Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM

Donghyuk Lee* Lavanya Subramanian* Rachata Ausavarungnirun* Jongmoo Choi† Onur Mutlu*

*Carnegie Mellon University †Dankook University

{donghyu1, lsubrama, rachata, onur}@cmu.edu choijm@dankook.ac.kr

***Abstract*—Memory channel contention is a critical performance bottleneck in modern systems that have highly parallelized processing units operating on large data sets. The memory channel is contended not only by requests from different user applications (*CPU access*) but also by system requests for peripheral data (*IO access*), usually controlled by *Direct Memory Access* (*DMA*) engines. Our goal, in this work, is to improve system performance by eliminating memory channel contention between CPU accesses and IO accesses.**

To this end, we propose a hardware-software cooperative data transfer mechanism, *Decoupled DMA* (*DDMA*) that provides a specialized low-cost memory channel for IO accesses. In our DDMA design, main memory has two independent data channels, of which one is connected to the processor (*CPU channel*) and the other to the IO devices (*IO channel*), enabling CPU and IO accesses to be served on different channels. System software or the compiler identifies which requests should be handled on the IO channel and communicates this to the DDMA engine, which then initiates the transfers on the IO channel. By doing so, our proposal increases the effective memory channel bandwidth, thereby either accelerating data transfers between system components, or providing opportunities to employ IO performance enhancement techniques (e.g., aggressive IO prefetching) without interfering with CPU accesses.

We demonstrate the effectiveness of our DDMA framework in two scenarios: *(i)* CPU-GPU communication and *(ii)* in-memory communication (bulk data copy/initialization within the main memory). By effectively decoupling accesses for CPU-GPU communication and in-memory communication from CPU accesses, our DDMA-based design achieves significant performance improvement across a wide variety of system configurations (e.g., 20% average performance improvement on a typical 2-channel 2-rank memory system).

1. Introduction

Applications access data from multiple data sources (i.e., storage, network, GPGPU, and other devices) in a system. In order to enable a convenient interface for applications to access data from multiple sources, modern systems *decouple* these different sources from applications by adding an abstraction layer, *main memory*, between them, thereby constructing a logical hierarchy: *user applications – main memory – IO devices*, as shown in Figure 1a. In this logical system hierarchy, most data migrations happen between adjacent layers. Specifically, when executing an application, the system first transfers the required data from the corresponding IO devices, such as storage or network, to the main memory (*IO access*). The processor then accesses the data from main memory (*CPU access*), and operates on it. Therefore, these two data transfers, namely CPU and IO accesses, are logically decoupled from each other through the use of main memory.

However, the physical implementation of a modern system is not in-line with this logical abstraction. This causes contention for system resources. As Figure 1b shows, the DRAM-based main memory is connected to the memory controller, which is managed by the processor. Therefore, both CPU and IO accesses contend for two major system resources. First, both types of accesses need to be generated and controlled by the processor cores. Therefore, when a core is busy performing a data transfer, it cannot perform other tasks. We call this *the control logic contention*. Second, the data transfers resulting from both types of accesses contend for *the memory channel*. Although the main memory is used to perform both CPU and IO accesses, there exists *only one physical connection to the main memory* (through the processor core and memory controller), leading to serialized memory accesses.

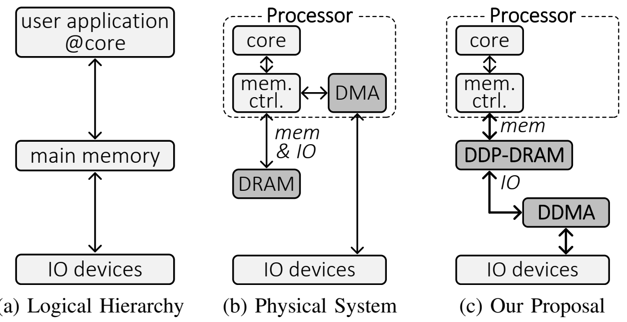


Figure 1: Methods to Connect Cores and Data Sources

One obvious solution to tackle this problem is to decouple communication between these layers, thereby enabling parallel accesses. Integrating Direct Memory Access (DMA) [3, 51] successfully decouples the control logic from the processor by directly generating IO requests to main memory from the DMA engine (Figure 1b). However, the main memory channel still suffers from contention between both CPU and IO accesses. Considering that modern systems support high IO bandwidth (e.g., PCIe 4.0 [72]: $32 GB/s$), which is similar to the bandwidth of a two-channel memory system (e.g., DDR3-2133 [52]: $17 GB/s \times 2\, channels = 34 GB/s$), IO accesses to memory can consume significant fraction of the memory bandwidth. Two recent computing trends, the move to multi-core systems and increasingly data-intensive applications, increase both CPU and IO accesses rapidly, which in turn makes memory channel contention an increasingly more serious performance bottleneck in modern systems [31, 55, 58].

Conventionally, processor and DRAM manufacturers have increased the memory channel bandwidth by either adding more channels or increasing the channel frequency. However, each additional memory channel requires a large number of processor pins (e.g., 144 pins per channel in Intel i7 [27]), significantly increasing the chip area cost. Increasing the channel frequency results in signal integrity problems, which require expensive solutions such as more precise I/O circuitry (high area cost) and more aggressive channel termination mechanisms (high power cost) [15, 23]. Therefore, increasing memory channel bandwidth at low cost remains a challenging problem in both today's and future systems [44].

Our goal, in this work, is to mitigate memory contention due to the shared channel to improve system performance. As we observe above, the difference between the logical system hierarchy and its physical implementation leads to contention at the intermediate layer, the main memory. To mitigate this memory channel contention, we propose a hardware-software cooperative data transfer mechanism, *Decoupled DMA* (*DDMA*), which provides mostly-independent data channels to the processor and IO devices, enabling concurrent accesses to memory from both layers.

As Figure 1c shows, DDMA consists of two hardware components: *(i)* a specialized DRAM that has two independent data ports, *Dual-Data-Port DRAM (DDP-DRAM)*, which is conceptually similar to but lower cost than dual-port DRAM [39], and *(ii)* off-the-processor-chip control logic (DDMA in the figure) that orchestrates the data transfers between the IO devices and the main memory. One data port of DDP-DRAM is connected to the processor (*CPU channel*) and the other data port is connected to the off-chip control logic (*IO channel*). The off-chip control logic, in turn, is connected to all the IO components, i.e., storage, network, and graphics units. The CPU channel serves only the processor cores' memory requests (CPU accesses), while the IO channel serves the IO accesses that transfer data between IO devices and main memory.

In a DDMA-based system, system software or the compiler determines which requests are served through the *IO channel*, and issues the corresponding data movement instructions. We give two examples. First, when migrating data from a page in storage to main memory, the operating system lets the hardware know that the data transfers need to be initiated on the IO channel of DDMA. Second, when migrating a set of data to/from external accelerators (e.g., GPGPU), the corresponding compilers of the external accelerators translate these data migration requests as IO channel data transfers on the DDMA.

The DDMA design not only provides an additional channel to efficiently handle such existing data transfers, but can also be used to enable new optimizations. For instance, the additional bandwidth provided by the IO channel can be used to perform IO operations more aggressively without interfering with the CPU channel (e.g., aggressive IO prefetching from storage, balancing requests across memory channels/banks, as we describe in Section 5).

We make the following contributions.

