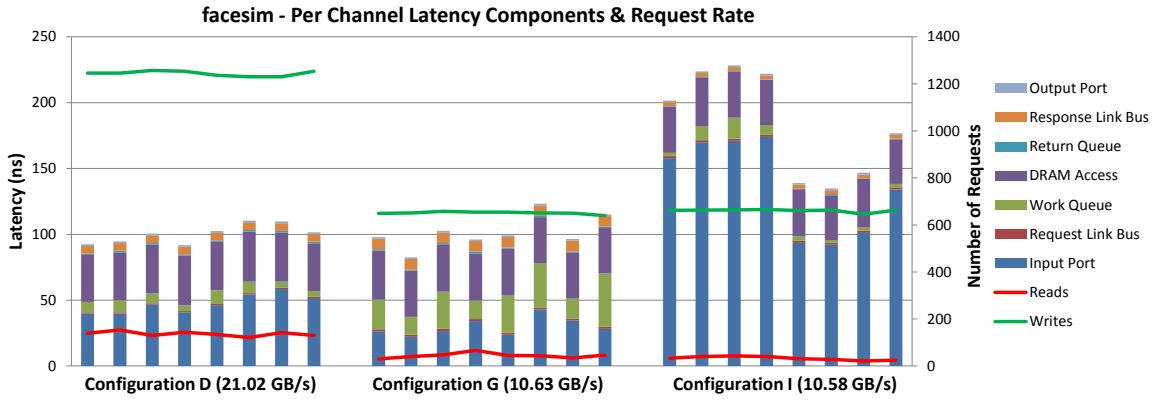


Figure 54: Latency components for requests sent to each DRAM channel during an epoch of the *facesim* benchmark

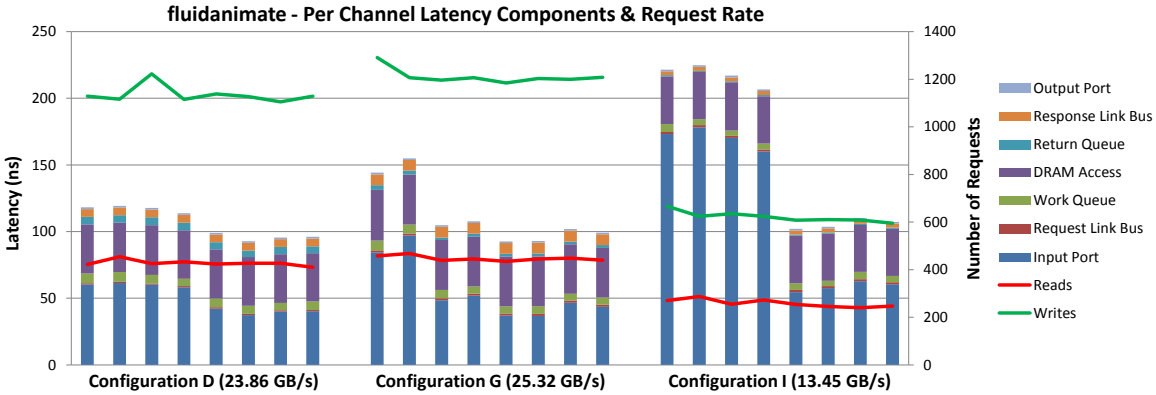


Figure 55: Latency components for requests sent to each DRAM channel during an epoch of the *fluidanimate* benchmark

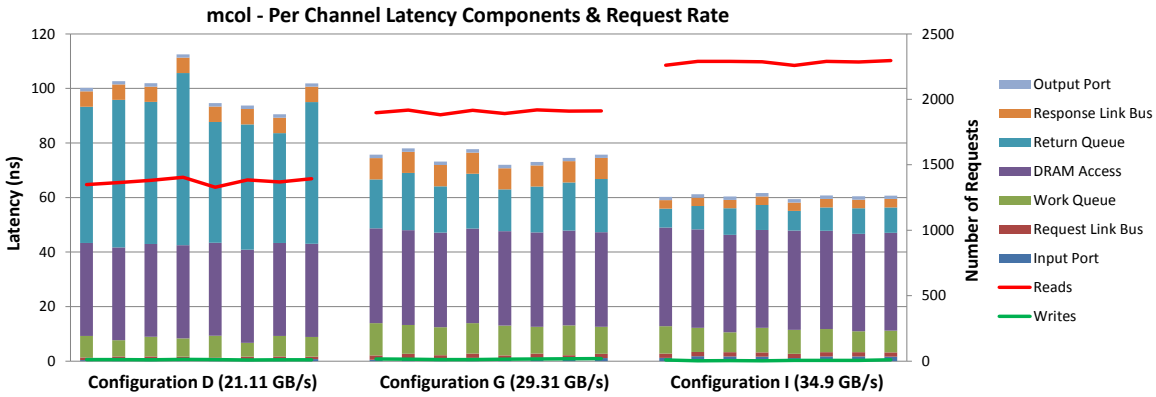


Figure 56: Latency components for requests sent to each DRAM channel during an epoch of the *mcol* benchmark

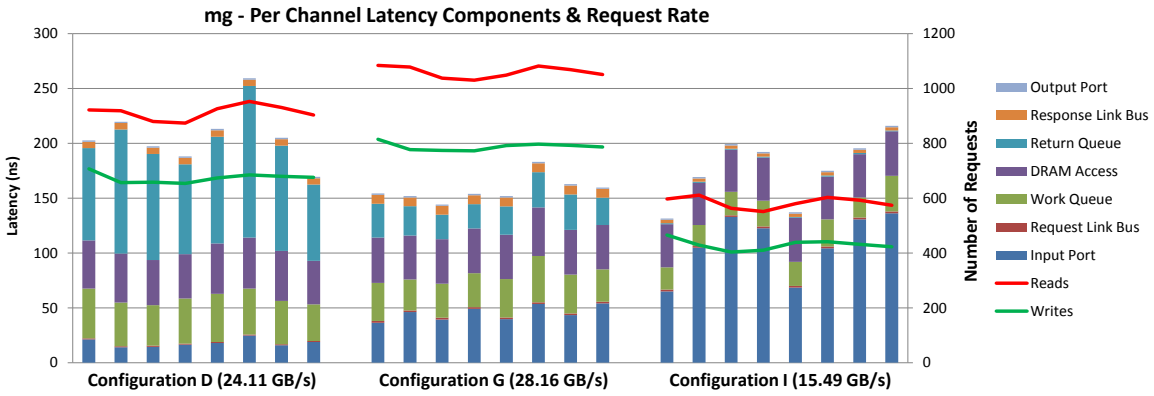


Figure 57: Latency components for requests sent to each DRAM channel during an epoch of the *mg* benchmark

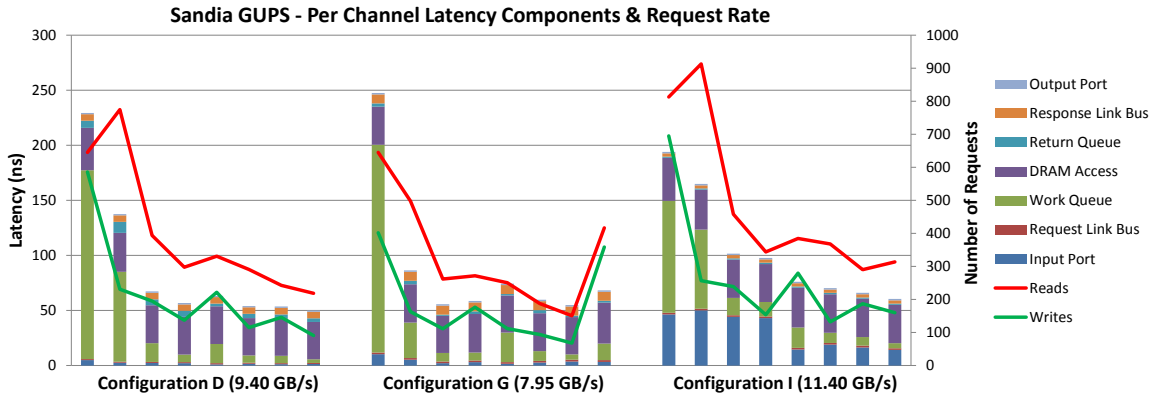


Figure 58: Latency components for requests sent to each DRAM channel during an epoch of the *Sandia GUPS* benchmark

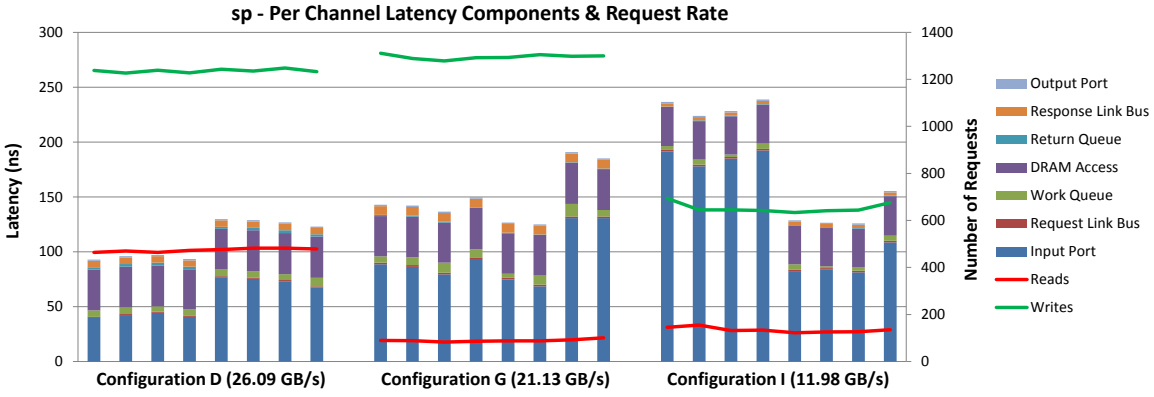


Figure 59: Latency components for requests sent to each DRAM channel during an epoch of the *sp* benchmark

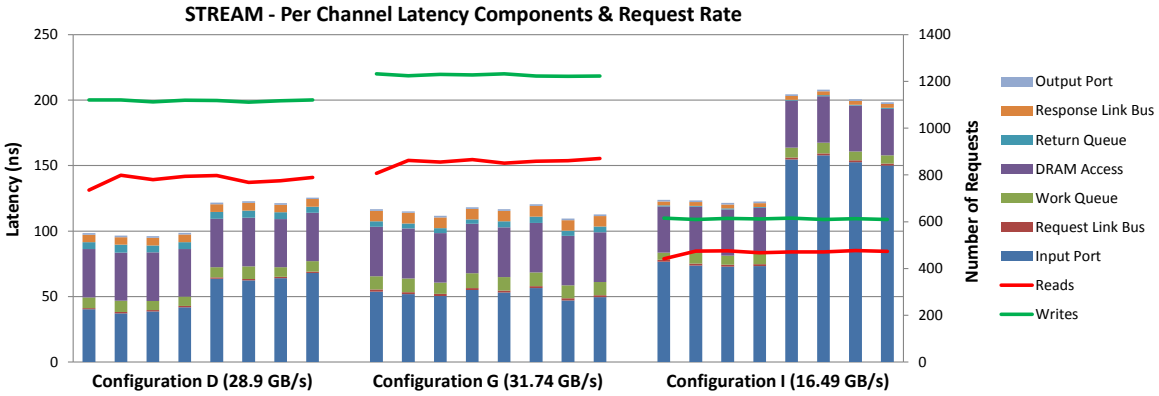


Figure 60: Latency components for requests sent to each DRAM channel during an epoch of the *STREAM* benchmark

4.2.3 Address Mapping

During a limit-case simulation, each request has a randomly generated address. Any mapping that would take place on this address would have no impact on the resulting use of resources because all bit combinations within the address are equally likely. During a full system simulation, the addresses within the request stream are a product of actual program execution and operating system functionality, and will therefore have address space locality that will cause some bit combinations to happen more frequently. When applying a mapping scheme to this address stream, particular resources may be used more often than others, thereby causing resource conflicts and loss of performance. This is why it is essential to find mapping schemes that optimally use the resources within the system.

In a BOB memory system, address mapping occurs in two separate but equally important places – within the main BOB controller and within a simple controller. The mapping that occurs within the BOB controller takes a portion of the physical address to determine which channel should receive that request. Ideally, this process

evenly spreads out requests over all available channels. An imbalanced mapping will flood a particular channel and cause contention on the link buses and subsequently within the actual DRAM.

The mapping that occurs within the simple controller is equivalent to the mapping that occurs in a standard memory system. This has been shown to be essential to the over-all performance and efficiency of the memory system [38, 42, 24, 39]; this applies to the BOB memory system as well. Before DRAM specific commands can be issued from the simple controller, the physical address of the request must be mapped to the available resources, such as a particular rank, bank, row, and column. A good address mapping scheme will prevent resource conflicts and utilize the parallelism available in modern DRAM devices to increase efficiency. Unfortunately, finding an optimal mapping scheme can be difficult due to variations in different address streams. An address mapping scheme that may be optimal with one workload could perform poorly with another.

To discuss the various mapping schemes which are used and analyzed, a convention is created that allows easy reference to how that scheme maps bits to resources. **Table 8** details the various bit fields required to map all available resources within a BOB memory system and the notation used for each of these fields. Using this convention, an address mapping scheme can now be referenced using the name for each bit field and the location that they are taken from the physical address. For example, the scheme RW:BK:RK:CH:CL:BY takes the highest-order bits of an address for the row address, the next-highest-order bits for the bank address, and so on.

Channel Bits	CH
Rank Bits	RK
Bank Bits	BK
Row Bits	RW
Column Bits (or) Column Bits High/Low	CL CLH and CLL
Byte Offset	BY

Table 8: Naming convention for each bit field in an address mapping scheme

The first mapping which takes place is the channel mapping within the main BOB controller, and determines which channel a request should be sent. This mapping is essential to evenly spread out requests over all the available channels. While determining which bits should be used for this mapping, the relative ordering of all other mappings is maintained to ensure that no other factors impact the performance. Once the optimal channel mapping bits have been determined, the mapping of other resources will be explored. **Table 9** display the mapping schemes used to determine which bits should be used for mapping the channel. **Note:** Relative order of other mappings are maintained.

Address Mapping Schemes
CH:RW:BK:RK:CL:BY
RW:CH:BK:RK:CL:BY
RW:BK:RK:CH:CL:BY
RW:BK:RK:CLH:CH:CLL:BY

Table 9: Mapping schemes used for determining optimal channel mapping bits

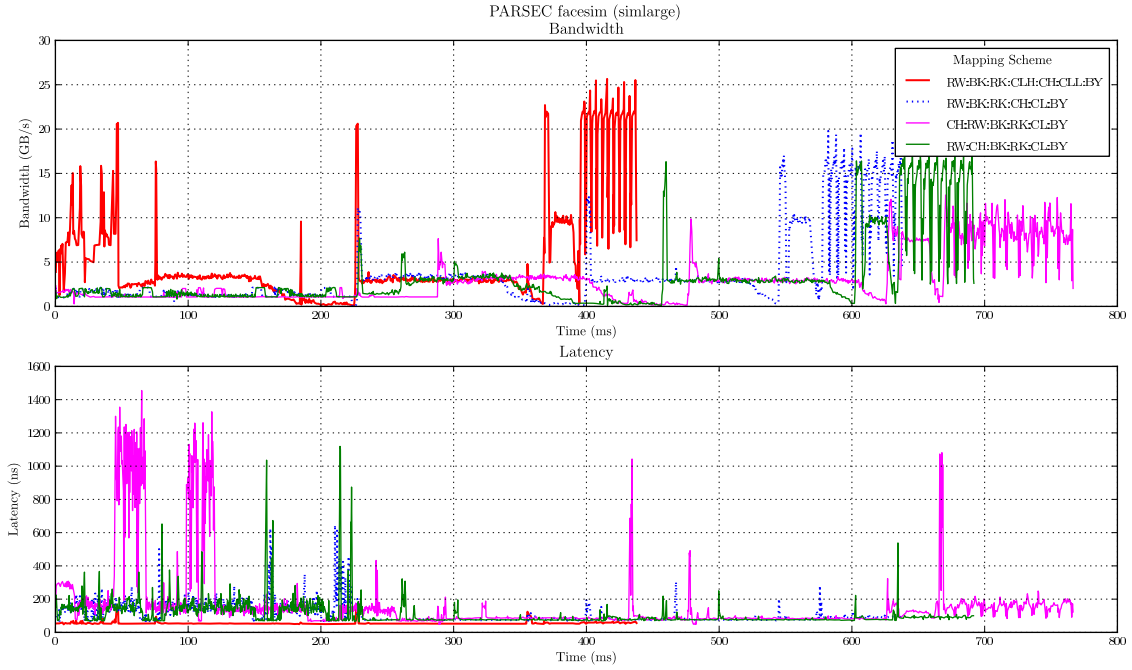
While executing various benchmarks during a full system simulation of configuration G, the mapping schemes shown in **Table 9** are used, and the resulting performance can be seen in **Figures 61** through **65**. The different mapping schemes produce a wide variety of achievable performance, due solely to how evenly they

spread the request stream across the DRAM channels. A clear pattern can be seen where configurations that use lower order bits to map the channel address can more evenly spread requests across DRAM channels and therefore achieve better performance. This behavior occurs because higher-order bits do not flip as frequently as lower-order bits [34]. As the channel mapping is taken from subsequently lower-order bits, the performance increases due a more even spread of requests across all channels. It is important to note that the lowest-order bits in the physical address are always reserved for a portion of the column address and the bus width offset. These bits account for both the width of the DRAM data bus and the amount of data returned in a single burst and are always 0 due to cache block alignment.