- We propose a new hardware-software cooperative data transfer mechanism, *DDMA*, which decouples IO accesses from CPU accesses. This prevents IO accesses from interfering with CPU accesses, conserving valuable memory bandwidth for CPU accesses while also increasing IO bandwidth, thereby improving overall system performance. (Sections 2 and 3)
- We demonstrate that DDMA is a new use case for the previously proposed dual-port DRAM design, which can reap significant performance benefits. We propose a reduced-cost Dual-Data-Port DRAM (DDP-DRAM) implementation by exploiting the characteristics of DDMA. (Section 3.2)
- We identify different scenarios that can take advantage of the DDMA substrate to achieve significant performance improvements such as: *(i)* CPU-GPU communication, *(ii)* in-memory communication (bulk data copy/initialization within the main memory) and *(iii)* memory-storage communication. (Section 5)
- We demonstrate that DDMA can be used across a wide variety of workloads and system configurations. Our evaluations, using CPU-GPU-communication-intensive and in-memory-communication-intensive applications, show that DDMA provides *(i)* significant system performance improvement, compared to a baseline system that employs a conventional memory system, and *(ii)* 83% of the performance of doubling the number of channels (Section 7). DDMA provides these benefits while at the same time *reducing* the pin count of the processor chip (see Sections 3.3 and 7.3 for more detail).
- Our DDMA design enables a new system paradigm, wherein the processor no longer needs to shoulder the burden of acting as an intermediary for *all* data movement. In modern systems, the processor die serves as the chief intermediary for data movement across various agents, which results in high logic complexity, large die area and high memory contention. Our DDMA design, on other hand, reduces processor die area and pin count, increasing system scalability. Doing so potentially enables more compute bandwidth or cache capacity on the processor, thereby enabling even higher system performance.

2. Motivation and Our Approach

Figure 2 shows the off-chip memory hierarchy in modern systems, consisting of the main memory system along with the IO devices. Communication between the processor and the off-chip main memory happens through the memory channel, which is managed by the *Memory Controller* [37, 78]. The memory controller, in turn, connects to the *DMA* (engine). The memory controller and the DMA engine manage communication between main memory and IO devices in a cooperative manner.

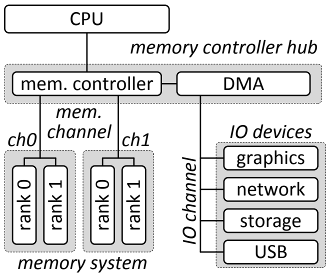


Figure 2: Modern System Organization with DMA

We examine three different kinds of accesses to main memory. First, when an application is executing on a processor core, if data accessed by the application is not present in the on-chip caches, the processor core accesses the main memory through the memory channel to retrieve this data (This is called a *CPU access*). Second, if the data is not present in main memory, the operating system sends out an IO request to the corresponding IO device to bring the required data into main memory (called an *IO access*), while pausing the application's execution at the processor core. This data is also written into main memory through the memory channel. Third, the operating system could migrate/copy data in bulk from one set of memory locations to another (called *in-memory communication* or *in-memory access*).

These three kinds of accesses happen through the memory channel, resulting in contention between different data accesses. Furthermore, IO and in-memory accesses are generally on the order of kilobytes [76, 85], occupying the memory channel for a larger number of cycles than CPU accesses, which typically move tens of bytes of data from main memory to the on-chip caches. Hence, sharing the memory channel between IO and in-memory accesses is a major source of contention, delaying CPU accesses.

Figure 3 shows the fraction of execution time spent on IO accesses, when executing applications from the Polybench [21, 22] suite that run on the CPU and the GPU, using the memory channel to communicate data between the CPU and the GPU. In a GPGPU context, typically, the CPU sends data to the GPU initially and sets up the GPU for execution. After execution, the GPU sends the results back to the CPU. As can be observed, for several applications, a large fraction (i.e., $\geq 40\%$) of the execution time is spent on transferring data between the CPU and GPU (through the main memory). Hence, these data transfers consume a significant fraction of the memory channel bandwidth, contending with CPU

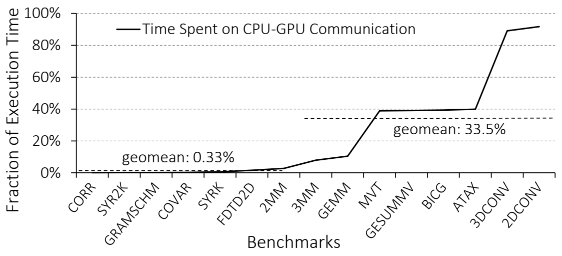


Figure 3: Time Consumed for CPU-GPU Communication

accesses from other applications, which leads to both CPU and system performance degradation, as also observed by previous work [85].[1]

Our goal is to address the problem of contention between CPU and IO/in-memory accesses in the shared memory channel, to improve overall system performance. To this end, we propose *Decoupled DMA (DDMA)*, which separates CPU and IO accesses by serving them on different channels. To enable DDMA, we first devise the Dual-Data-Port DRAM (DDP-DRAM), which can serve CPU and IO/in-memory accesses concurrently on each of its two data ports. Our DDP-DRAM design is a new and low-overhead version of the original dual-port DRAM [39]. Note that even though the concept of a dual-port DRAM is not new, leveraging it to isolate CPU and IO traffic to mitigate memory channel contention is. We describe our new DDP-DRAM design and how we leverage it to build a DDMA system in Section 3. We then describe the system support required to appropriately distribute accesses on the two channels of the DDMA system, in Section 4.

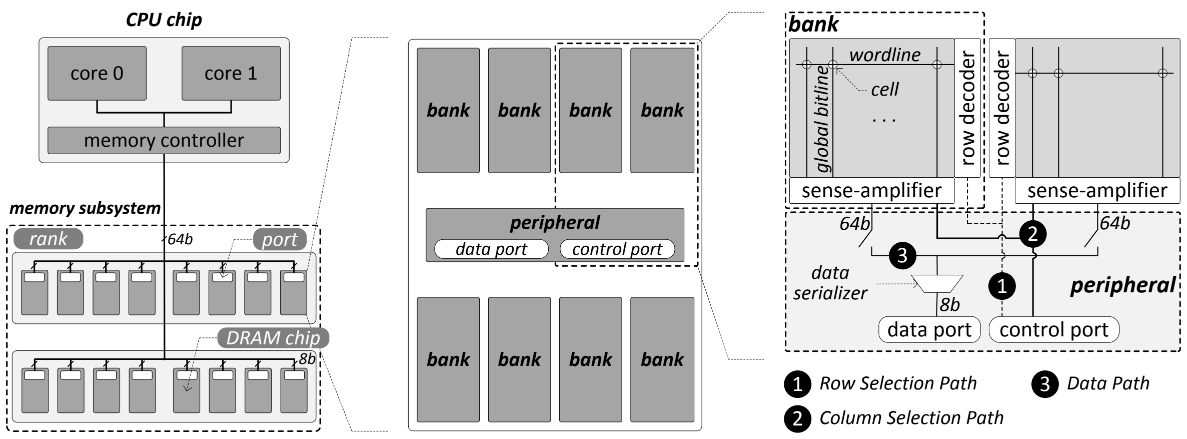
3. Hardware Implementation for DDMA

In this section, we first present a brief background on the organization of a DRAM-based memory system. We then present our Dual-Data-Port DRAM design. Finally, we describe how DDMA can be built based on DDP-DRAM and its control logic.

3.1. Organization of a Memory Subsystem

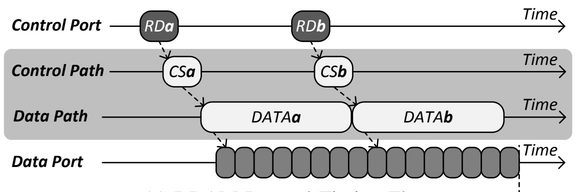
Figure 4a shows the organization of a typical DRAM-based memory system. A DRAM main memory system consists of *channels* that operate independently. A channel has one or more *ranks* that share the control and data buses of the channel. A rank typically consists of multiple DRAM chips (typically eight) and each chip, in turn, has multiple (typically eight) banks. All chips in a rank share the command bus, resulting in lockstep operation. Each bank has a 2-D array of DRAM cells that can be accessed in parallel with those of other banks, because each bank has its own separate peripheral logic (row decoder and sense-amplifiers in Figure 4c) to access its own cell array. Figure 4b shows the configuration of a DRAM chip that consists of eight banks and peripheral circuitry that connects the channel to the banks through the control and data ports.

[1]Tang et al. [85] also showed that a large fraction of memory accesses are IO accesses (File Copy: 39.9%, TPC-H: 20.0%).

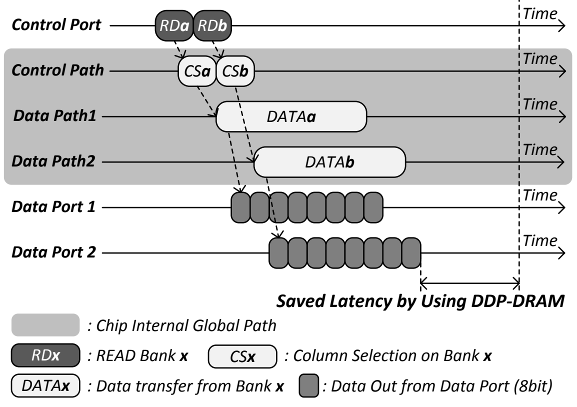


(a) Memory Subsystem (b) DRAM (c) DRAM Bank Operation

Figure 4: Hierarchical Organization of the DRAM-based Memory Subsystem



(a) DRAM Internal Timing Flow


Control Port
RDa
RDb
Time
Control Path
CSa
CSb
Time
Data Path1
DATAa
Time
Data Path2
DATAb
Time
Data Port 1
Time
Data Port 2
Time
Saved Latency by Using DDP-DRAM
: Chip Internal Global Path
RDx
: READ Bank x
CSx
: Column Selection on Bank x
DATAx
: Data transfer from Bank x
: Data Out from Data Port (8bit)

(b) DDP-DRAM Internal Timing Flow

Figure 5: Timing Comparison: DRAM vs. DDP-DRAM

Although banks can operate in parallel, there are shared interfaces inside DRAM, due to which the internal bank-level parallelism cannot be exploited fully. This *in-DRAM interface contention* happens because *all banks within a chip* share one global *control path* (row and column selection) and *data path*. Figure 4c shows how banks in a DRAM chip are organized and the peripheral logic around them. To access data present in a bank, the memory controller first issues a row selection command (referred to as `ACTIVATE`) that is transferred through the row selection path (❶ in Figure 4c). The row decoder that is local to the bank selects the appropriate row, following which the sense-amplifiers read the data from the selected row. The memory controller then issues a column selection command (commonly referred to as `READ`) that is transferred through the shared column selection path (❷ in Figure 4c), to select a fraction (typically 64 $bits$) of the entire row worth of sensed data (typically 16 $Kbits$ per DRAM logic). Then, the selected data in the row buffer is transferred to the data port through the shared data path (❸ in Figure 4c). Therefore, although each bank has its own row and column selection logic, which enables banks to operate in parallel, the control and data paths that carry commands and data, respectively, are shared across all the banks, thereby serializing accesses to banks.[2]

To illustrate the effect of contention in the shared internal DRAM paths, we show a timeline of commands and data on the in-DRAM interface (the grey region), as well as the external control port and data port (memory channel), in Figure 5a. The timeline shows two column selection commands (`READ` commands) destined to two different banks (*a* and *b*). To issue the first `READ` command for bank *a* (*RDa*), its decoded column selection signal (*CSa*) is transferred to its corresponding bank through the *control path*. The bank, then, starts responding with the corresponding data (*DATAa*) through the 64-bit *data path*. The data is divided into eight 8-bit chunks and transferred over the 8-bit *data port* in four clock cycles (due to Dual Data Rate – two transfers/clock).

When we extend this scenario by having a second `READ` command for bank *b* (*RDb*) right after *RDa*, we observe that the second request (*RDb*) suffers significant delay due to two shared resources. First, due to the shared control path, the memory controller can issue only one of the two requests during each cycle — *control path contention*. Second, due to the shared data path between banks, the second command (*RDb*) can be issued only after fully transferring data (*DATAa*) for the previous command (*RDa*), incurring a delay of four cycles — *data path contention*. Hence, both shared control and data paths lead to request serialization within DRAM. However, the effect of the data path contention (a four-cycle delay for the subsequent data request) is more significant than that of the control path contention (a one-cycle delay for the subsequent command). Therefore, we conclude that the shared data path/port is the main source of request serialization and hence, a critical performance bottleneck, as also observed by past work [31].

[2]We refer the reader to [44–47, 76] for the internals of DRAM operation.

3.2. Dual-Data-Port DRAM (DDP-DRAM)

We seek to address the shared data path bottleneck by introducing an additional data path and port in a DRAM chip. Figure 6 shows the additional data port (`dp1`) and data path (`data path 1`). The control port is not changed since the control port is likely not a bottleneck in data transfers where the channel is contended, as a command can be issued on the control port every cycle, as we described in Section 3.1. Each of the two data paths can be connected selectively to any of the banks and can be used to transfer data from any bank to the corresponding data port. The data paths are selectively connected to a bank using data switches. In order to manage these data switches, the memory controller needs to issue an additional signal, *data port selection*, along with a read/write command. For instance, if the memory controller sends a `READ` command to a bank with the data port selection signal set to `data port 1`, the corresponding bank is connected to `data port 1` by closing the bank's switch corresponding to `data path 1`. The requested data is then transferred from the bank to `data port 1` over `data path 1`.

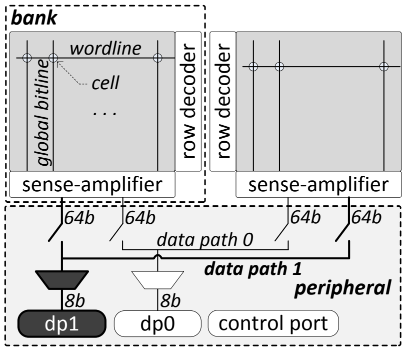


Figure 6: Dual-Data-Port DRAM Implementation

We refer to this specific dual-port DRAM design as *Dual-Data-Port DRAM (DDP-DRAM)*. By adding just an additional data port and path, DDP-DRAM achieves twice the data bandwidth of conventional DRAM. Previous work by Kim et al. has proposed two-channel mobile DRAM [39] which has two *fully independent channels* (each having its own control and data port). Our DDP-DRAM design, on the other hand, has *only one control port* (which does not affect command throughput significantly since the control port is not the bottleneck). Hence, DDP-DRAM has lower overhead than the previously proposed dual-port DRAM design.