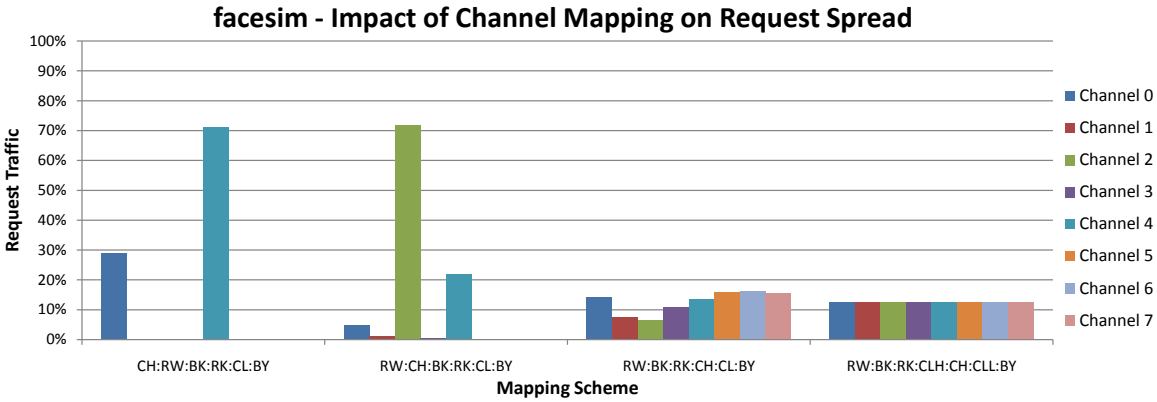
The CH:RW:BK:RK:CL:BY mapping scheme, which uses the highest-order bits to map the channel address, performs considerably worse during every benchmark because the mapping directs between 80% and 98% of the requests to the same channel, depending on the benchmark. This results in resource conflicts at every point along a request's path and the performance is limited to that of what is achievable by a single channel – in configuration G, which uses DDR3-1333, this is approximately 7.5 to 8 GB/s. Such poor performance leads to execution times significantly longer than other mappings, with *STREAM* and *mcol* executing almost six times longer than they do under the best mapping.

Conversely, mapping scheme RW:BK:RK:CLH:CH:CLL:BY performs the best during all benchmarks. In this mapping scheme, the lowest-order bits in the address that flip most regularly are used to map the request to a DRAM channel. These bits flip most frequently and, as a result, can evenly spread requests (within 1%) across all

channels. This makes resource conflicts less likely by utilizing all available parallelism within the system. The achieved resource-balancing causes the execution time of each benchmark to be better than all other mapping schemes. Therefore, the conclusion can be made that this mapping scheme is the most ideal; therefore, these bits will be used to map the DRAM channel in all subsequent simulations.



(a) Impact of Channel Mapping - *facesim*

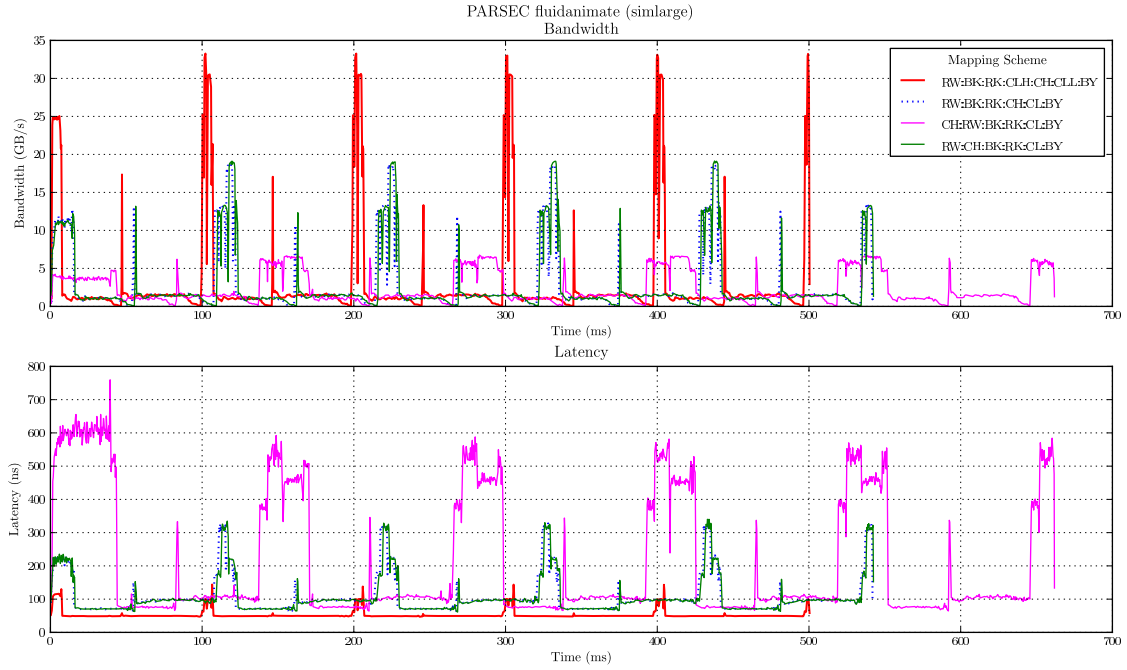


(b) Channel Spread - *facesim*

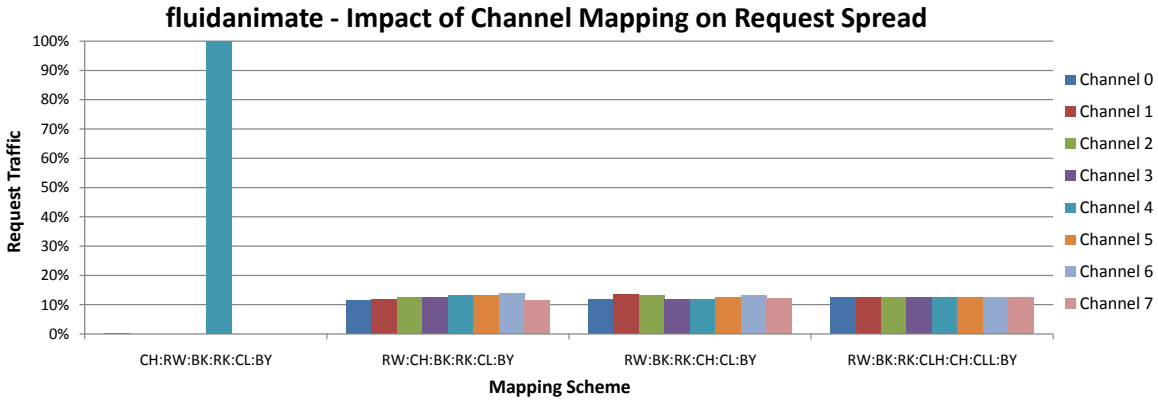
Figure 61: Impact of channel mapping scheme on performance and channel spread during *facesim* benchmark

facesim		
Mapping Scheme	**Average Bandwidth**	**Execution Time**
CH:RW:BK:RK:CL:BY	2.96 GB/s	766.6 ms
RW:CH:BK:RK:CL:BY	3.12 GB/s	691.9 ms
RW:BK:RK:CH:CL:BY	3.51 GB/s	637.8 ms
RW:BK:RK:CLH:CH:CLL:BY	5.22 GB/s	437.8 ms

Table 10: The average bandwidth and execution time using each address mapping scheme while executing *facesim*



(a) Impact of Channel Mapping - *fluidanimate*

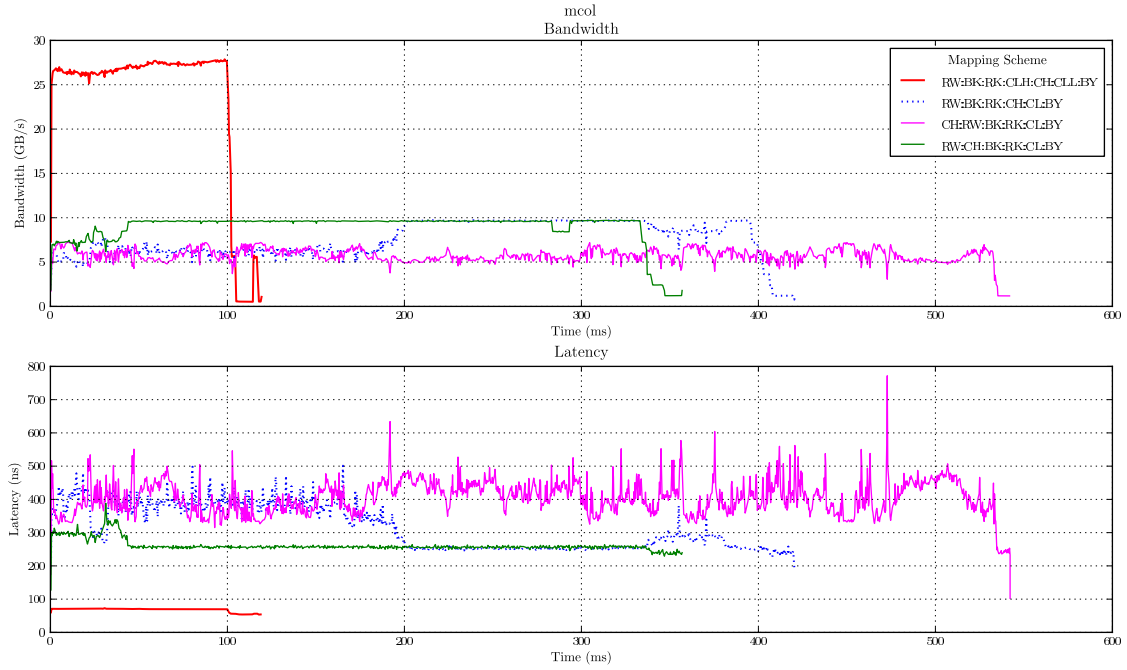


(b) Channel Spread - *fluidanimate*

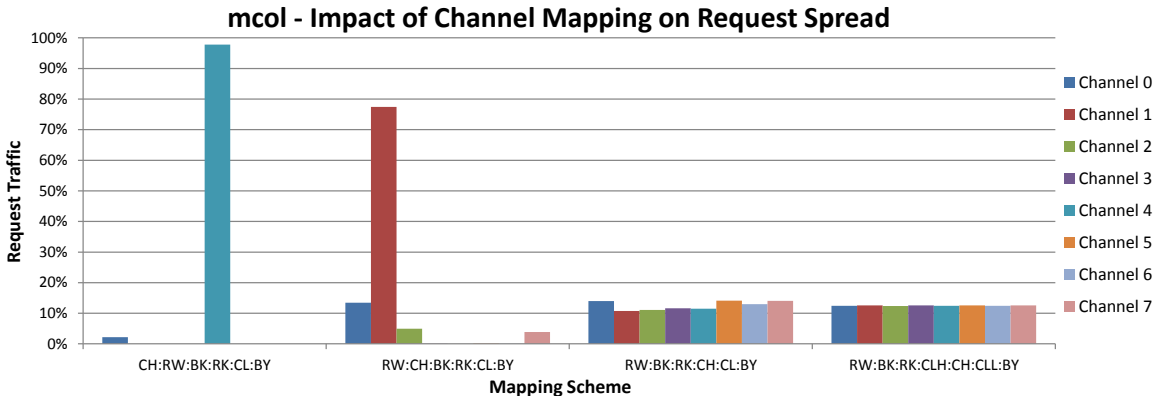
Figure 62: Impact of channel mapping scheme on performance and channel spread during *fluidanimate* benchmark

fluidanimate		
Mapping Scheme	**Average Bandwidth**	**Execution Time**
CH:RW:BK:RK:CL:BY	2.36 GB/s	661.8 ms
RW:CH:BK:RK:CL:BY	2.86 GB/s	542.5 ms
RW:BK:RK:CH:CL:BY	2.87 GB/s	541.9 ms
RW:BK:RK:CLH:CH:CLL:BY	3.10 GB/s	500.4 ms

Table 11: The average bandwidth and execution time using each address mapping scheme while executing *fluidanimate*



(a) Impact of Channel Mapping - *mcol*

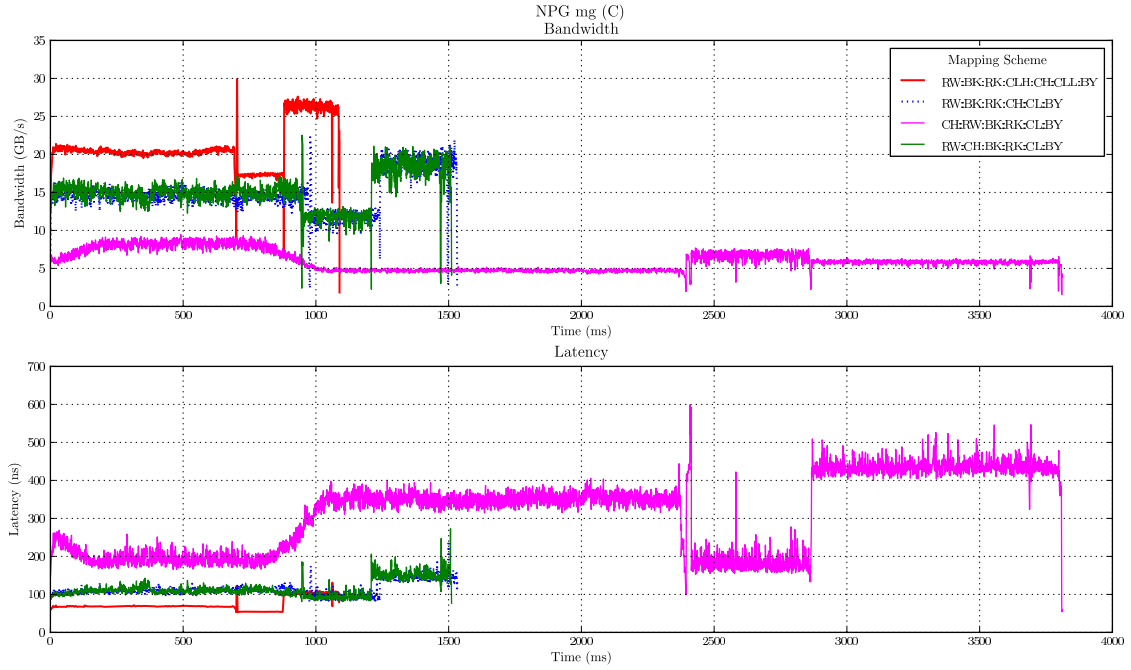


(b) Channel Spread - *mcol*

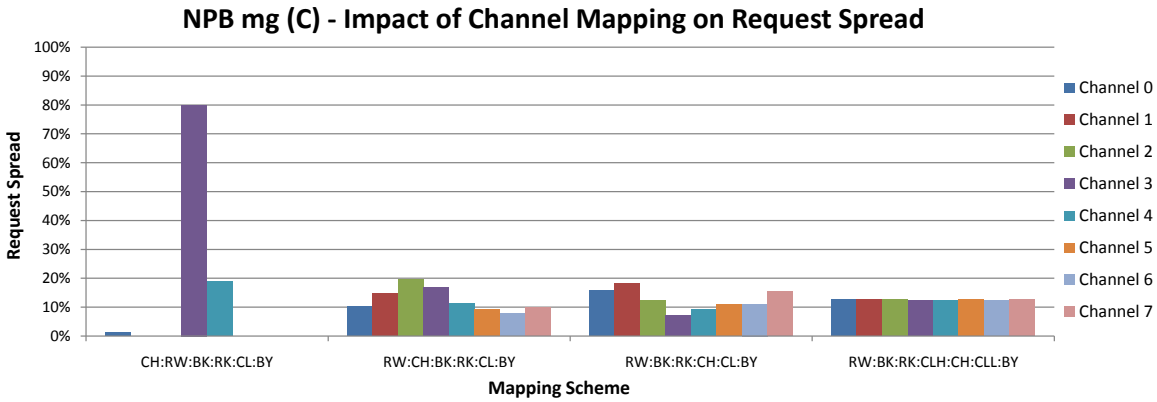
Figure 63: Impact of channel mapping scheme on performance and channel spread during *mcol* benchmark

mcol		
Mapping Scheme	**Average Bandwidth**	**Execution Time**
CH:RW:BK:RK:CL:BY	5.74 GB/s	542.2 ms
RW:CH:BK:RK:CL:BY	8.86 GB/s	356.9 ms
RW:BK:RK:CH:CL:BY	7.52 GB/s	420.4 ms
RW:BK:RK:CLH:CH:CLL:BY	23.18 GB/s	119.3 ms

Table 12: The average bandwidth and execution time using each address mapping scheme while executing *mcol*



(a) Impact of Channel Mapping - *mg*

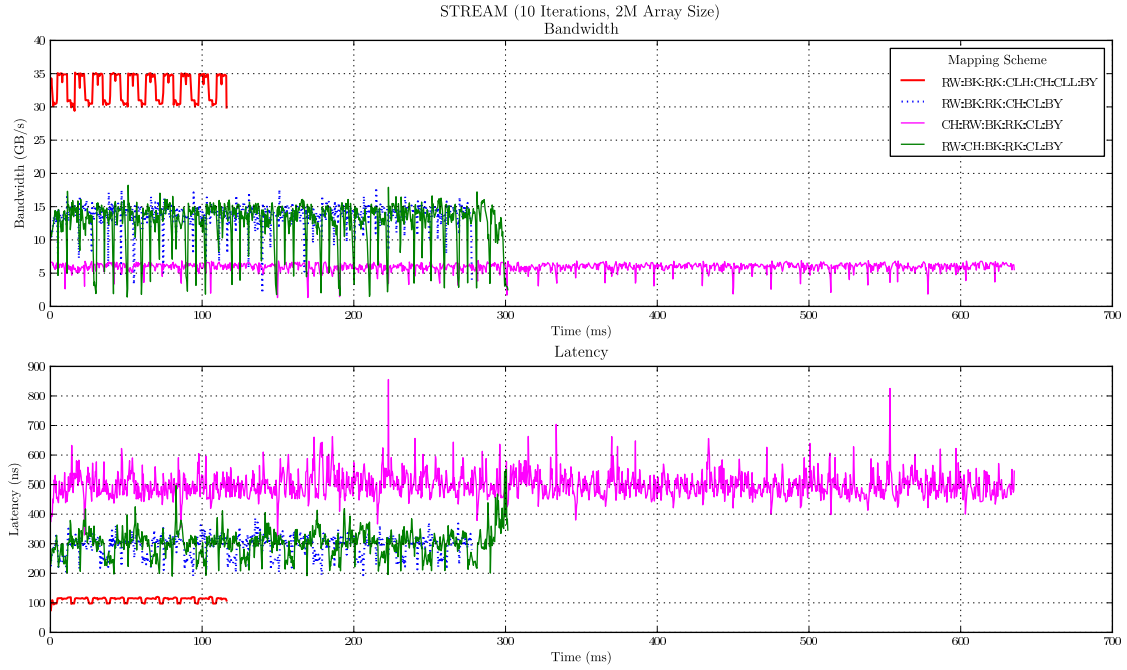


(b) Channel Spread - *mg*

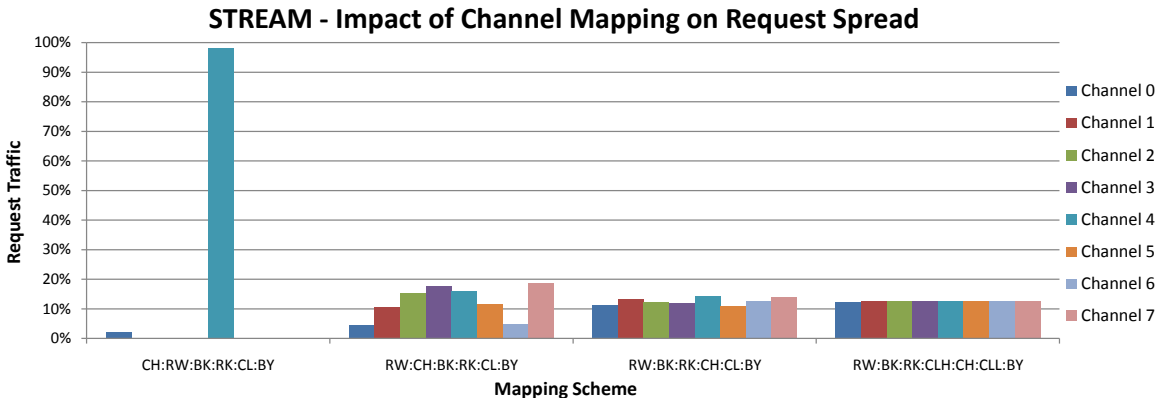
Figure 64: Impact of channel mapping scheme on performance and channel spread during *mg* benchmark

mg		
Mapping Scheme	**Average Bandwidth**	**Execution Time**
CH:RW:BK:RK:CL:BY	5.99 GB/s	3810 ms
RW:CH:BK:RK:CL:BY	15.09 GB/s	1511 ms
RW:BK:RK:CH:CL:BY	14.82 GB/s	1532 ms
RW:BK:RK:CLH:CH:CLL:BY	20.94 GB/s	1089 ms

Table 13: The average bandwidth and execution time using each address mapping scheme while executing *mg*



(a) Impact of Channel Mapping - *STREAM*



(b) Channel Spread - *STREAM*

Figure 65: Impact of channel mapping scheme on performance and channel spread during *STREAM* benchmark

STREAM		
Mapping Scheme	**Average Bandwidth**	**Execution Time**
CH:RW:BK:RK:CL:BY	5.96 GB/s	642.2 ms
RW:CH:BK:RK:CL:BY	12.78 GB/s	301.5 ms
RW:BK:RK:CH:CL:BY	13.87 GB/s	277.8 ms
RW:BK:RK:CLH:CH:CLL:BY	33.08 GB/s	116.1 ms

Table 14: The average bandwidth and execution time using each address mapping scheme while executing *STREAM*

With the optimal channel mapping bits determined, the DRAM mapping which takes place within the simple controller can be examined. This mapping is identical to the address mapping that takes place within a commodity memory system where portions of the address are used to determine which rank, bank, row, and column hold the data for a request. As with the channel mapping, the likelihood of a particular bit flipping is a key aspect in how that bit should be used to map resources. To properly utilize the parallelism within a DRAM channel (and device), bits that flip more frequently should be used to map resources of greater parallelism within the system.

All of the DRAM mapping schemes tested below (shown in **Table 15**) use the channel mapping bits that were determined to be optimal and the remaining portion of the address to map the DRAM resources – that is, all of the following mapping schemes use the lowest-order usable bits to map the channel address. Different combinations of resource mappings ensure that a clear picture is uncovered about how the DRAM mapping scheme impacts resource utilization and overall performance. This utilization can be visualized by examining the state of all the banks in each system. Each DRAM bank can be in one of four possible states: Idle (all rows precharged), Active (row in sense amplifier), Precharging (preparing sense amplifier), or Refreshing (issuing refresh commands). Ideally, a channel should have as many banks in the Active state (or Precharge) as possible, thereby utilizing the parallelism available within the DRAM devices. The figures below show the average number of banks in each state over an epoch of execution. It is important to note that, because the refresh action is periodic and independent of all other factors, the number of refreshing

banks is always the same.

Address Mapping Schemes
RK:BK:RW:CLH:CH:CLL:BY
CLH:RW:RK:BK:CH:CLL:BY
RW:CLH:BK:RK:CH:CLL:BY
BK:CLH:RW:RK:CH:CLL:BY
RW:BK:RK:CLH:CH:CLL:BY

Table 15: Mapping schemes used for determining optimal DRAM mapping bits

The execution of various benchmarks using these DRAM mapping schemes in BOB configuration G can be seen in **Figure 66** through **70**. These figures (**Figures 66(b)** through **70(b)**) also display the bank utilization within the DRAM channel by displaying the number of DRAM banks in a particular state (Idle, Active, Precharging, Refreshing). As with channel mapping, DRAM mapping schemes that use lower-order bits to map higher levels of parallelism perform better than those that do not. For example, mapping scheme RW:CLH:BK:RK:CH:CLL:BY uses the lowest-order unused bits after channel mapping to address the bank and rank within the DRAM channel, and it performs the best across all benchmarks. The lower-order bits flip more frequently and will therefore be more likely to evenly spread requests across all the ranks and banks within that channel. This is confirmed in the bank utilization figures as well, where this DRAM mapping scheme always has the least number of idle banks.

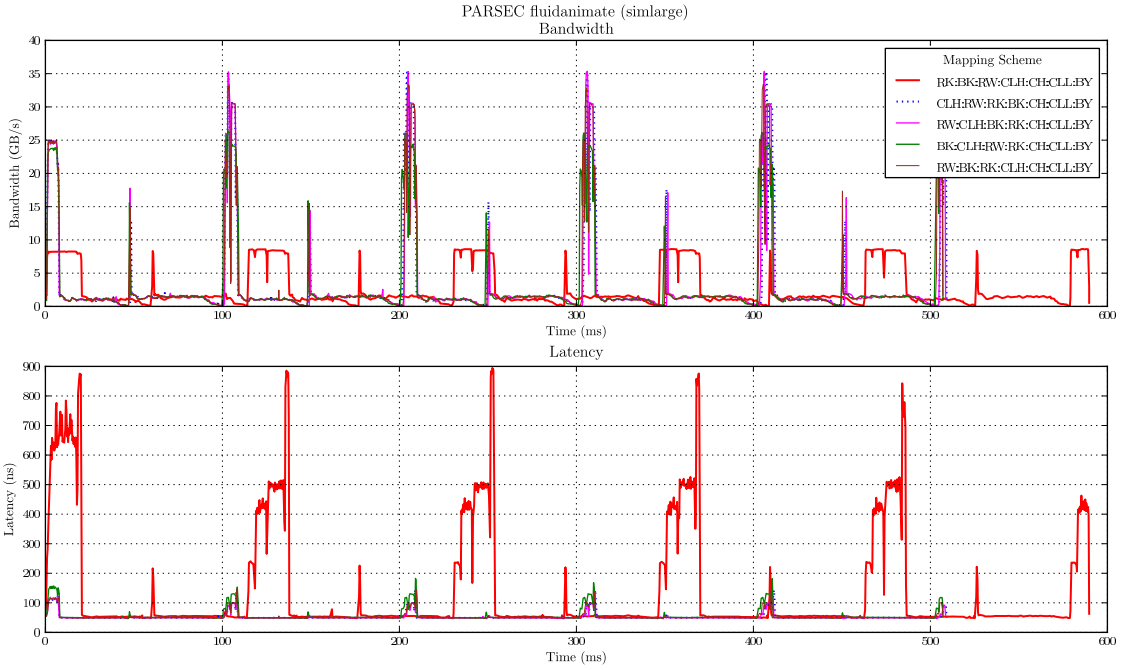
Conversely, mapping scheme RK:BK:RW:CLH:CH:CLL:BY uses the highest order bits to determine which rank a particular request maps to, and it performs the worst across all benchmarks. This scheme performs the worst because the higher-order bits that map the rank flip less frequently and will therefore route a majority of requests

to only a few ranks or banks within the channel, thus causing conflicts and reducing performance. The bank utilization of this mapping scheme (**Figures 66(b)** through **70(b)**) confirms this as well with the fewest active banks. This can be used to infer that requests are not fully utilizing the rank and bank parallelism within the DRAM channel. The RK:BK:RW:CLH:CH:CLL:BY mapping scheme also makes it clear that even when an optimal channel mapping scheme can evenly spread requests across all DRAM channels in the system, the DRAM mapping scheme implemented within the simple controller still has a major impact on performance.

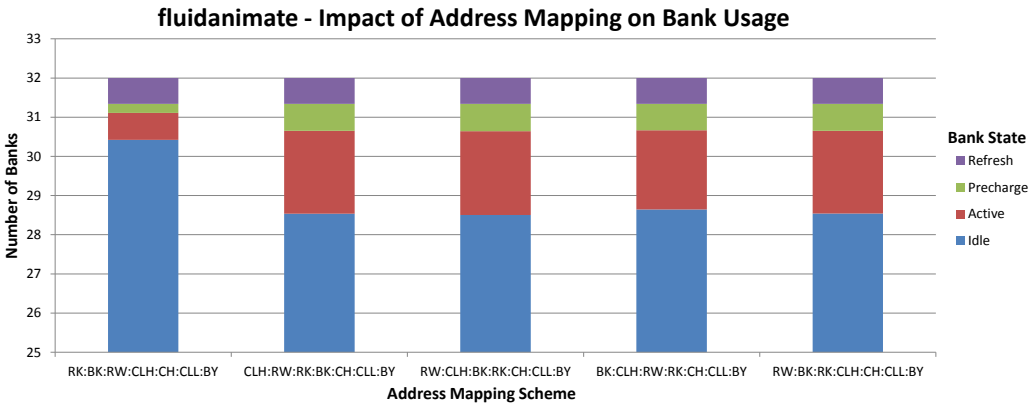
DRAM mapping scheme BK:CLH:RW:RK:CH:CLL:BY performs marginally worse than the best performing schemes but significantly better than RK:BK:CLH:CH:CLL:BY. When considering a bank address as the combination of both the rank and bank bits, this indicates that even if a portion of that address is taken from lower order bits (unlike RK:BK:CLH:CH:CLL:BY), performance can be significantly improved. The other DRAM mapping schemes had different relative performance depending on the benchmark and point of execution. This shows that while the DRAM and channel mapping scheme can have a dramatic impact on the performance of the system, the incoming address request stream is equally as important in determining how effective these mapping schemes can be.

As with the channel mapping, DRAM utilization has an impact on other parts of the system. If a DRAM mapping scheme results in a high number of conflicts, requests must be stalled in various queues. This can cause back-ups in other parts of the system, thereby negatively impacting other DRAM channels. For example, the RK:BK:RW:CLH:CH:CLL:BY mapping scheme has the least number of active

banks, and as a result the command queues have an average of 9.2 requests waiting to be issued. When the work queue is full, the request latency will increase as seen in these graphs, and there is a chance that requests will be stalled in the port buffers as well, causing other DRAM channels to stall. On the other hand, the RW:CLH:BK:RK:CH:CLL:BY mapping scheme has the most active banks and only has an average of 0.6 requests waiting in the queue. This means that requests can easily move throughout the rest of the system, preventing back-ups and reducing overall system latency.



(a) Impact of Address Mapping - *fluidanimate*

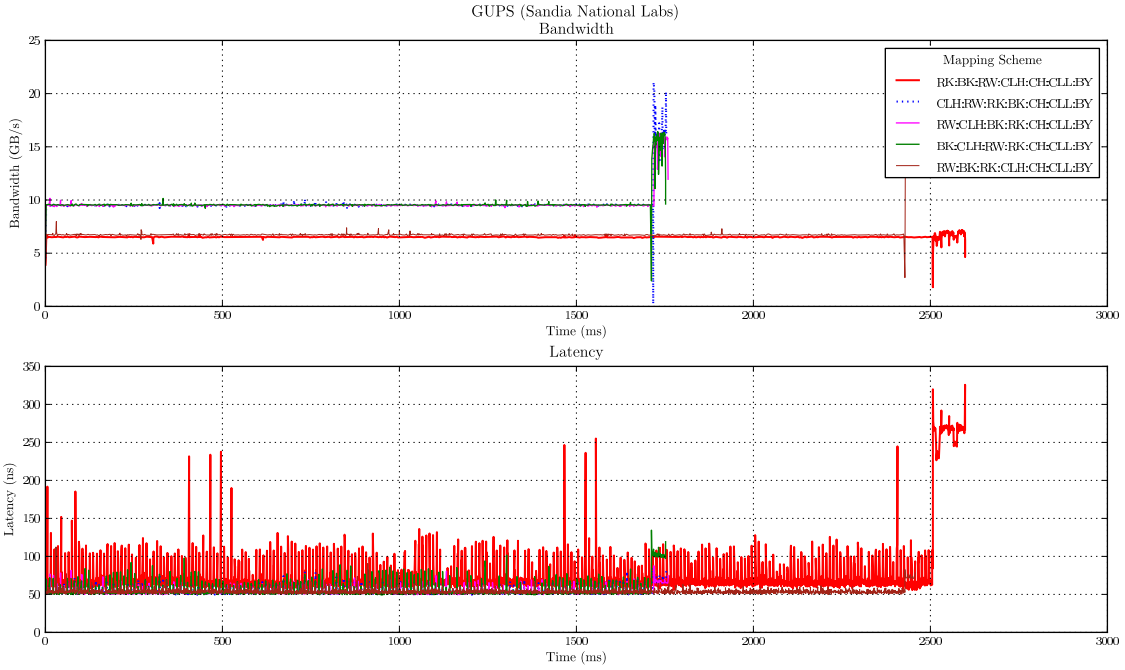


(b) Bank Utilization - *fluidanimate*

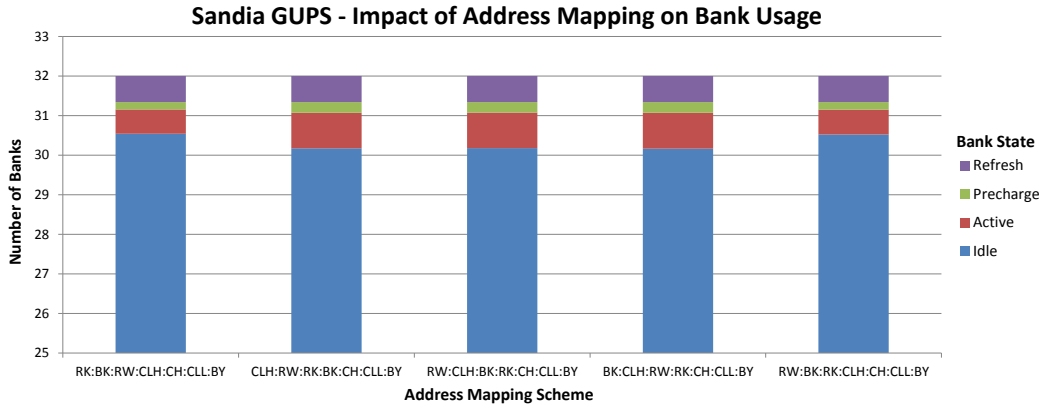
Figure 66: DRAM mapping impact on *STREAM* benchmark

fluidanimate		
Mapping Scheme	**Average Bandwidth**	**Execution Time**
RK:BK:RW:CLH:CH:CLL:BY	2.64 GB/s	589.7 ms
CLH:RW:RK:BK:CH:CLL:BY	3.05 GB/s	509.1 ms
RW:CLH:BK:RK:CH:CLL:BY	3.06 GB/s	508.6 ms
BK:CLH:RW:RK:CH:CLL:BY	3.07 GB/s	507.1 ms
RW:BK:RK:CLH:CH:CLL:BY	3.06 GB/s	507 ms

Table 16: The average bandwidth and execution time using each DRAM mapping scheme during *fluidanimate*