Figure 5b shows a timeline demonstrating the potential benefits of using the DDP-DRAM design. The data accesses of the two `READ` commands (to two different banks) are performed mostly in parallel. Only a single cycle delay occurs due to command serialization in the shared control port. Hence, by introducing an additional data port while retaining a single control port, the data accesses to the two banks can be almost completely overlapped, leading to significant latency savings to serve the two `READ` requests compared to the baseline DRAM design.

3.3. Decoupled DMA (DDMA)

Figure 7 shows how our Dual-Data-Port DRAM is integrated as part of the off-chip memory hierarchy we propose. As described in Sections 1 and 2, IO accesses contend with CPU accesses in the shared memory channel in conventional systems. Therefore, we separate CPU accesses from IO accesses by using DDP-DRAM, with the goal of improving overall system performance. To this end, one data port (`dp0`) of the DDP-DRAM is used for CPU accesses, while the other data port (`dp1`) is used for IO accesses.

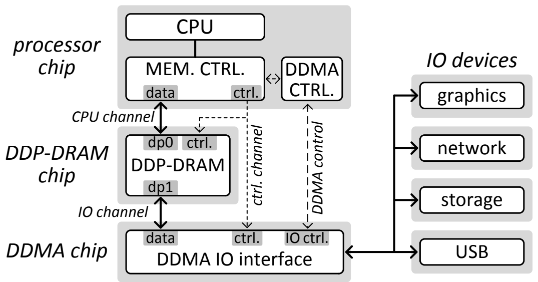


Figure 7: Proposed System Architecture

Figure 7 also shows the memory controller and DDMA and how they communicate with DDP-DRAM. DDMA consists of a *DDMA IO interface* chip (simply, *DDMA chip*) that connects DDP-DRAM to IO devices, and a *DDMA controller* (integrated into the processor chip) that manages the communication between the DDMA chip and IO devices. The *CPU channel* connects the memory controller (that is typically integrated into the processor chip) with `dp0` (`data port 0`) of the DDP-DRAM chip and carries data pertaining to CPU accesses that are initiated from the processor to main memory. The *IO channel* connects the DDMA chip with `dp1` (`data port 1`) of the DDP-DRAM chip and carries IO traffic. The memory controller controls *both* the CPU and IO ports of DDP-DRAM by issuing commands on the control channel, with the appropriate DDP-DRAM port selection. Note that while DDMA in Figure 7 is logically decoupled from the main memory and processors, there are many ways to physically implement the DDMA chip, including placing it as part of the DRAM chip or package (e.g., at the logic layer of the Hybrid Memory Cube [25] which contains a logic layer under DRAM layers in a 3D-stacked manner).

To control both the CPU and IO channels, the DDMA-based system has three main data communication modes: *(i)* CPU access, *(ii)* IO access and *(iii)* Port-bypass. CPU data accesses to DDP-DRAM happen through the CPU channel, while IO accesses happen through the IO channel. Although these are the two major kinds of data transfers, many modern systems also provide a direct communication interface between IO devices and processor caches: the CPU can send data directly to the IO interfaces/devices and vice versa. One example is Intel Direct Data I/O (DDIO) [24, 28] that supports direct communication between the network and the last level cache. To support such direct communication between the IO devices and the CPU, we add the *Port-Bypass mode* that transfers data using both the CPU and the IO channels. The *port selection signal*, explained in Section 3.2, is managed by the memory controller in the CPU to determine the communication mode. We next describe the READ operation of each communication mode, as shown in Figure 8. For WRITE, data moves in the opposite direction as for READ.

CPU Access Mode (Figure 8a). To bring data from DDP-DRAM, the memory controller issues READ commands with the port selection signal set to indicate the *CPU port*. Then, DDP-DRAM transfers data on the CPU channel and the memory controller senses the data from the CPU channel.

IO Access Mode (Figure 8b). To manage the data transfers between DDP-DRAM and IO devices, the memory controller issues a READ command with the port selection signal set to indicate the *IO port*. Note that both DDMA and DDP-DRAM observe the command when the port selection signal is set to indicate the *IO port*. DDP-DRAM transfers data on the IO channel for the READ request and DDMA senses the data from the IO channel.

Port-Bypass Mode (Figure 8c). When the memory controller issues a READ command with the port selection signal indicating both ports, DDP-DRAM internally connects its two data paths to directly connect both the IO and CPU channels. Then, DDMA transfers data to the memory controller through both CPU and IO channels (across DDP-DRAM). Note that during the port-bypass mode, no command is issued to DRAM banks and only peripheral logic is activated for transferring data between the two ports.

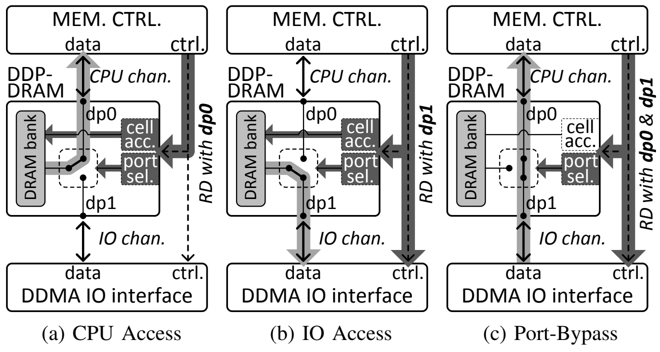


Figure 8: Three Data Communication Modes of DDMA (lighter arrow: data flow, darker arrow: control flow)

In the DDMA-based system, the DDMA chip includes IO interfaces (e.g., PCIe) to connect IO devices. To control the communication between the DDMA chip and IO devices, the processor requires an interface to issue commands to the DDMA chip. There are at least three ways to implement such a DDMA control interface: *(i)* adding a DDMA control channel with additional processor pins (shown as the connection between the DDMA controller and the DDMA chip in Figure 7), *(ii)* assigning a specific DDP-DRAM region as storage locations for DDMA control registers and transferring DDMA control commands through that region in DDP-DRAM, and *(iii)* using Port-Bypass mode to transfer commands directly to the DDMA engine. While the first approach requires additional processor pins, the other two approaches do not require any additional pins.

4. System Support for DDMA

So far, we described the hardware organization of the DDMA-based system. In this section, we describe the hardware and software interfaces to control the DDMA-based system.

4.1. Processor Support

We propose to both *(i)* leverage existing instructions to designate data transfers onto the IO channel and *(ii)* create new instructions to better leverage DDMA. First, we propose to map instructions for IO access to the IO channel. Intel IA-32 [26] provides variants of the IN and OUT instructions that transfer data to processor registers or the main memory from IO devices. In a DDMA-based system, data transfers corresponding to these instructions are simply mapped to the IO channel.

Second, we create new instructions for memory access through the IO channel. Load and Store instructions manage data transfers between the processor and main memory. Currently, these instructions are also used for bulk copy/initialization within main memory. As we will show in Section 5, such bulk in-memory communication can happen through the IO channel in a much more efficient manner. To this end, we augment the processor with two new instructions, LoadIO and StoreIO, which are variants of the memory access instructions, mapped to the IO channel.

Third, data transfers managed by the conventional DMA engine can be performed through the IO channel in the DDMA-based design. For instance, when CPU sets a range of memory to be read/written from/to IO devices in the DMA registers, data is transferred through the IO channel.