(a) Impact of Address Mapping - *Sandia GUPS*

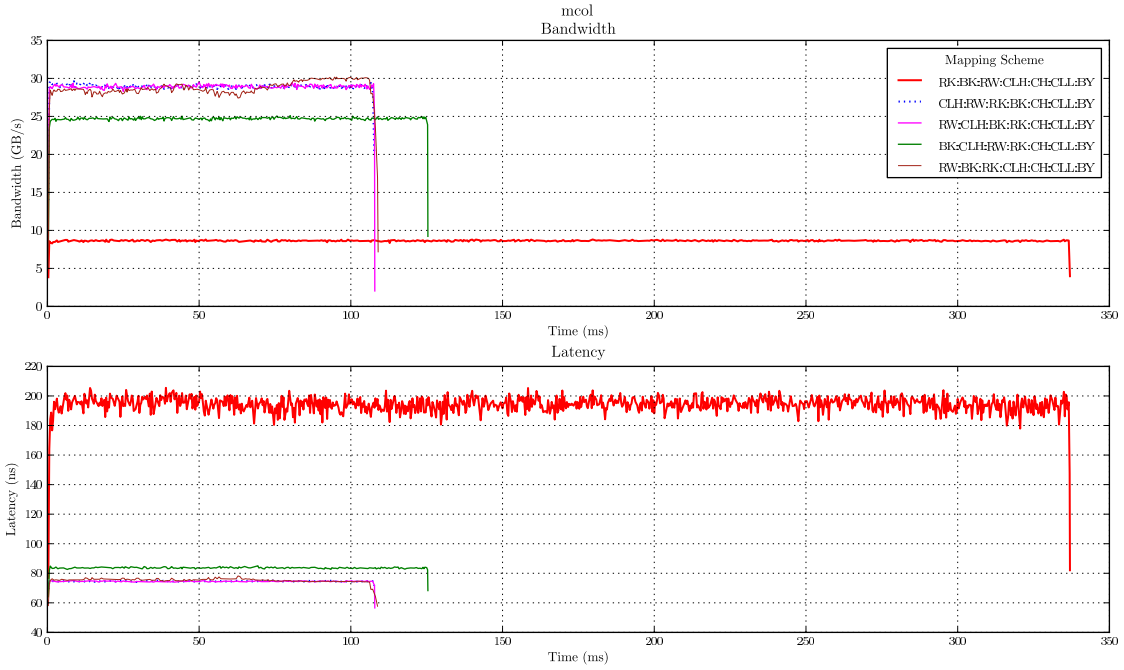


(b) Bank Utilization - *Sandia GUPS*

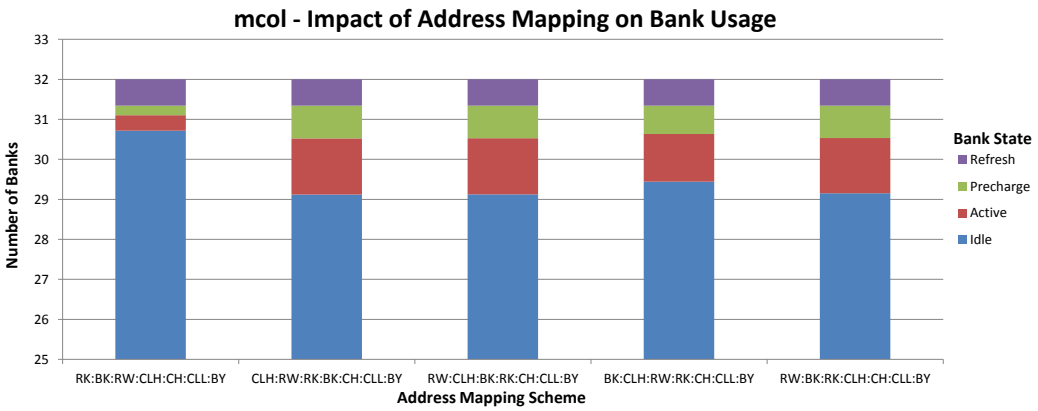
Figure 67: DRAM mapping impact on *Sandia GUPS* benchmark

Sandia GUPS		
Mapping Scheme	**Average Bandwidth**	**Execution Time**
RK:BK:RW:CLH:CH:CLL:BY	6.52 GB/s	2598 ms
CLH:RW:RK:BK:CH:CLL:BY	9.65 GB/s	1754 ms
RW:CLH:BK:RK:CH:CLL:BY	9.63 GB/s	1759 ms
BK:CLH:RW:RK:CH:CLL:BY	9.66 GB/s	1752 ms
RW:BK:RK:CLH:CH:CLL:BY	6.89 GB/s	2457 ms

Table 17: The average bandwidth and execution time using each DRAM mapping scheme during *Sandia GUPS*



(a) Impact of Address Mapping - *mcol*

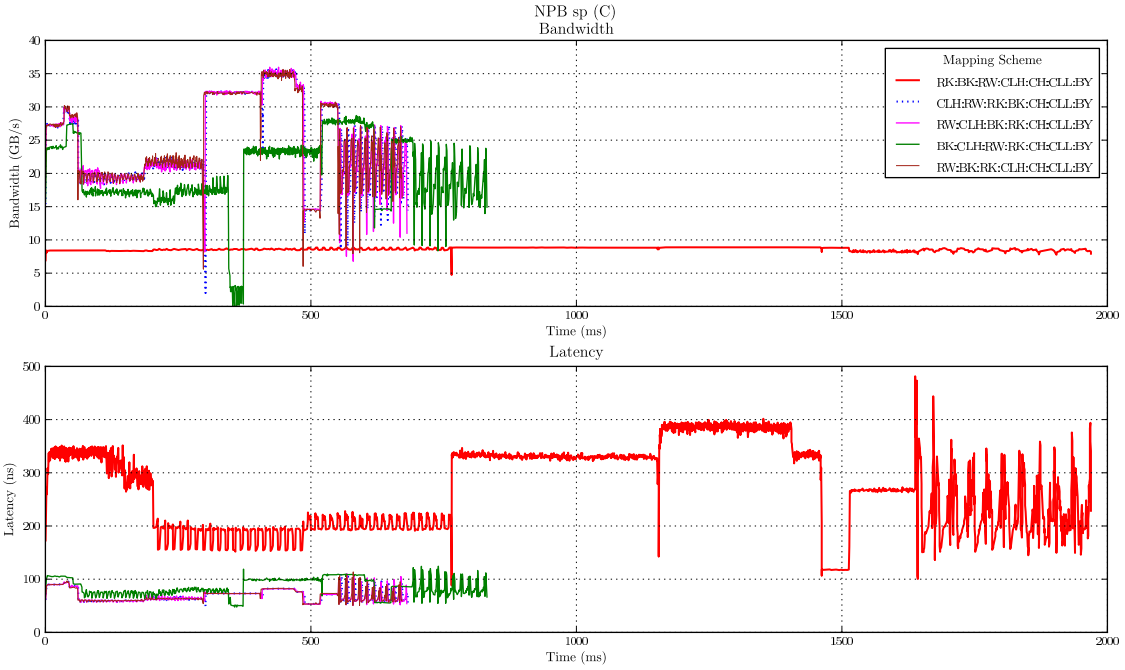


(b) Bank Utilization - *mcol*

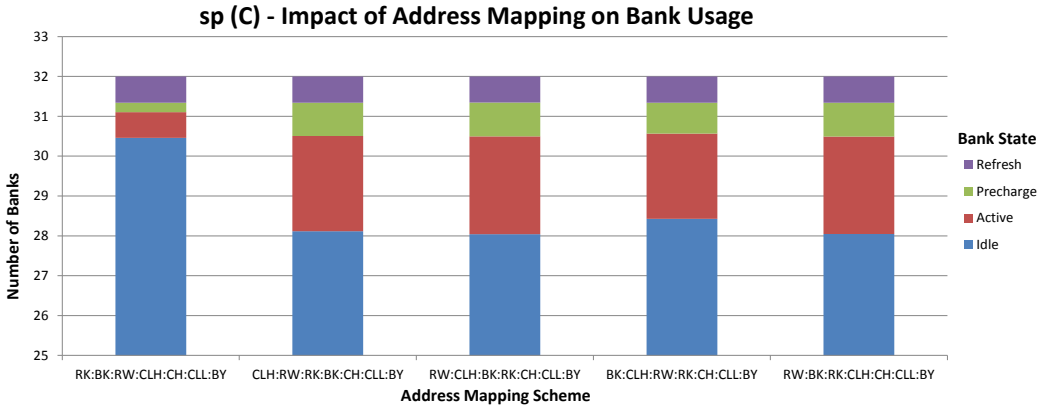
Figure 68: DRAM Mapping impact on *mcol* benchmark

mcol		
Mapping Scheme	**Average Bandwidth**	**Execution Time**
RK:BK:RW:CLH:CH:CLL:BY	8.63 GB/s	337 ms
CLH:RW:RK:BK:CH:CLL:BY	28.91 GB/s	107.7 ms
RW:CLH:BK:RK:CH:CLL:BY	28.79 GB/s	107.9 ms
BK:CLH:RW:RK:CH:CLL:BY	24.62 GB/s	125.4 ms
RW:BK:RK:CLH:CH:CLL:BY	28.58 GB/s	109 ms

Table 18: The average bandwidth and execution time using each DRAM mapping scheme during *mcol*



(a) Impact of Address Mapping - *sp*

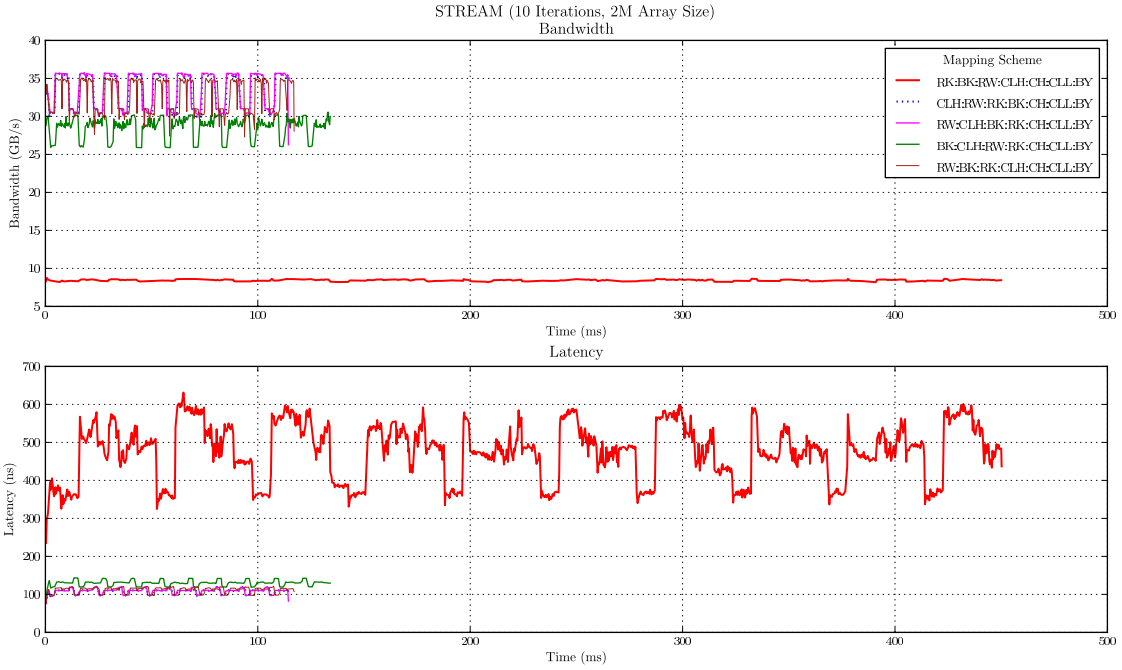


(b) Bank Utilization - *sp*

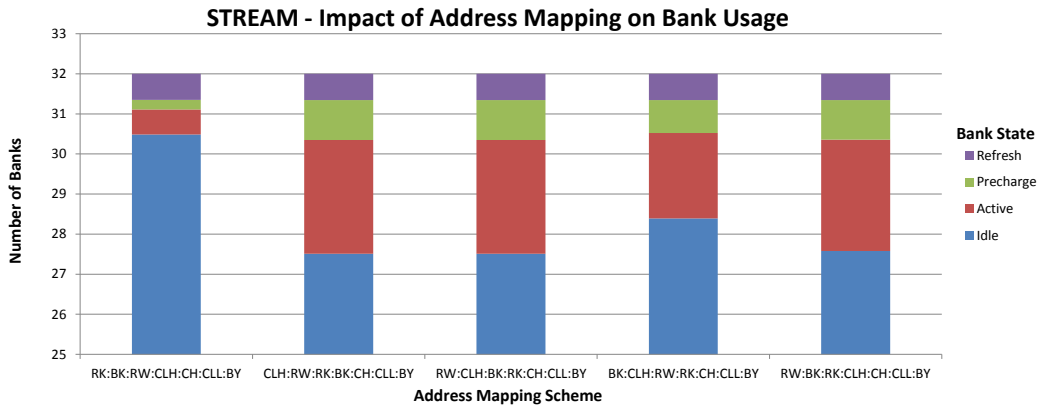
Figure 69: DRAM mapping impact on the *sp* benchmark

sp		
Mapping Scheme	**Average Bandwidth**	**Execution Time**
RK:BK:RW:CLH:CH:CLL:BY	8.63 GB/s	1969 ms
CLH:RW:RK:BK:CH:CLL:BY	24.88 GB/s	682.5 ms
RW:CLH:BK:RK:CH:CLL:BY	24.90 GB/s	680.9 ms
BK:CLH:RW:RK:CH:CLL:BY	20.14 GB/s	831.9 ms
RW:BK:RK:CLH:CH:CLL:BY	25.01 GB/s	671.9 ms

Table 19: The average bandwidth and execution time using each DRAM mapping scheme during *sp*



(a) Impact of Address Mapping - *STREAM*



(b) Bank Utilization - *STREAM*

Figure 70: DRAM mapping impact on *STREAM* benchmark

STREAM		
Mapping Scheme	**Average Bandwidth**	**Execution Time**
RK:BK:RW:CLH:CH:CLL:BY	8.42 GB/s	450.2 ms
CLH:RW:RK:BK:CH:CLL:BY	33.46 GB/s	114.7 ms
RW:CLH:BK:RK:CH:CLL:BY	33.52 GB/s	114.4 ms
BK:CLH:RW:RK:CH:CLL:BY	28.60 GB/s	134.3 ms
RW:BK:RK:CLH:CH:CLL:BY	32.80 GB/s	117.1 ms

Table 20: The average bandwidth and execution time using each DRAM mapping scheme during *STREAM*

In conclusion, the importance of both the channel and address mapping is significant. Both are essential for effectively using all the available resources in the system, from properly spreading requests over channels to using all of the parallel banks within a DRAM device. Ultimately, the incoming address stream will always be a strong determining factor in how efficient the mapping scheme is, yet the principles outlined above will always apply; that is, higher levels of parallelism should be mapped with lower order bits within the address, as they are more likely to evenly spread requests across all resources. The reasoning and simulations detailed above show clear evidence that this is the case, with mapping scheme RW:CLH:BK:RK:CH:CLL:BY being the most optimal mapping scheme for a BOB memory system; this scheme is used in all subsequent simulations.

4.2.4 Read Return Queue

The read return queue within each simple controller is responsible for storing requested read data while awaiting arbitration on the response link bus. To ensure proper request ordering, if the queue is full at any point, no read or write commands will be issued to the DRAM until space for new data is available. During a limit-case simulation, the random address stream dictates a constant read-to-write ratio and an even spread across all channels. While this will show some of the impact that the queue has on the system, it is not indicative of actual program execution. During a full-system simulation, locality (both spatial and temporal) in the address stream will cause certain channels to receive more requests than others, and the read-to-write ratio will change throughout the various parts of the benchmark's execution.

Full system simulation results will give a far more accurate picture of how the amount of storage given to this queue will impact both the attached DRAM channel and other parts of the system as well.

Figures 71 through **75** display each BOB configuration executing various benchmarks while changing the storage capacity of the read return queue in each simple controller. The differences between each BOB configuration will illustrate how the organization of the link buses and simple controllers in each configuration will interact with this queue and the resulting impact it has on system performance. Each configuration will behave differently because the determining factors in how quickly items are removed from this queue are a) the width of each response link bus and b) the total number of these buses in the configuration; each BOB configuration is different in this regard. The relative difference in performance between various queue capacities will also vary during each benchmark as a result of the read-to-write ratio and request frequency changing depending on the point of the benchmark's execution. A period of a more read requests will require more capacity in this queue than a period of greater writes and will therefore have a greater effect on the performance.

As within the limit-case simulations, when the return queue only has the capacity for a single request (64 bytes), the performance is the worst across all benchmarks and BOB configurations. Such a restrictive queue size will stall the DRAM operation as soon as a single read request is issued, thereby causing significant back-ups throughout the system. Performance can be similar to configurations with a greater queue depth when the request rate is slow, as in parts of *fluidanimate*, or during periods of a greater number of writes than reads. An example of this can be seen in portions of

the *STREAM* benchmark that have a greater number of writes than reads and the achieved bandwidth is within 1% of configurations with a larger queue capacity. In these situations, the read return queue is not the bottleneck to performance and can therefore achieve similar bandwidth to other configurations. Regardless, these are uncommon situations and a storage capacity of 64 bytes is far too restrictive.

Simply doubling the queue size to accommodate two response packets (128 bytes) can increase the average bandwidth during a benchmark by up to 76%. The greater the read-to-write ratio is during a benchmark, the more likely an increased capacity in this queue will be beneficial to performance. The *mcol* benchmark is an extreme example of this with a request stream of approximately 98% reads; the average bandwidth during the execution is increased by 76% when the return queue size is doubled to 128 bytes. Other benchmarks see significant performance gains as well with the average bandwidth during *sp*, *STREAM*, and *mg* increasing by 27%, 22%, and 19%, respectively.

Further increasing the queue capacity from two (128 bytes) to eight (512 bytes) provides marginal bandwidth gains of up to 6%. This is because there is only a small likelihood that during typical program execution the return queue will require such capacity over a period which would otherwise stall DRAM operation. Regardless, all configurations see the best performance when the storage capacity in this queue is 512 bytes. For such a relatively small amount of storage, it would seem to be an obvious decision that the queue be given such a capcity.