4.2. System Software Support

While processors provide hardware interfaces to manage DDMA, system software needs to use these interfaces effectively to exploit the features provided by DDMA, to improve system performance. To this end, system software first categorizes memory accesses into two groups: *(i)* CPU channel accesses and *(ii)* IO channel accesses. It then issues instructions corresponding to the appropriate group of memory accesses. Note that CPU channel accesses are performed similarly to a conventional memory system. However, IO channel accesses need to be managed carefully when they are moved to the IO channel, in order to ensure that the overall functionality is the same as in a conventional system. As Figure 9 shows, system software can easily categorize communications based on the source and destination addresses of data. Intuitively, when either the source or destination of a communication is the CPU, then the communication is identified as a CPU channel access, whereas if neither source nor destination of the communication is the CPU, the communication is categorized as an IO channel access. One exception is the *direct* communication between IO devices and the CPU, passing through both channels. To support such communication, DDMA provides a port-bypass mode as described in Section 3.3.

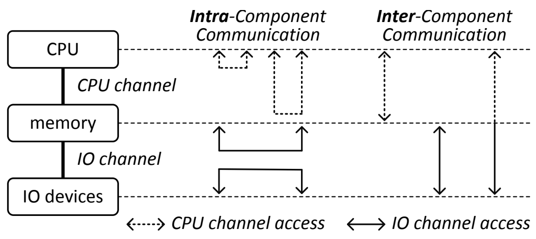


Figure 9: Types of Communication in DDMA

In a DDMA-based system, system software performs such identification and classification of requests at either compile-time or run-time while handling system calls requested by applications. For example, when an application invokes the *read* system call for a file, the requested data is transferred from a device to the buffer cache, and then from the buffer cache to the user address space, and is then finally moved to the CPU cache. The first and second transfers, which are the IO and in-memory accesses respectively, are performed on the IO channel, while the last one is performed on the CPU channel. When a task maps a file into its address space using the *mmap* system call, the page fault handling triggered by the first reference to a mapped page is served through the IO channel, while later references to the page are served through the CPU channel. Compilers can also support DDMA. For instance, GPGPU frameworks such as CUDA can provide specialized compilers that enable communication between main memory and the memory of the GPU, using DDMA.

5. DDMA Use Cases

DDMA can be used to implement an efficient memory system by decoupling CPU and IO accesses to improve system performance. In this section, we describe five scenarios where the DDMA-based system can be leveraged to achieve performance benefits.

5.1. Communication between Compute Units

Modern systems often consist of different kinds of processing units that are tailored to perform specific kinds of computation. In such systems, the processing units communicate with each other to share data. Such communication happens between the memory systems of the different processing units through IO accesses and can often be the bottleneck for system performance. We describe a GPGPU system as an example of such systems.

A GPGPU (General-Purpose Graphics Processing Unit) system [9, 59, 65] is a parallel computing platform that utilizes the data-parallel computation capability of multiple shader cores in a GPU (Graphics Processing Unit) to accelerate computation by dividing work across many parallel threads. In CUDA, one of the dominant GPGPU frameworks, the CPU (host) first sets up the GPGPU by transferring the data required to execute an application to the GPGPU (through the main memory). After finishing execution, the GPGPU transfers the results back to the CPU (through the main memory). These data transfers are carried out via IO accesses between the main memory and the GPU. As we show in Figure 3, the data communication between the CPU and the GPU is a significant fraction of execution time in a variety of GPGPU applications [21, 22] due to the resulting IO accesses. These IO accesses are mapped to the IO channel in our DDMA-based system, thereby preventing them from contending with CPU accesses.

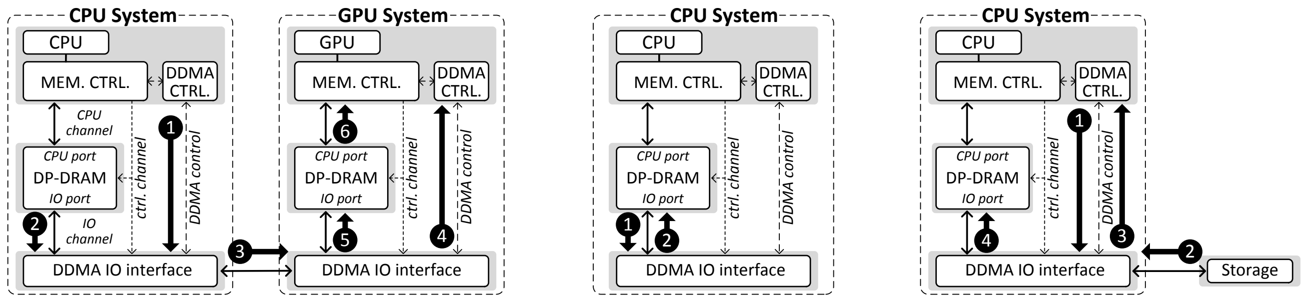
Figure 10a shows an example communication scenario involving a CPU and a GPU. When an application is executing on the CPU, it might need to offload some computation to the GPU. In order to enable this, the data on which the computation needs to be carried out has to be transferred to the GPU memory. In this scenario, the CPU memory controller issues commands for transferring data to the CPU system's DDMA chip through the IO channel (❶). The CPU DDMA controller then issues commands (❷) for communication between the DDMA chips of the CPU system and the GPU system, leading to data transfer to the GPU system's DDMA chip (❸). The GPU system's DDMA chip then lets the GPU know of the data being transferred (❹). Then, the GPU memory controller accesses the data through the IO channel (❺) and the CPU channel (❻).

In such a system that has multiple compute units, one critical challenge is how to efficiently provide a unified view of the different memory systems to a programmer [62, 66, 74]. One approach is to provide a single virtual address space across the memories of the different compute units and allow direct access to any part of the physical memory space from any compute unit (*Unified Virtual Addressing* [66]). Another approach is to keep the physical memory spaces of the different compute units separate (as in Figure 10a), while, however, providing a hardware mechanism underneath to automatically transfer data between the different memory systems (*Unified Memory* [62]). Only one copy of data is maintained in the unified virtual addressing approach, whereas data is duplicated and copied over across the different compute units in the unified memory approach. Our DDMA-based system can enable efficient data transfer mechanisms and thus, efficient implementations of such unified memory systems.

A major challenge in implementing a unified memory system is how to provide cache coherence across the physically separated systems. Designing such a cache coherence protocol is out of the scope of this work. However, our DDMA framework can support similar cache coherence protocols as existing memory systems. This is because, in our DDMA framework, all memory accesses (over both the CPU and IO channels) are managed by the processor. Hence, the processor can use this information for maintaining cache coherence.

5.2. In-Memory Communication and Initialization

Applications or system software often migrate/copy data in bulk from one set of memory locations to another, or initialize memory locations in bulk [76, 77]. We call these accesses that can happen completely within memory as *In-Memory Communication/Initialization*. There are many use cases for such in-memory accesses. We describe three of them. First, after a new process is forked, pages are copied



(a) CPU-GPU Communication (b) In-DRAM Communication (c) Memory-Storage Communication

Figure 10: DDMA-Based System Operations for Different DDMA Use Cases

from the parent process to the child process when they are updated, which is called Copy-on-Write [76, 77] (in-memory communication). Second, when multiple compute units on the same chip communicate through main memory (e.g., in an SoC or in a CPU-GPU hybrid architecture), pages are copied within main memory (in-memory communication). Third, when an application or system software allocates/deallocates a page in main memory, it initializes the memory region (in-memory initialization). Such in-memory accesses require no communication with other system components (e.g., CPU). However, modern systems require that they be performed through the memory channel. Therefore, the in-memory accesses contend with CPU accesses at the memory channel.

To mitigate memory channel contention, we propose to map such in-memory accesses to the IO channel of our DDMA-based system. First, we can use the DDMA chip (the DDMA IO interface chip) as an intermediary for in-memory communication. As shown in Figure 10b, the memory controller simply issues `LoadIO` (❶) to bring source data from memory to the DDMA chip. After that, it issues `StoreIO` (❷) to transfer the data to the target location in memory. Second, to support in-memory initialization, we add two new instructions, *init-zero* and *init-one*, that enable data initialization through the IO channel with predetermined data (i.e., zeroes or ones). When the memory controller issues an init-zero/init-one command with the IO port selected, DDMA holds the IO channel data port at zeroes or ones, thereby writing zero/one data to the specified addresses (❷).

For performing in-memory accesses, the only difference our proposal introduces (compared to conventional memory systems) is that DDMA uses the IO channel instead of the CPU channel. Therefore, the processor still knows which data is being updated and can achieve cache coherence by simply invalidating/updating the corresponding cache lines before starting DDMA-based in-memory accesses (This is similar to the cache coherence mechanisms in conventional systems [26, 76]).

5.3. Communication between Memory and Storage

Modern systems employ virtual memory to manage the memory system. A user application's code and data are maintained in storage and are brought into memory when the application requests them. The processor core executing the user application assumes that all pages are in main memory and sends data access requests to main memory (CPU access) when the requested data is not present in the on-chip caches. If the requested data is present in main memory, it is retrieved and the application continues execution. If the data is not present in main memory, the application's execution is paused and a *page fault* exception is generated to bring the requested data from storage to main memory. Figure 10c shows a communication scenario between memory and storage in the DDMA-based system. The operating system generates requests (❶) to the DDMA engine to bring the accessed page from storage to the DDMA chip (the DDMA IO interface chip). After finishing the data transfer from storage (❷), the DDMA chip sends an acknowledgment to the DDMA controller on the CPU chip (❸). Then, the memory controller issues commands for transferring data to main memory over the IO channel (❹). In today's systems, data for both CPU and IO accesses are transferred over the same memory channel. Since an entire page (several kilobytes or megabytes) is transferred from storage to memory on an IO access, IO accesses occupy the channel for a large number of cycles, contending with CPU accesses. DDMA mitigates the effect of this contention by isolating CPU and IO accesses to different ports.

The case for file read/write operations is similar to that of page fault handling. In order to access files in storage, the user application requesting the file issues system calls (*read/write*) with a pointer to the area reserved in memory for the file and the size of the requested file. The operating system then generates multiple IO requests to storage to bring the file from the IO device to the reserved memory area (file read), or to write the file from memory to the IO device (file write). To mitigate the contention caused by such file operations on the CPU's memory accesses, we map such file read/write operations to the IO channel of DDMA. Hence, the only difference between our framework and conventional systems is that data is transferred from the storage device *directly* to main memory without going through the on-chip memory controller as the intermediary. The on-chip memory controller still controls the corresponding accesses by issuing the appropriate commands (but it does not receive data). Therefore, simply invalidating the corresponding cache lines before starting DDMA provides cache coherence for communication between memory and storage (This is similar to the cache coherence mechanisms in conventional systems [26, 76]).

5.4. Aggressive IO Prefetching

To hide the long latency of IO devices, modern systems adopt IO prefetching techniques that speculatively bring data from IO devices to main memory, in advance of demand requests for the data from the CPU. In conventional memory systems, IO requests contend with memory requests at the memory channel, delaying demand requests to main memory. Previous works [6, 8, 10, 16, 53, 68, 70] proposed efficient IO prefetching techniques to mitigate memory channel contention. However, none of these techniques can fundamentally eliminate the memory channel contention caused by speculative requests.

Our DDMA design provides a fully-isolated IO channel, which can bring data from IO devices to main memory without interfering with CPU requests. Therefore, DDMA drastically reduces the cost of data transfer for IO prefetching. Hence, it can enable more aggressive IO prefetching for better performance. Note that DDMA is orthogonal to IO prefetching mechanisms, and can be used in conjunction with any of them.

5.5. Load Balancing in Memory Channels & Banks

The main memory system consists of multiple hierarchical components (channels/ranks/banks) that enable concurrent accesses. When requests contend at these components, the system suffers from high latency due to memory bandwidth contention, which can degrade system performance. Previous works mitigate such resource contention using two major approaches: *(i)* dynamic page migration [7, 12, 60], and *(ii)* application-aware data placement [5, 14, 54]. Both of these techniques require moving large amounts of data over the memory channel, which increases the memory channel contention. The IO channel of our DDMA design can migrate data completely within the memory system, without interfering with data accesses from the CPU cores, thereby enabling efficient or more aggressive implementation of data migration/placement techniques.

6. Evaluation Methodology

We evaluate our proposal with multi-programmed workloads on multi-core systems. We explore two of our case studies from Section 5, CPU-GPU communication and in-memory communication, using three different applications: *(i)* memory-intensive applications, *(ii)* CPU-GPU communication intensive-applications and *(iii)* in-memory communication-intensive applications. We describe our workload selection and simulation methods in the next subsections.

6.1. CPU–GPU Communication

GPGPU applications could run alongside other CPU-only applications. A GPGPU application's execution on the GPU is set up by a CPU core by first transferring data to the GPU. The GPU performs computation on the data and transfers the results back to the CPU (main memory). IO transfers due to the CPU-GPU communication could interfere with other applications' memory accesses.

Benchmarks. In order to construct workloads that would model this scenario, we use two categories of applications: *(i)* memory-intensive benchmarks, *(ii)* CPU-GPU communication-intensive benchmarks, as shown in Table I. For the memory-intensive benchmarks, we use Pinpoints [50, 69] to collect traces. We use 31 benchmarks from SPEC CPU2006 [80], stream [2] and TPC [86]. For CPU-GPU communication-intensive benchmarks, we collect memory access traces of Polybench [21, 22], executing on top of a CUDA [61] platform by using the Nvidia Visual Profiler [63]. We use a total of eight benchmarks for which data transfers between the CPU and the GPU constitute a significant fraction of execution time (data transfer time $> 10\%$ of total execution time).

Type	Benchmarks
Memory -Intensive	**cpu2006**: Total 23 benchmarks **tpc** [86]: *tpcc64, tpch17, tpch2, tpch6* **stream** [2]: *add, copy, scale, triad*
GPU-CPU Comm.	**polybench** [21, 22]: *2dconv, 3dconv, 3mm, atax, bicg gemm, gesummv, mvt*
In-Memory Comm.- Intensive	*apache* (web server), *bootup* (system bootup) *compiler* (gcc, make), *filecopy* (cp -R), *mysql* *fork* (system call), *shell* (a Linux shell script) *memcached* (in-memory key-value store)

Table I: Evaluated Benchmarks

Workloads. We construct workloads by combining memory intensive benchmarks and CPU-GPU communication intensive benchmarks. We construct three categories of 4-core, 8-core and 16-core workloads with different fractions of CPU-GPU communication-intensive and memory-intensive benchmarks - *(i)* 25% – 75%, *(ii)* 50% – 50%, and *(iii)* 75% – 25%. With 16 workloads in each category, we present results for 48 workloads for each multi-core system.

Simulation Methodology. We use a modified version of Ramulator [40], a fast cycle-accurate DRAM simulator that is available publicly [41]. We use Ramulator combined with a cycle-level x86 multi-core simulator. Table II shows the system configuration we model.