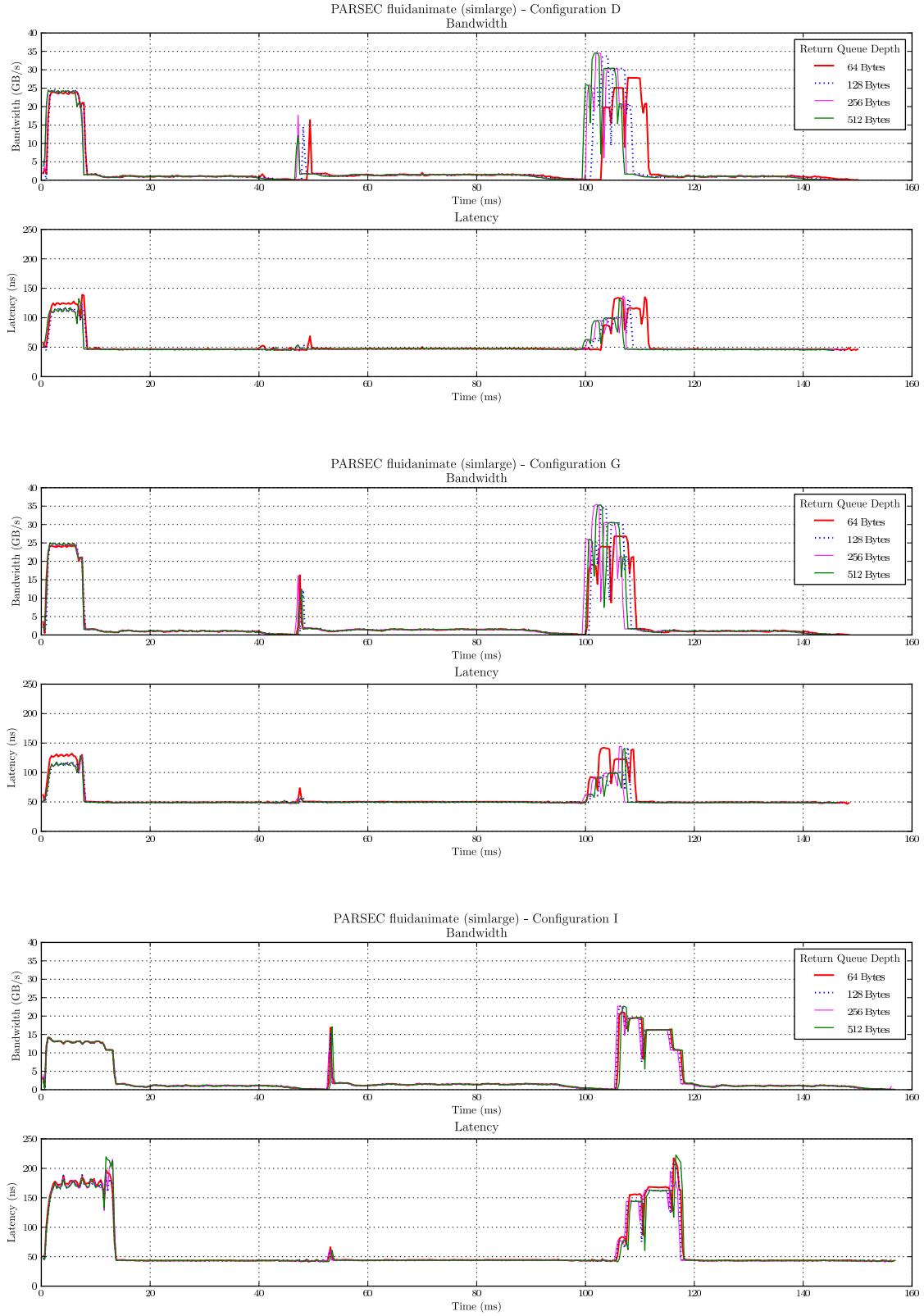


Figure 71: The impact of the return queue depth on performance during *fluidanimate* benchmark

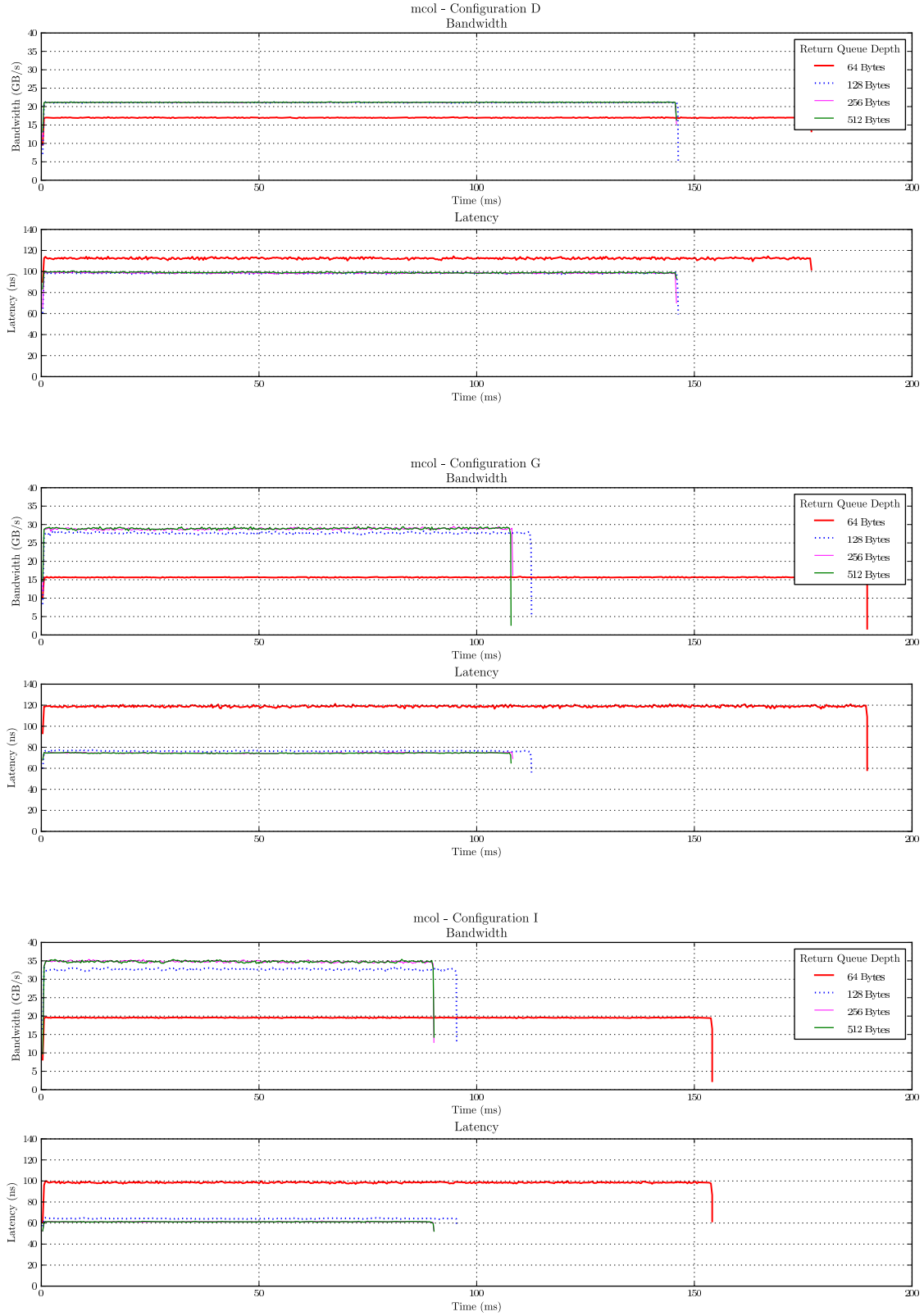


Figure 72: The impact of the return queue depth on performance during *mcol* benchmark

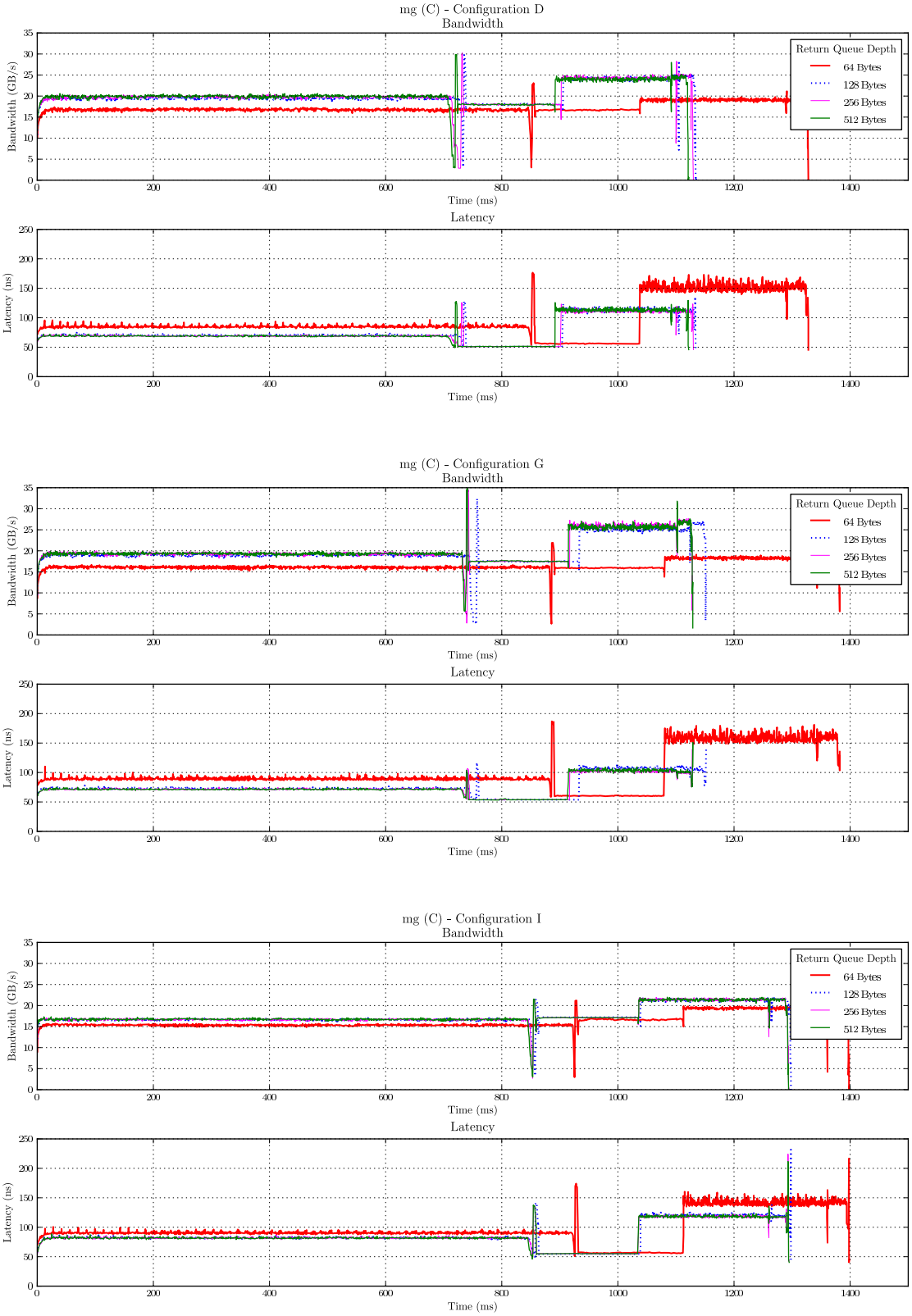


Figure 73: The impact of the return queue depth on performance during *mg* benchmark

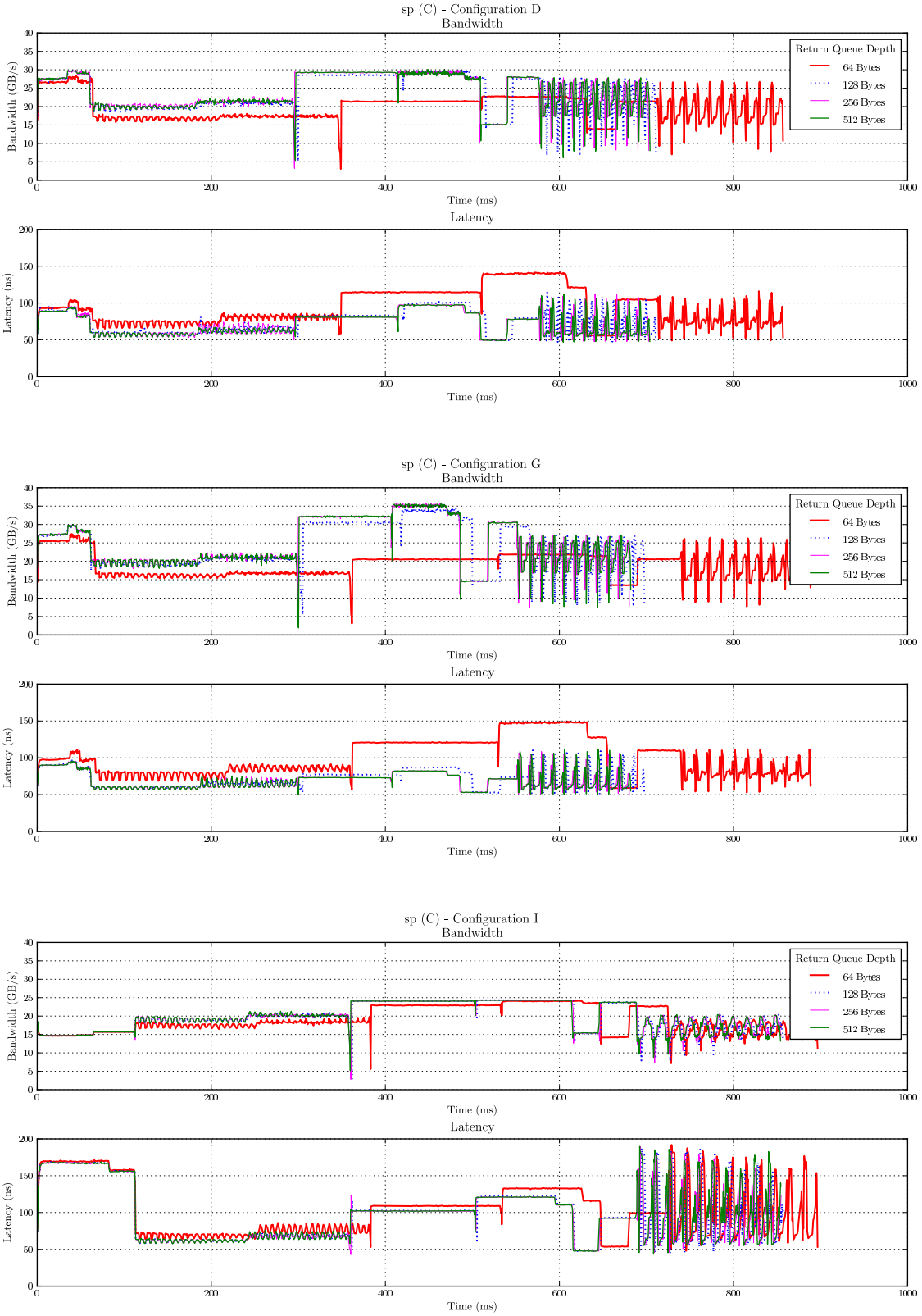


Figure 74: The impact of the return queue depth on performance during *sp* benchmark

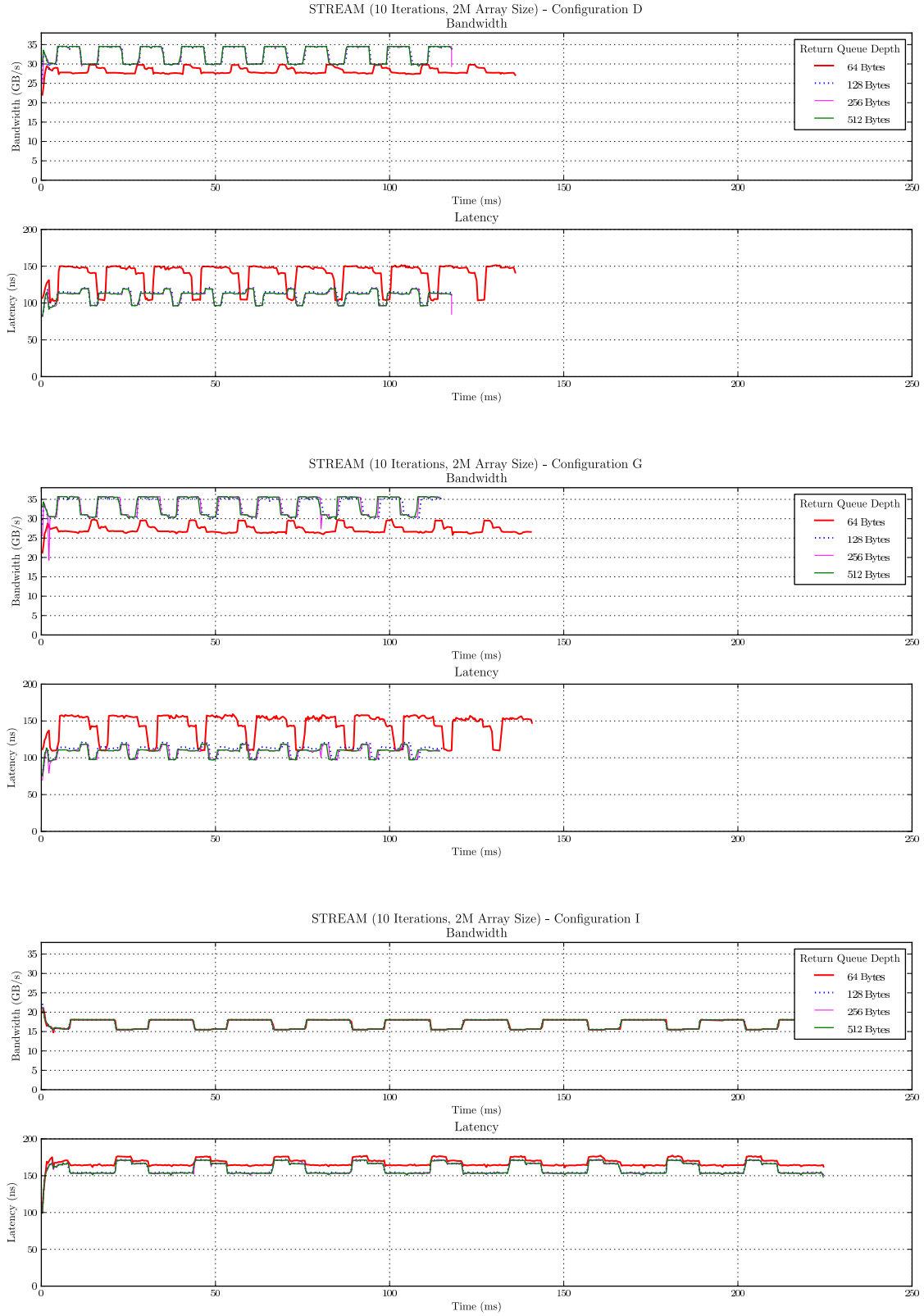


Figure 75: The impact of the return queue depth on performance during *STREAM* benchmark

The figures above also indicate that the return queue depth can have a significant impact on request latency as well. This can be further examined when the latency components of requests are displayed for each configuration during each benchmark; these are shown in **Figure 76**. While increasing the capacity of this queue always reduces the request latency across all configurations, it will also alter the relative sizes of each of the latency components. For example the time spent in the work queue and the DRAM access time are universally reduced due to less frequent stalling because of a full read return queue. Conversely, the time spent in the read return queue increases as there is now the possibility for multiple entries to reside there, some of which must wait to be sent out on the response link bus. In some cases, the time spent in the port input buffer also marginally increases. This is because the increased return queue capacity allows a greater number of requests to be in the memory system at any one time, thereby allowing program execution to stall less frequently and the CPU to issue more requests.