Component	Parameters
Processor	4–16 cores, 5.3 GHz, 3-wide issue, 8 MSHRs/core, 128-entry instruction window
Last-level cache	64B cache-line, 16-way associative, 512kB private cache-slice per core
Memory controller	64/64-entry read/write queues/controller, FR-FCFS scheduler
Memory system	DDR3-1066 (8–8–8) [33], 1–4 channels, 1–4 ranks-per-channel, 8 banks-per-rank
IO interface	PCI Express 3.0 [71]: 16 GB/s (8 GT/s)
GPGPU	Model: Nvidia C2050 [64], 448 CUDA cores, 3 $GByte$ GDDR5: 144 $GByte/s$

Table II: Configuration of Simulated Systems

We simulate the CPU-GPU communication-intensive benchmarks until completion. Since the CPU-GPU communication-intensive benchmarks execute for much longer than the co-running computation-intensive benchmarks, the computation-intensive benchmark traces wrap around upon reaching the end of the trace.

Furthermore, in order to reduce simulation times while realistically modeling contention between CPU and IO accesses, we scale down the length of the CPU-GPU communication and the GPU execution phases by 1/10 by randomly sampling and simulating communication and computation. We use the weighted speedup metric [19, 79] for performance evaluation.

6.2. In-Memory Communication

Benchmarks. We perform evaluations of this scenario using eight *in-memory communication-intensive benchmarks,* listed in Table I. We collect traces of these benchmarks using a full-system emulator (Bochs [1]) with small modifications. Since in-memory communication typically happens at a page granularity ($4\,KB$ in our evaluations), when there are 64 consecutive `Load` or `Store` operations, we translate those `Load` and `Store` operations into `LoadIO` and `StoreIO` operations that can be served through the IO channel in the DDMA-based system.

Workloads. We construct our workloads by combining computation-intensive and in-memory communication-intensive benchmarks. We build 48 four-, eight-, and sixteen-core workloads, resulting in a total of 144 workloads, as explained in Section 6.1.

Simulation Methodology. We use Ramulator [40] with a cycle-level x86 multi-core simulator (Section 6.1). We use a similar simulation methodology to that described in Section 6.1. We simulate each benchmark for 100 million representative instructions as done in many prior works [4, 11, 42–44, 54, 81–83].

7. Evaluation Results

We evaluate the benefits of employing DDMA in the context of systems where CPU-GPU communication and in-memory communication interfere with memory accesses from other CPU applications.

7.1. Performance Benefits

Figures 11a and 11b show the system performance improvement on a 2-channel 2-rank system for 4-, 8-, and 16-core workloads with CPU-GPU communication and in-memory communication, respectively. We draw two major conclusions. First, our DDMA-based system provides significant performance improvement over the baseline, across all core counts. Second, the performance improvements increase as the number of cores increases, due to the higher degree of contention between CPU and IO accesses, and, hence the larger opportunity for our DDMA-based system to provide performance improvement.

Sensitivity to Channel and Rank Count. Figures 12a and 13a show performance (weighted speedup) improvement for workloads with CPU-GPU communication and in-memory communication respectively across a variety of system configurations with varying number of cores, channels, and ranks.

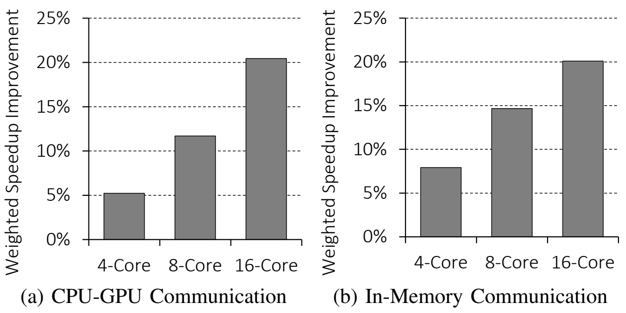
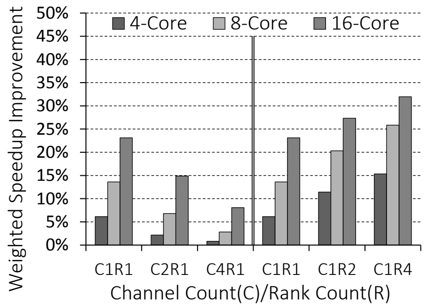


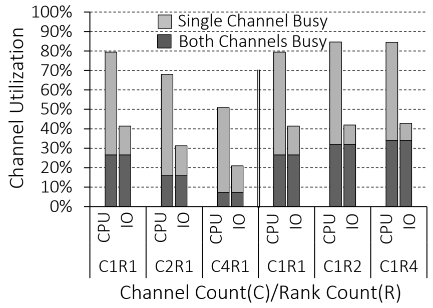
Figure 11: Performance on a 2-Channel 2-Rank System

Three conclusions are in order. First, our mechanism significantly improves performance across all system configurations (4–16 cores, 1–4 ranks, 1–4 banks). Second, as the number of channels increases (for a fixed rank count), the performance improvement decreases. This is due to reduced channel conflicts, which provide less opportunity for DDMA to reduce contention. Third, as the number of ranks increases (for a fixed channel count), the performance improvement is higher since the contention for memory bandwidth is higher.

Bandwidth Consumption of Both Channels. The source of the performance gains can be understood better by breaking down the bandwidth consumption on both CPU and IO channels, as shown in Figures 12b and 13b. The bottom dark grey portions represent the fraction of execution time when both the CPU and IO channels transfer data, while the light grey portions represent the fraction of execution time when only one of the data buses transfer data. In conventional memory systems, the dark grey portion would not exist, since requests would have to be served on only the CPU channel, whereas our DDMA-based system enables data transfers in parallel, as indicated by the dark grey fraction. As we would expect, on systems that have low channel counts and high rank counts, the fraction of data transferred in parallel (the dark grey fraction) is higher, since there is higher degree of contention.

Sensitivity to Workload Communication Intensity. Another observation we make from Figures 12b and 13b is that the IO channel utilization is relatively low ($20\% - 40\%$ for the CPU-GPU communication workloads and $10\% - 20\%$ for the in-memory communication workloads), while the CPU channel utilization ranges from 50% to 90%. These results reveal that the CPU channel is indeed overloaded. Therefore, when the CPU and IO accesses are transferred on the same channel, they interfere with each other, resulting in performance degradation. The results also show that there is still room for performance enhancement, e.g., by prefetching data from IO devices on the IO channel. We expect that our proposed memory system would provide more benefits when there are more accesses performed on the IO channel. To verify this hypothesis, we study the performance improvements for workloads with different communication intensities in Figure 12c and 13c. Our proposed mechanism provides more performance improvement for workloads that contain more CPU-GPU communication-intensive benchmarks and in-memory communication-intensive benchmarks


(a) DDMA Performance on Various Systems


(b) Channel Bandwidth (16-Core)