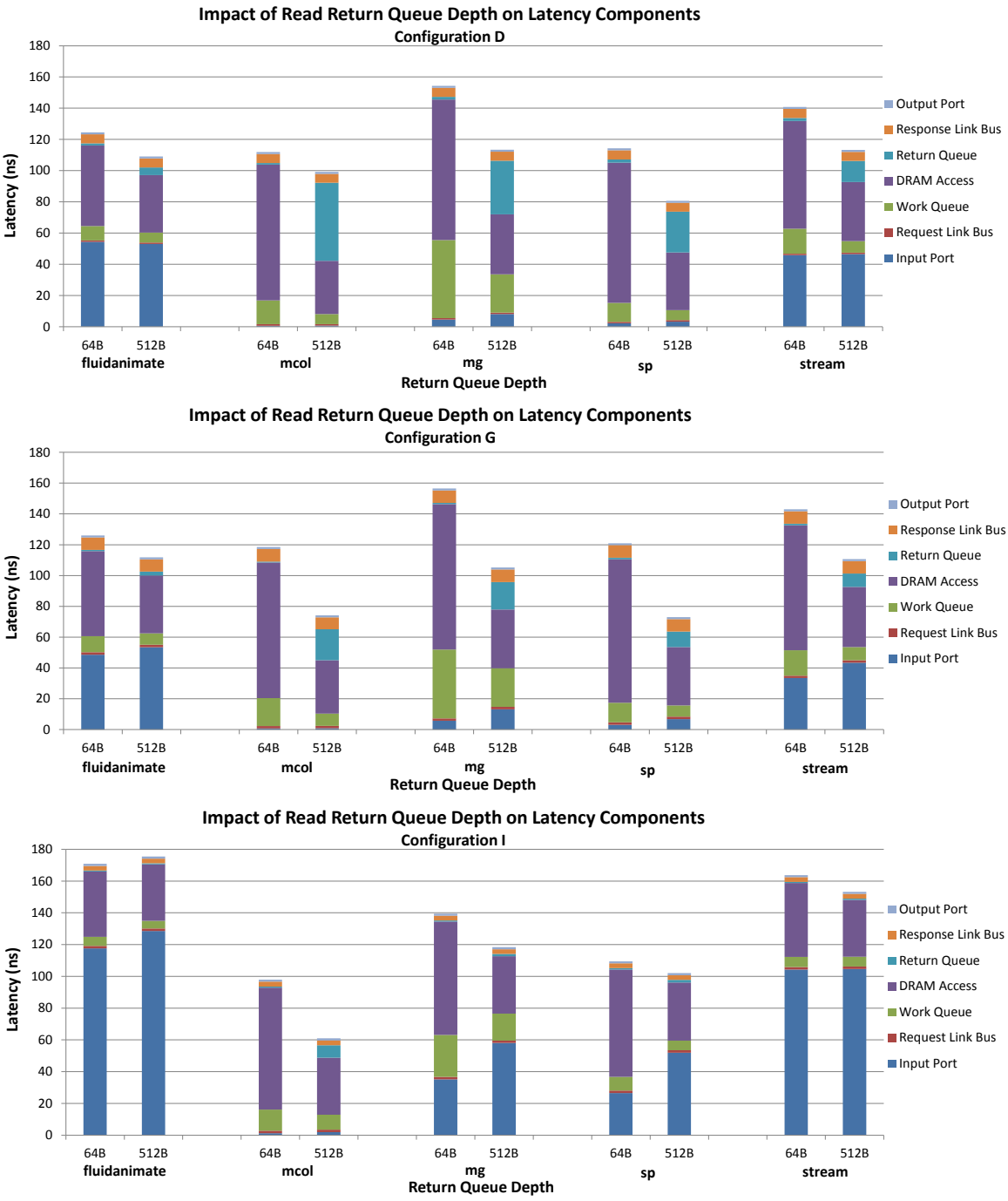


Figure 76: The impact of the return queue depth on latency components during each benchmark

While it is clear that increasing the capacity of this queue grants an increase in performance by allowing more requests to be issued to the DRAM, the relative

performance increase is different between each BOB configuration. As previously mentioned, this is due to the drastically different link bus organization in each configuration simulated. Configuration G has the least response link bus bandwidth of all the configurations (four link buses with a response link bus bandwidth of 9.6 GB/s each) and benefits the most from an increased read return queue depth. The impact the queue capacity has on the achieved bandwidth during each of the benchmarks and the reduction in execution time can be seen in **Table 21**. Across all benchmarks, the achieved bandwidth of configuration G increases an average of 32% when increasing the depth of the return queue from one (64 bytes) to eight (512 bytes). On the other hand, configurations D and I see an average bandwidth increase of 16% and 18% respectively. This is because the response link bus in configuration G takes the longest amount of time to return a response packet and therefore benefits the most from extra storage as other requests must wait longer to be issued if the queue is full.

Configuration D				
Benchmark	**64 B**	**512 B**	**BW Increase**	**Speedup**
fluidanimate	3.32 GB/s	3.41 GB/s	2.7%	2.9%
mcol	16.95 GB/s	21.12 GB/s	24.6%	17.5%
mg	17.18 GB/s	20.32 GB/s	18.2%	15.5%
sp	19.80 GB/s	24.11 GB/s	21.7%	17.8%
STREAM	28.21 GB/s	32.61 GB/s	15.6%	13.4%
Configuration G				
Benchmark	**64 B**	**512 B**	**BW Increase**	**Speedup**
fluidanimate	3.34 GB/s	3.41 GB/s	2.1%	1.5%
mcol	15.62 GB/s	28.80 GB/s	84.3%	43.1%
mg	16.51 GB/s	20.19 GB/s	22.3%	18.3%
sp	19.09 GB/s	24.86 GB/s	30.2%	23.2%
STREAM	27.23 GB/s	33.56 GB/s	23.3%	18.8%
Configuration I				
Benchmark	**64 B**	**512 B**	**BW Increase**	**Speedup**
fluidanimate	3.17 GB/s	3.17 GB/s	0.0%	0.0%
mcol	19.49 GB/s	34.62 GB/s	77.6%	41.4%
mg	16.29 GB/s	17.63 GB/s	8.2%	7.4%
sp	18.92 GB/s	19.85 GB/s	4.9%	4.7%
STREAM	17.04 GB/s	17.04 GB/s	0.0%	0.0%

Table 21: The difference in bandwidth when increasing the return queue capacity in each configuration from one (64 bytes) to eight (512 bytes)

This can be further quantified by examining statistics about the read return queue in each configuration during program execution. For example during the *STREAM* benchmark the return queue is full (and thus preventing new requests from being issued) 72.5% of the time when the queue capacity is one (64 bytes) while configurations D and I must stall 71.2% and 36.0% of the time, respectively. Once the queue depth is two (128 bytes), the amount of time in which the DRAM is stalled is drastically reduced, yet configuration G still stalls more frequently – 28.3% of the time compared to 27.8% and 5.2% in configurations D and I, respectively. With a capacity of eight, each configuration stalls as little as 0.01%. Even in a read dominant

benchmark like *mcol*, a capacity of eight is sufficient and DRAM operation stalls less than 1% in configuration D, whose performance was the worst across all configurations during this benchmark. The performance difference beyond this capacity would be negligible and would not be worth the cost of increasing the capacity; therefore a total capacity of eight is ideal.

4.2.5 Port Parameters & Heuristics

There are three main parameters when discussing port configurations: the number of independent ports, the width of each port, and the storage capacity of the input and output buffer attached to each port. The number of ports corresponds to the number of independent buses which can be written to or read from on the same cycle. The width of a port dictates how many bytes of data and packet overhead it may move on each CPU cycle (either from a single beat of a wide bus or from dual-edged data transfer). The buffer depth for both the input and output bus of each port corresponds to how many entries that buffer can hold, regardless of what type of packet. The frequency of each port is dictated by the CPU clock, and is set to 3.2 GHz for these simulations. As shown in the limit-case simulations, the decisions made about these parameters can have a drastic impact on the overall system performance. Examining the organization of the ports during a full system simulation will give a complete picture about the interaction the ports have with the rest of the system.

Along with BOB configurations D, G, and I, configuration A will be used in the examination of the ports as this will include all degrees of multi-channel configurations, and therefore all possible numbers of link buses for a system with eight DRAM

channels. This will ensure that any interaction between port configurations and link bus configurations will be uncovered. The number of ports, the width of each port, and the capacity of each port buffer is changed within these configurations during the execution of *STREAM* and the resulting average bandwidth can be seen in **Figures 77** through **80**. In each of these figures, the port configuration is referenced as XxYB where X is the number of independent ports and the Y is the number of bytes that port can move on a single CPU cycle (i.e., the width). The port buffer depth is the number of entries that buffer can hold until it has reached maximum capacity. **Tables 22** through **25** show the same results as each of the corresponding graphs, but groups the configurations with similar resource usage by color to make comparing configurations easier.

Each parameter in the configuration of the ports has a drastically different impact on the achieved performance. The relative impact of each parameter is determined by the link bus organization in the attached BOB system, although, across all configurations, increasing the port buffer depth will always increase performance by as much as 45%. An increased buffer capacity gives the main BOB controller a greater pool of requests with which to schedule in the event that some requests must be stalled. This will also allow the CPU to issue a greater number of requests as the port buffer is the initial place requests are stored once issued. In some extreme cases performance will begin to drop – this will be discussed later.

In configurations with only two link buses (D and I), parameters such as the width and the number of ports do not have a significant impact on the performance relative to the capacity of the input and output buffer. This is because with only two

link buses, there is a greater likelihood that each link bus will already be occupied sending requests. In these situations, there are some clear optimal decisions which can be made about how ports should be configured. For example, if a system is provided with resources to be used as data lanes and buffer space in port configurations, a greater number of independent ports will perform better than if these resources are used for fewer, wider ports. This is because the limited bandwidth of the request link bus will not be able to utilize the additional port bandwidth, if increased. Such behavior can be seen in configuration I (**Figure 80**) where a single port with a width of 32 bytes and a buffer capacity of eight (1x32Bx8) achieves 15.52 GB/s which is worse than the performance of two ports with 16 bytes of width and a storage capacity of four for each (2x16Bx4), which achieves 15.55 GB/s. Both of these configurations use the same resources for buffer space and data lanes, but one performs better than the other as a result of increased parallelism (as opposed to increased bandwidth). While the difference in performance is marginal, this is uniformly the case across all tested configurations.

Another conclusion that can be drawn from these results is that a port width of eight bytes is almost universally worse when using a fixed amount of resources. For example in configuration D, 4x8Bx8 achieves 32.57 GB/s while 2x16Bx16 achieves 32.59 GB/s; both of these port configurations use the same resources. This is due to the increased time necessary to transmit both write request packets and read response packets (both 72 bytes); with an 8 byte port this would require 9 CPU cycles, and will stall subsequent requests for an inordinate amount of time.

Conversely, when there is a higher level of link bus parallelism, as in configuration

A (eight link buses) and G (four link buses), a greater benefit can be seen when increasing both the number of ports or the width of a port. This is because there is a greater possibility that a link bus will be idle and able to receive a request to send to the DRAM. Therefore, a wider port performs better because it is capable of transmitting packets to idle link buses in less time (especially write requests whose packets are 72 bytes). An example of this can be see in configuration A (**Figure 77**); of all possible combinations of 32 bytes worth of lanes and 16 entries worth of buffer storage (i.e., 1x32Bx16, 2x16Bx8, and 4x8Bx4), the configuration which performs the best is a single monolithic 32 byte port with a 16 entry buffer. The differences in performance between these port configurations is only 2.3% with port configuration 1x32Bx16 achieving an average bandwidth of 46.73 GB/s, 2x16Bx8 achieving 46.01 GB/s, and 4x8Bx4 achieving 45.64 GB/s, yet this principle applies to all other configurations possible with a fixed amount of resources.

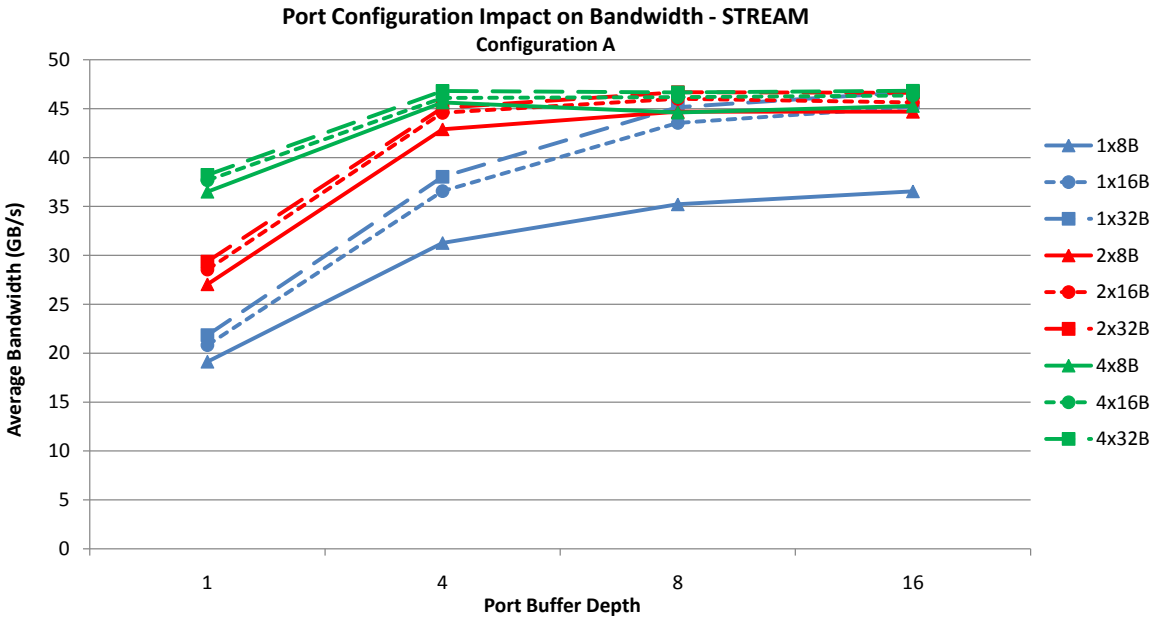


Figure 77: The impact of various port configurations in configuration A running the *STREAM* benchmark

Configuration A						
# Ports	Width	Depth	Total Storage	Total Width	BW	Max
1	32	1	1	32	**21.86**	-
1	32	4	4	32	**38.05**	X
1	32	8	8	32	**45.15**	X
1	32	16	16	32	**46.73**	X
2	16	1	2	32	**28.56**	-
2	16	4	8	32	**44.58**	-
2	16	8	16	32	**46.01**	-
2	16	16	32	32	**45.63**	X
4	8	1	4	32	**36.52**	-
4	8	4	16	32	**45.64**	-
4	8	8	32	32	**44.61**	-
4	8	16	64	32	**45.29**	-

Table 22: Various port organizations in configuration A colored by similar resources

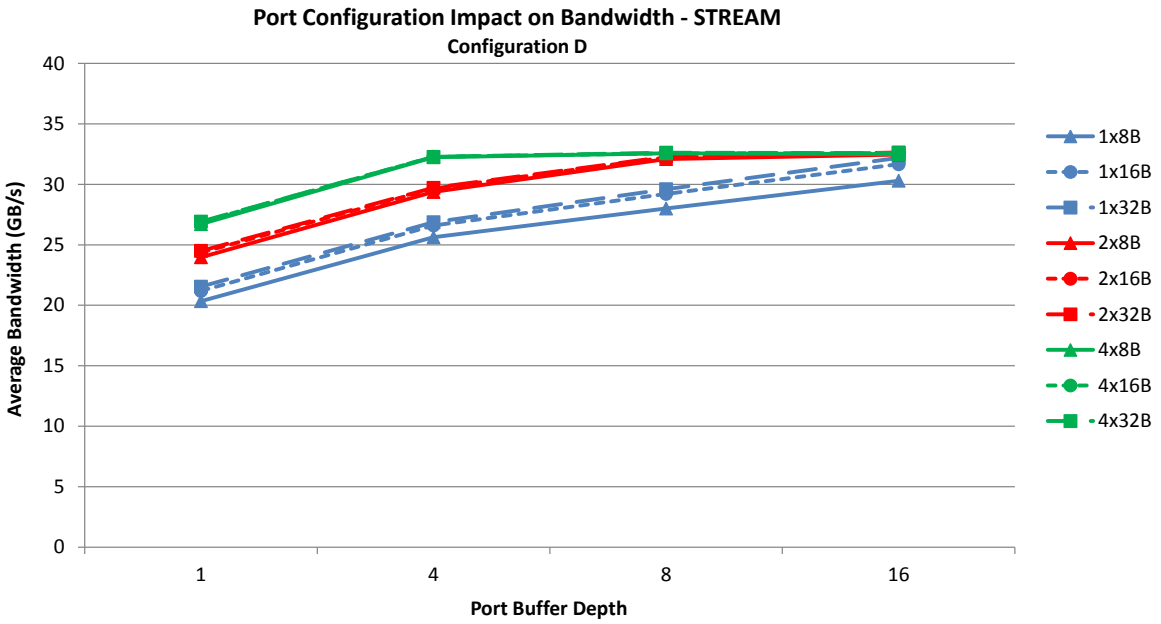


Figure 78: The impact of various port configurations in configuration D running the *STREAM* benchmark

Configuration D						
# Ports	Width	Depth	Total Storage	Total Width	BW	Max
1	32	1	1	32	**21.54**	-
1	32	4	4	32	**26.87**	X
1	32	8	8	32	**29.60**	-
1	32	16	16	32	**32.19**	-
2	16	1	2	32	**24.35**	-
2	16	4	8	32	**29.65**	X
2	16	8	16	32	**32.23**	-
2	16	16	32	32	**32.59**	X
4	8	1	4	32	**26.71**	-
4	8	4	16	32	**32.24**	X
4	8	8	32	32	**32.57**	-
4	8	16	64	32	**32.5**	-

Table 23: Various port organizations in configuration D colored by similar resources

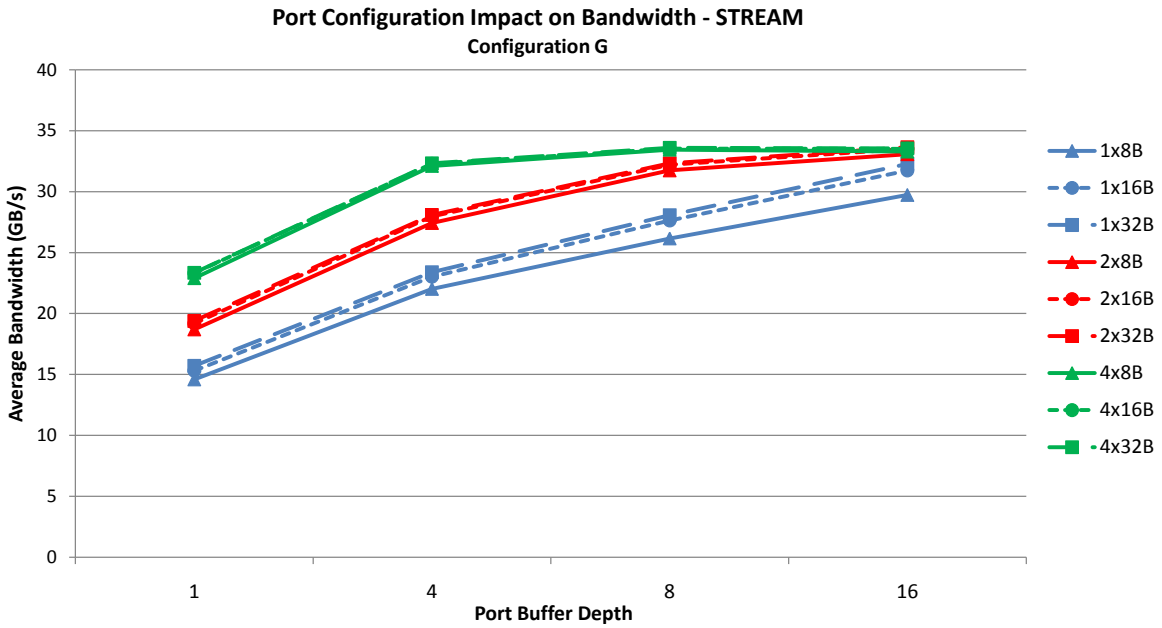


Figure 79: The impact of various port configurations in configuration G running the *STREAM* benchmark

Configuration G						
# Ports	**Width**	**Depth**	**Total Storage**	**Total Width**	**BW**	**Max**
1	32	1	1	32	**15.71**	-
1	32	4	4	32	**23.38**	X
1	32	8	8	32	**28.06**	X
1	32	16	16	32	**32.27**	X
2	16	1	2	32	**19.19**	-
2	16	4	8	32	**27.97**	-
2	16	8	16	32	**32.21**	-
2	16	16	32	32	**33.51**	X
4	8	1	4	32	**22.90**	-
4	8	4	16	32	**32.11**	-
4	8	8	32	32	**33.45**	-
4	8	16	64	32	**33.29**	-

Table 24: Various port organizations in configuration G colored by similar resources

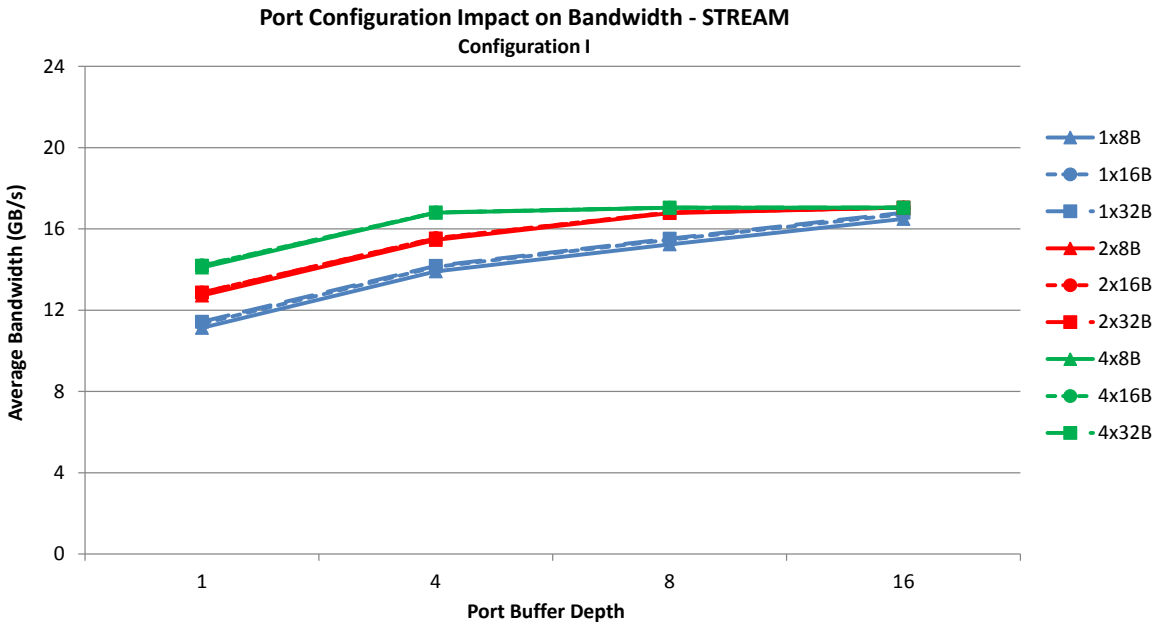


Figure 80: The impact of various port configurations in configuration I running the *STREAM* benchmark

Configuration I						
# Ports	Width	Depth	Total Storage	Total Width	BW	Max
1	32	1	1	32	**11.43**	-
1	32	4	4	32	**14.18**	X
1	32	8	8	32	**15.52**	-
1	32	16	16	32	**16.80**	-
2	16	1	2	32	**12.87**	-
2	16	4	8	32	**15.55**	X
2	16	8	16	32	**16.81**	X
2	16	16	32	32	**17.05**	X
4	8	1	4	32	**14.10**	-
4	8	4	16	32	**16.80**	-
4	8	8	32	32	**17.04**	-
4	8	16	64	32	**17.05**	-

Table 25: Various port organizations in configuration I colored by similar resources

As stated above, increasing the port buffer depth typically increases performance due to a greater number of requests that are capable of being scheduled. In some cases, though, performance is shown to decrease marginally. An example of this can be seen in configuration G when increasing the buffer depth of port configuration 4x8B from 8 to 16 entries; the performance decreases slightly from 33.45 GB/s to 33.29 GB/s (0.5%). The reason for this behavior is the heuristic used to add requests to the ports during these simulations. All prior full-system simulations have used a heuristic where the first available port (by index) is used to add a request to the memory system. This particular heuristic can lead to an over-utilization of ports with lesser indices and subsequently a backup in that port's input buffer. Therefore when increasing the capacity of the input buffer, a greater number of requests must wait to be issued and performance is decreased.

Heuristics for adding requests to the main BOB controller's ports include :

- **First Available** (FA) - Ports are searched in order of increasing index for the first available port
- **Per Core** (PC) - Each CPU core (or a subset of cores) is assigned a particular port to issue requests
- **Round Robin** (RR) - Requests are added to all ports in a round-robin fashion
- **Least Full** (LF) - Requests are added to the port that has the input buffer with the fewest number of entries

In these heuristics, a port is "available" when it is not currently being written to

and has space in its input buffer. If a request is issued to a port and is rejected as a result of not being available, the request stays within the last-level cache and tries again on the next CPU cycle. The heuristics above have a range of implementation complexities. The *per core* heuristic is the simplest to implement and simply assigns a particular core (or set of cores) to always communicate with the same port. The *first available* heuristics is slightly more complicated in that it requires logic to search incrementally over all ports. The *round robin* and *least full* require the most amount of state and logic to implement but are the most fair heuristics.

Each of these heuristics are used while performing full-system simulations of various benchmarks on each BOB configuration. Each BOB configuration uses port configuration 4x16B with a buffer capacity of eight. The average bandwidth and the average number of transactions in each of the four ports during execution can be seen in **Tables 26** through **29**. Rows highlighted in yellow indicate heuristics which performed the best for that particular benchmark and BOB configuration.

The most obvious and apparent impact that each of these heuristics has is in the buffer utilization during program execution (as shown as the average number of entries in each buffer in the tables below). The *round robin* and *least full* heuristics typically provide the most even spread of requests over all available buffers, regardless of the benchmark. The *per core* heuristic can evenly spread requests as well as, yet is entirely dependent on the benchmark being executed. For example, during the *STREAM* benchmark, this heuristic more evenly spreads requests than all others, yet during *Sandia GUPS*, it does not issue requests to three of the available ports for over 95% of the execution of the benchmark. Conversely, the *first available* heuristic

will always utilize a particular buffer more than others.

With such drastic differences in buffer utilization, one would expect that these heuristics should also have a meaningful impact on overall system performance. The results clearly show otherwise, and that there is an insignificant difference in the achieved bandwidth between each method used to add requests to each port. The largest difference between heuristics is 3.5% in BOB configuration I while executing *Sandia GUPS*, yet the average difference across all benchmarks and configurations is only 1.2%.

The main reason for such a minuscule difference in performance is both the drastic difference in bandwidth of the 16 byte wide ports and the link buses, and the fact that port buffers are searched out of order to find a viable request packet. A port that is 16 bytes wide and operates at a frequency of 3.2 GHz has a bandwidth of 51.2 GB/s and can transmit a read request packet (8 bytes) in a single cycle and a write request packet (72 bytes) in 5 cycles. Therefore, even if transactions are added to ports in a manner which does not evenly spread packets across the available buffers, the ports are capable of evacuating packets from their buffers fast enough to prevent it from being a detriment to performance.

With write packets taking multiple cycles to transmit, write-heavy benchmarks are slightly more susceptible to performance variations between heuristics. An example of this can be seen during *Sandia GUPS* which has the largest relative difference between different heuristics (3.5%) and has a request stream of 97% writes during the entirety of the benchmark. Regardless, the relative difference can still be considered insignificant.

BOB Configuration A					
STREAM		**Port Index**			
Heuristic	**Achieved Bandwidth**	**0**	**1**	**2**	**3**
PC	46.43	1.645	1.681	1.690	1.697
FA	46.37	4.302	2.268	0.451	0.050
RR	46.38	1.696	1.689	1.661	1.660
LF	46.35	1.931	1.768	1.578	1.391
mcol		**Port Index**			
Heuristic	**Achieved Bandwidth**	**0**	**1**	**2**	**3**
PC	30.69	0.052	0.054	0.053	0.053
FA	30.34	0.182	0.028	0.000	0.000
RR	30.72	0.053	0.052	0.053	0.053
LF	30.54	0.096	0.085	0.019	0.006
sp.C		**Port Index**			
Heuristic	**Achieved Bandwidth**	**0**	**1**	**2**	**3**
PC	26.08	0.759	0.758	0.791	0.776
FA	26.06	2.133	0.882	0.169	0.018
RR	26.16	0.758	0.778	0.775	0.764
LF	26.10	0.980	0.841	0.689	0.524
Sandia GUPS		**Port Index**			
Heuristic	**Achieved Bandwidth**	**0**	**1**	**2**	**3**
PC	10.89	1.001	0.00	0.00	0.00
FA	11.10	0.779	0.366	0.068	0.003
RR	10.99	0.302	0.299	0.300	0.300
LF	11.11	0.360	0.314	0.263	0.224

Table 26: Average bandwidth and number of requests waiting across all input port buffers

BOB Configuration D					
STREAM		**Port Index**			
Heuristic	**Achieved Bandwidth**	**0**	**1**	**2**	**3**
PC	32.62	5.959	5.939	6.004	6.007
FA	32.59	7.602	7.391	6.709	5.302
RR	32.61	6.801	6.768	6.765	6.784
LF	32.60	6.902	6.788	6.648	6.510
mcol		**Port Index**			
Heuristic	**Achieved Bandwidth**	**0**	**1**	**2**	**3**
PC	20.51	0.056	0.056	0.055	0.056
FA	21.10	0.210	0.020	0.000	0.000
RR	20.55	0.056	0.057	0.056	0.057
LF	20.59	0.068	0.131	0.021	0.008
sp.C		**Port Index**			
Heuristic	**Achieved Bandwidth**	**0**	**1**	**2**	**3**
PC	24.06	3.669	3.716	3.678	3.632
FA	24.13	6.544	5.659	2.836	0.652
RR	24.03	4.009	3.994	4.082	4.071
LF	24.03	4.255	4.033	3.807	3.577
Sandia GUPS		**Port Index**			
Heuristic	**Achieved Bandwidth**	**0**	**1**	**2**	**3**
PC	9.48	3.320	0.00	0.00	0.00
FA	9.64	3.061	2.735	1.805	0.128
RR	9.64	2.026	2.028	1.983	1.988
LF	9.63	2.116	2.009	1.898	1.777

Table 27: Average bandwidth and number of requests waiting across all input port buffers

BOB Configuration G					
STREAM		**Port Index**			
Heuristic	**Achieved Bandwidth**	**0**	**1**	**2**	**3**
PC	33.41	5.647	5.668	5.778	5.763
FA	33.56	7.534	7.235	6.435	5.035
RR	33.58	6.580	6.586	6.604	6.591
LF	33.54	6.755	6.634	6.479	6.329
mcol		**Port Index**			
Heuristic	**Achieved Bandwidth**	**0**	**1**	**2**	**3**
PC	27.84	0.101	0.099	0.102	0.100
FA	28.76	0.381	0.044	0.001	0.000
RR	27.89	0.098	0.097	0.097	0.097
LF	27.98	0.113	0.202	0.054	0.017
sp.C		**Port Index**			
Heuristic	**Achieved Bandwidth**	**0**	**1**	**2**	**3**
PC	24.82	3.536	3.495	3.496	3.912
FA	24.89	6.586	5.435	2.725	0.649
RR	24.95	3.687	3.653	3.749	3.681
LF	24.91	4.064	3.858	3.626	3.395
Sandia GUPS		**Port Index**			
Heuristic	**Achieved Bandwidth**	**0**	**1**	**2**	**3**
PC	9.56	3.252	0.00	0.00	0.00
FA	9.61	2.925	2.522	1.286	0.071
RR	9.70	1.779	1.733	1.787	1.737
LF	9.73	1.901	1.802	1.704	1.600

Table 28: Average bandwidth and number of requests waiting across all input port buffers

BOB Configuration I					
STREAM		**Port Index**			
Heuristic	**Achieved Bandwidth**	**0**	**1**	**2**	**3**
PC	17.07	7.781	7.762	7.770	7.759
FA	17.04	7.958	7.952	7.941	7.925
RR	17.04	7.937	7.937	7.939	7.938
LF	17.04	7.948	7.944	7.940	7.936
mcol		**Port Index**			
Heuristic	**Achieved Bandwidth**	**0**	**1**	**2**	**3**
PC	35.22	0.278	0.280	0.287	0.279
FA	34.74	0.941	0.135	0.002	0.00
RR	35.25	0.258	0.262	0.260	0.262
LF	35.21	0.312	0.401	0.219	0.113
sp.C		**Port Index**			
Heuristic	**Achieved Bandwidth**	**0**	**1**	**2**	**3**
PC	19.87	6.955	6.923	6.925	6.945
FA	19.82	7.752	7.629	7.300	6.469
RR	19.86	7.291	7.282	7.249	7.268
LF	19.91	7.425	7.346	7.243	7.141
Sandia GUPS		**Port Index**			
Heuristic	**Achieved Bandwidth**	**0**	**1**	**2**	**3**
PC	7.34	5.174	0.00	0.00	0.00
FA	7.59	5.020	4.602	3.695	1.563
RR	7.57	3.761	3.782	3.753	3.787
LF	7.52	3.821	3.668	3.505	3.271

Table 29: Average bandwidth and number of requests waiting across all input port buffers

In conclusion, the port configuration has been shown to have a marginal impact on the overall system performance during full-system simulations. Optimal organizations are dictated by other aspects of the system; BOB systems with fewer link buses tend to benefit more from resources being used as multiple, independent ports, while BOB configurations with a great amount of link bus parallelism benefit more from ports which have a greater bandwidth. The mechanism employed to add requests to the ports was shown to have little impact on system performance and the simplest

or easiest to implement heuristic could be chosen with little impact to achievable performance.

5 Conclusion

After over 15 years of widespread adoption, the commodity memory system's inability to scale, both in operating speed and overall capacity, has caused the memory bottleneck to become one of the largest hindrances to system performance. These limitations are brought about by the physical contact based electrical connections that each DIMM uses to communicate with the rest of the system. As the memory clock is increased to keep pace with the CPU clock, the signal integrity seen at each of these physical contacts is significantly reduced. This problem is also exacerbated as more DIMMs are attached to a particular channel. As a result, system manufacturers are forced to reduce the total number of DIMMs allowed in a system as they increase the memory clock which necessarily reduces both capacity and available concurrency. This fact paired with the inability to increase single DIMM capacity without unreasonably increasing its cost has caused serious issues that must be solved in order to fully utilize a system's computational ability.

The FB-DIMM standard was introduced in 2004 in an attempt to solve many of the problems facing modern memory system design. By placing logic called the advanced memory buffer (AMB) on each DIMM, the CPU could now communicate with the memory system via a fast and narrow bus instead of a slow, wide one. This granted both an increase in clock rate and signal integrity seen at the DRAM devices, allowing an increase in overall possible capacity. Each of these narrow buses were logically separated into channels called the northbound and southbound bus, where requests and responses were packetized into frames and sent over multiple

clock cycles. The AMB on each DIMM was responsible for interpreting these frames and routing requests and responses accordingly.

Unfortunately, unexpected problems arose out of this design which eventually led to its failure. These problems included excessive heat and power dissipation as a result of the AMB's high-speed IO, variable request latency due to the chained nature of the DIMMs, and relatively expensive DIMMs compared to similar capacity DDR modules (caused by the addition of the AMB and its heat spreader). The standard was eventually removed from vendor road-maps, and system designers were forced to implement a new architecture that could finally give them both the increase in speed and capacity that their systems and applications demand.

By taking the lessons learned from FB-DIMM, vendors such as Intel, IBM, and AMD have implemented a new memory architecture which also places logic between the CPU and DIMMs and communicates via logically separate, fast and narrow buses. The key differentiation between this buffer-on-board design and FB-DIMM is that the new logic is not responsible for communicating with other logic nor is it chained together. Instead, it is only used to control the attached DIMMs and communicate with the CPU. The DIMMs used in this system are standard DDR modules (U-DIMMs, R-DIMMs, or LR-DIMMs) which also reduces costs relative to an FB-DIMM system. All of the above mentioned vendors have implemented such memory systems, yet each system varies in their specifics, dictating the need for an exploration of this new design.

To fully explore this new memory architecture, a cycle accurate and hardware verified simulator was developed in order to characterize the behavior of all aspects

of the system, and determine optimal use of the resources involved based on outside constraints. Two types of simulations were performed: a limit-case simulation where requests are issued directly to the memory system as fast as possible and a full-system simulation where the simulator interacts with the CPU, cache, operating system, and application. To do this, MARSSx86 was integrated with the BOB simulator, and a variety of multi-threaded benchmarks were executed.

From both of these types of simulations, basic principles and optimizations about the buffer-on-board system's design were discovered. This includes :

- Confirmation that insights and optimizations which apply to commodity memory systems also apply to the DRAM attached to each individual BOB channel and the simple controller which operates it. This includes the importance of a proper address mapping scheme to fully utilize the parallelism available in each device. Also, the importance of adequate queue depths necessary to maintain peak DRAM efficiency. Lastly, the negative impact that numerous ranks has on the utilization of the DRAM data bus caused by the necessity to idle when switching between ranks.

- Insights into the importance of proper configuration of each BOB channel's request and response link bus. The efficiency of the DRAM is key in the system's performance and each link bus must be configured in such a way as to not impede efficiency. This includes enough bandwidth to provide a sufficient number of requests to fully utilize the DRAM bus as well as the ability to remove responses quickly enough so as not to stall DRAM operation. Unfortunately, it

was clear that the read-to-write request ratio had the biggest impact on what was considered to be an optimal configuration. A set of formulas were developed to determine the proper bandwidth necessary to maintain maximum attainable efficiency based on read-write ratio and DIMM type.

- The realization that the total possible system bandwidth is dictated only by the types of DIMM which populate each channel and the total number of independent channels in the system. When the rest of the system has been configured in such a way so that it is possible to achieve optimal performance, the peak bandwidth is simply a product of the number of channels the achievable peak bandwidth of the DIMM which occupies these channels.

- Outlining a key optimization which can reduce system costs while maintaining performance and overall capacity. Referred to as the multi-channel optimization, this concept is based on the fact that a link bus can provide far more bandwidth than a DIMM needs to reach maximum attainable efficiency. Therefore, some link bus configurations can support multiple DRAM channels without negatively impacting system performance thereby saving on significantly system costs such as CPU pin-out, physical space, and simple controller fabrication costs.

- Understanding the proper configuration and use of the main BOB controller. This includes optimal channel mappings which are necessary to evenly spread out requests over all available channels. This is essential since over-loading a particular channel with requests will significantly reduce overall performance.

Also, an understanding about the impact the main BOB controller's ports has on the movement of requests and responses and the subsequent performance.

5.1 Future Work

The buffer-on-board memory architecture is relatively new. While current implementations already alleviate many of the issues facing the commodity memory system, it is clear that this architecture provides the possibility for optimizations or functionality which could benefit other parts of the system as well. The introduction of the logic provides a chance for capabilities which were not possible before and could improve both efficiency and performance of many other parts of the system.

The most straight-forward optimization to the regular BOB system could increase performance while not requiring any modifications to other parts of the system. This is the addition of a cache within each simple controller. This could reduce DRAM access times, power consumption (in the DRAM), and contention on the DRAM bus. Granted, adding a cache to the simple controller would increase power consumption and transistor count so it remains to be seen whether these benefits outweigh the additional costs. It is possible that the SRAM based cache would require significant amount of power resulting in the same issues as FB-DIMM or that the concurrency in a BOB memory system results in improbable address space locality. Further analysis would need to be done to determine whether or not this is feasible and/or beneficial.

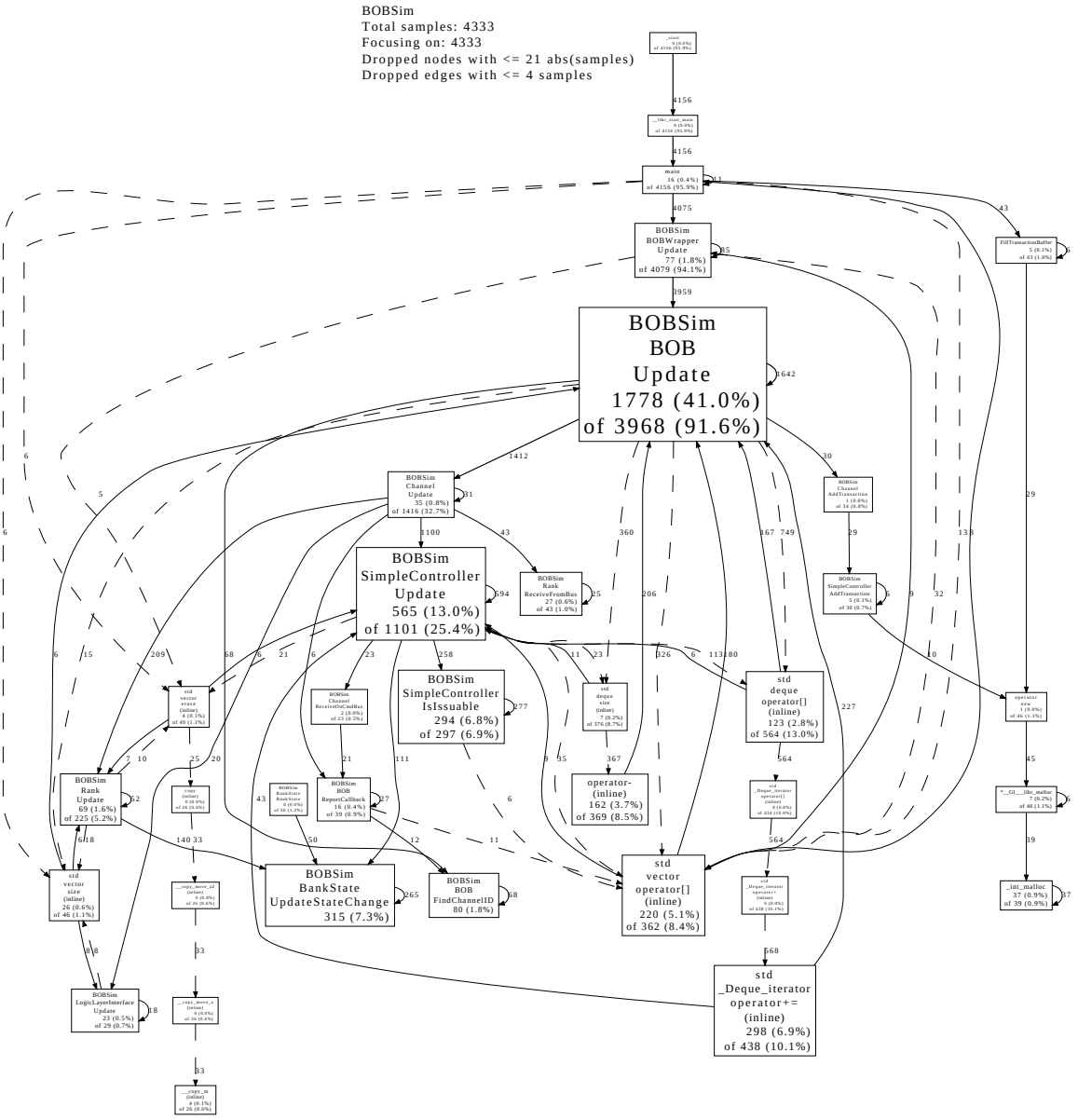
Another more complex modification could be the addition of features within the operating system so as to be aware of the BOB aspects of the memory system. For

instance, the operating system could map instruction memory to a BOB channel that has a wider link bus relative to other channels to ensure quick access to subsequent instructions. This could be expanded to allow direct memory access (DMA) from the network or other peripherals directly to a simple controller. This would reduce traffic through the CPU and cache while providing faster access times to the other parts of the system.

The most involved modification to the system would be the addition of active memory operations (AMOs). With this, each simple controller would have the ability to perform logic operations along with its standard memory controller functionality. Offloading work to the memory controller is not a new idea but this new architecture circumvents many of the issues facing AMOs in the past. Having logic so relatively close to the memory prevents contention, lowers CPU utilization, and reduces latency of certain requests. Unfortunately, this would require changes to the operating system, CPU, cache, and simple controllers, making it non-trivial to implement, both in simulators and in actual systems.

Regardless of whether or not these additional features are added to a BOB memory system, it has been shown to be a viable solution to many of the issues facing modern memory architectures. It allows an increase in clock rate, an increase in overall capacity, while decreasing pin count required by the CPU. Accurate simulation and modeling have shown this and provided insights into optimal implementations and organizations in hopes that future systems will be even better than the ones already in use.

6 Appendix A


BOBSim
Total samples: 4333
Focusing on: 4333
Dropped nodes with <= 21 abs(samples)
Dropped edges with <= 4 samples
main
15 (0.4%)
of 4156 (95.9%)
BOBSim
BOBWrapper
Update
77 (1.8%)
of 4079 (94.1%)
BOBSim
BOB
Update
1778 (41.0%)
of 3968 (91.6%)
BOBSim
Channel
Update
35 (0.8%)
of 1416 (32.7%)
BOBSim
SimpleController
Update
565 (13.0%)
of 1101 (25.4%)
BOBSim
SimpleController
IsIssuable
294 (6.8%)
of 297 (6.9%)
BOBSim
Rank
Update
69 (1.6%)
of 225 (5.2%)
BOBSim
BankState
UpdateStateChange
315 (7.3%)
BOBSim
BOB
FindChannelID
80 (1.8%)
BOBSim
Rank
ReceiveFromBus
27 (0.6%)
of 43 (1.0%)
BOBSim
BOB
ReportCallback
16 (0.4%)
of 39 (0.9%)
BOBSim
LogicLayerInterface
Update
23 (0.5%)
of 29 (0.7%)
std
deque
operator[]
(inline)
123 (2.8%)
of 564 (13.0%)
std
vector
operator[]
(inline)
220 (5.1%)
of 362 (8.4%)
operator-
(inline)
162 (3.7%)
of 369 (8.5%)
std
_Deque_iterator
operator+=
(inline)
298 (6.9%)
of 438 (10.1%)
std
vector
size
(inline)
26 (0.6%)
of 46 (1.1%)
std
vector
erase
(inline)
4 (0.1%)
of 49 (1.1%)
std
deque
size
(inline)
7 (0.2%)
of 376 (8.7%)
FillTransactionBuffer
5 (0.1%)
of 43 (1.0%)
operator
new
1 (0.0%)
of 46 (1.1%)
_int_malloc
37 (0.9%)
of 39 (0.9%)
BOBSim
Channel
AddTransaction
1 (0.0%)
of 34 (0.8%)
BOBSim
SimpleController
AddTransaction
5 (0.1%)
of 30 (0.7%)

References

[1] Calculating Memory System Power for DDR - TN-46-03. Technical Note, Micron Technology, Inc., 2001.

[2] Calculating Memory System Power For DDR2 - TN-47-04. Technical Note, Micron, Inc., 2005.

[3] FBDIMM - Channel Utilization (Bandwidth and Power) - TN-47-21. Technical Note, Micron Technology, Inc., 2006.

[4] Calculating Memory System Power for DDR3 - TN41-01. Technical Note, Micron Technology, Inc., 2007.

[5] DDR3 Power Estimates, Affect of Bandwidth, and Comparisons to DDR2. Technical report, Micron Technology Inc., April 2007.

[6] FBDIMM : Architecture and Protocol - JESD206. ISO, JEDEC Solid State Technology Association, January 2007.

[7] Low-Power Fully Buffered DIMM. Whitepaper, Netlist Inc., 51 Discovery, Suite 150, Irvine, CA, 2007.

[8] Netlist FBDIMM preliminary power analysis Training. Technical report, Netlist, September 2007.

[9] DDR2 SDRAM FBDIMM. Datasheet, Micron Technology, Inc., 2008.

[10] FBDIMM Advanced Memory Buffer (AMB) - JESD82-20A. ISO, JEDEC Solid State Technology Association, March 2009.

[11] DDR3 SDRAM Specification. Technical report, Association, JEDEC Solid State Technology, July 2010.

[12] IBM Power 795 Technical Overview and Introduction. Datasheet, IBM, September 2010.

[13] Intel 7500 Scalable Memory Buffer. Technical report, Intel, March 2010.

[14] Intel CoreTM i7-900 Desktop Processor Series. Technical report, Intel, February 2010.

[15] *Intel® Xeon® Processor 7500 Series*, March 2010.

[16] R. J. Baker. *CMOS: Circuit Design, Layout, and Simulation.* IEEE Press, 2nd edition, 2005.

[17] Jason Clark and Ross Whitehead. Lower Power Server CPU Shoot-out. Technical report, AnandTech, July 2007.

[18] E. Cooper-Balis and B. Jacob. Fine-Grained Activation for Power Reduction in DRAM. *Micro, IEEE*, 30(3):34–47, 2010.

[19] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. High-performance DRAMs in workstation environments. *IEEE Transactions on Computers*, pages 1133–1153, 2001.

[20] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge. A Performance Comparison of Contemporary DRAM Architectures. In *Proc. 26th Annual In-*

ternational Symposium on Computer Architecture (ISCA'99), pages 222–233, Atlanta GA, may 1999. Published by the IEEE Computer Society.

[21] Henrik Fredriksson and Christer Svensson. Improvement Potential and Equalization Example for Multidrop DRAM Memory Buses. *IEEE Transaction On Advanced Packaging*, 32(3):675–682, 2009.

[22] B. Ganesh, A. Jaleel, D. Wang, and B. Jacob. Fully-buffered DIMM memory architectures: Understanding mechanisms, overheads and scaling. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 109–120, 2007.

[23] Micron Inc. TN-47-21 : FBDIMM - Channel Utilization (Bandwidth and Power). Technical Note, 2006.

[24] Bruce Jacob, Spencer W. Ng, and David T. Wang. *Memory Systems : Cache, DRAM, Disk*. Morgan Kaufmann, 2008.

[25] Ron Kalla, Balaram Sinharoy, William J. Starke, and Michael Floyd. Power7: IBM's Next-Generation Server Processor. *IEEE Micro*, 30(2):7–15, 2010.

[26] B. Keeth, R. J. Baker, B. Johnson, and F. Lin. *DRAM Circuit Design: Fundamental and High-Speed Topics*. IEEE Press, 2008.

[27] Mark LaPedus. Micron rolls DDR3 LRDIMM. *EE Times*, 2009.

[28] Inc. Micron. DDR3 SDRAM MT41J512M4 Datasheet. Online, 2006.

[29] Inc. Micron. DDR3 SDRAM RDIMM MT72JS(Z)S1G72PZ. online, 2009.

[30] Inc. Micron. DDR3L 1.35V SDRAM LRDIMM MT72KSZS2G72LZ. online, 2010.

[31] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSSx86: A Full System Simulator for x86 CPUs. In *Design Automation Conference 2011 (DAC'11)*, 2011.

[32] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters*, 99(RapidPosts), 2011.

[33] Samsung. SAMSUNG Develops Industry's First DDR4DRAM, Using 30nm Class Technology. Press Release, Jan 2011.

[34] Jun Shao and Brian T. Davis. The bit-reversal SDRAM address mapping. In *Proceedings of the 2005 workshop on Software and compilers for embedded systems*, SCOPES '05, pages 62–71, New York, NY, USA, 2005. ACM.

[35] Gotou Shigehiro. AMD's Next Server Platform "Maranello". *PC Watch*, 2008.

[36] Sadagopan Srinivasan. *Prefetching vs The Memory System : Optimizations for Multi-Core Server Platforms*. PhD thesis, University of Maryland, 2007.

[37] Javier Suarez. Enterprise X-Architecture 5th Generation. Technical report, March 2010.

[38] D. Burger W. Lin, S. Reinhardt. Reducing DRAM latencies with an Integrated Memory Heirarchy Design. January 2001.

[39] David Tawei Wang. *Modern DRAM Memory Systems : Performance Analysis and a High Performance, Power-Constrained DRAM Scheduling Algorithm*. PhD thesis, University of Maryland, 2005.

[40] Steve Woo. DRAM and Memory System Trends. October 2004.

[41] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. pages 23–34, 2007.

[42] Z. Zhang Z. Zhu. A Performance Comparison of DRAM Memory System Optimzations for SMT Processors. Februrary 2005.