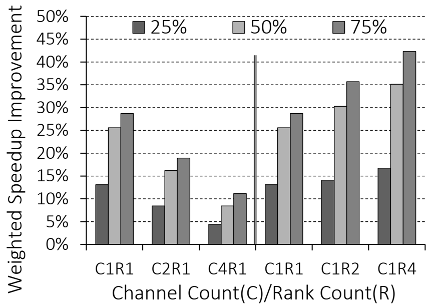
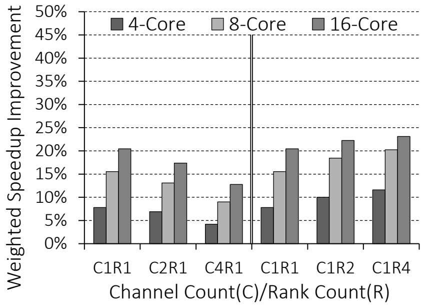
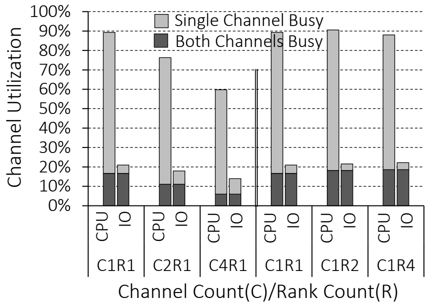

(c) DDMA Performance vs. Fraction of CPU-GPU Communication-Intensive Workloads (16-Core)

Figure 12: DDMA Performance for CPU-GPU Communication Workloads


(a) DDMA Performance on Various Systems


(b) Channel Bandwidth (16-Core)

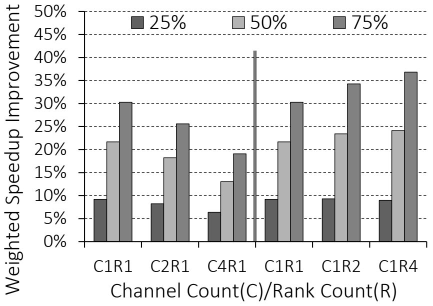
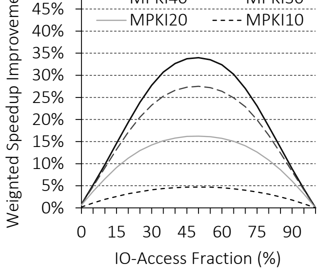

(c) DDMA Performance vs. Fraction of In-Memory Communication-Intensive Workloads (16-Core)

Figure 13: DDMA Performance for In-Memory Communication Workloads

(75%). This is because for workloads that have more CPU-GPU communication or in-memory communication benchmarks, a larger fraction of memory accesses can be served on the IO channel, reducing the memory bandwidth contention that exists in the baseline.

7.2. Sensitivity to Balance of Accesses on Channels

To evaluate the sensitivity of our DDMA-based system to the balance of accesses across the CPU and IO channels, we perform sensitivity studies with microbenchmarks that distribute accesses on the two channels. We generate the microbenchmarks by varying two factors: *(i)* memory intensity - Miss-Per-Kilo-Instructions (MPKI), *(ii)* the fraction of accesses on either channel, as shown in Figure 14. First, as expected, when accesses are distributed evenly on both channels, our DDMA mechanism provides the highest performance benefits. Second, DDMA provides higher performance improvement for workloads with higher memory intensities due to the higher contention caused by them.


(a) 8-Core System

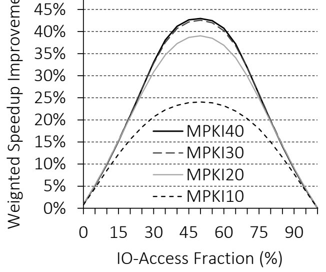
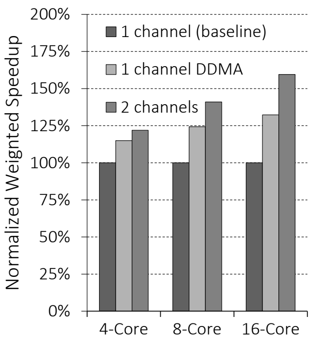

(b) 16-Core System

Figure 14: Channel Access Balance Analysis

7.3. Increasing Channel Count vs. Using DDMA

One important question that needs to be addressed to validate DDMA's effectiveness is: How does DDMA perform compared to using more memory channels? To investigate the effectiveness of our DDMA design, we evaluate the performance of three systems with the following memory system configurations in Figure 15a: *(i)* a conventional single-channel memory system, *(ii)* a *DDMA-based* single-channel memory system, and *(iii)* a conventional two-channel memory system. Compared to the conventional single-channel system, our DDMA design improves the performance of 4-, 8-, and 16-core workloads by 15%/24%/32%, respectively. We observe that doubling the number of channels improves performance by 22%/41%/60% over the baseline. Hence, our DDMA design achieves 94%/88%/82% of the performance benefits of doubling the channel count, for 4-, 8-, and 16-core workloads, respectively.

Figure 15b shows the processor pin count of the three


(a) Performance

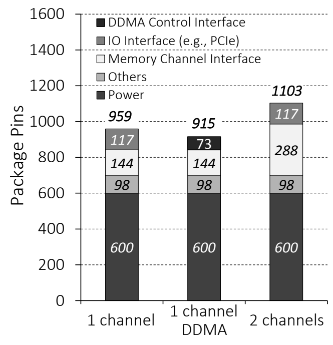

(b) Processor Pin Count [27, 29]

Figure 15: Increasing Channel Count vs. Using DDMA

systems. We make two observations. First, compared to the baseline single-channel memory system, using DDMA *reduces* the pin count by 4.5% (from 959 to 915). This is mainly due to moving the IO interface off the CPU chip. Second, doubling the number of channels *increases* pin count by 11.9% (from 959 to 1103). This is due to the new pins needed for the additional memory channel (114 pins). Hence, DDMA has the lowest pin count, yet its performance is much higher than the baseline and closer to the system with much higher pin count. We conclude that DDMA is a cost-effective approach to improving system performance.

7.4. Overheads in Implementing DDMA

Area Overhead of DDP-DRAM. To enable simultaneous data transfer, DDP-DRAM adds an additional data path and port, which consists of a 64-bit data bus per four banks (128 wires in total for an 8-bank DRAM chip), the corresponding logic and port selection switches, and data/power pads (eight data pads, eight power pads, two port selection pads, and two DQS pads). We estimate a 1.6% additional DRAM chip area overhead compared to the conventional single-port DRAM using $55\,nm$ $2\,GByte$ DDR3 models from Rambus [87]. A DDP-DRAM package would have 20 additional pins corresponding to the additional pads. Compared to the same baseline, previously proposed dual-port DRAM [39] has 6.7% additional area and 47 additional pads per chip (as it replicates control path and port as well as data). We conclude that our DDP-DRAM design has lower overhead than the state-of-the-art dual-port DRAM design.

Overall System Overhead vs. System Scalability. Our mechanism reduces CPU pin count mainly by removing IO interfaces from the CPU chip, as shown in Section 7.3. Even though our DDMA design leads to an additional chip (i.e., the DDMA chip, for the off-chip DDMA IO interface) and additional pins for DDP-DRAM, we believe that our DDMA-based system is cheaper and scalable than existing systems due to two reasons. First, CPU area is more expensive than off-chip area in terms of performance. In modern systems, the CPU is typically at the center of most processing activity. Therefore, the area nearby CPU cores is more expensive and critical for overall system performance. For example, 1 MByte of Last-Level Cache (on the CPU chip) is much more expensive and helpful for performance than 1 MByte of main memory (off the CPU chip). Second, our DDMA design enables good system scalability. Due to the growing bandwidth demand for IO interfaces, modern systems have integrated more IO interfaces onto the CPU chip at the expense of CPU area. We envision that these overheads will limit CPU scaling in the future. Our proposal provides a way to increase the bandwidth between the main memory (DRAM) and the IO devices while at the same time moving the IO interfaces off the CPU chip. Thus, our DDMA design enables both high bandwidth and scalability.

8. Related Work

To our knowledge, this is the first work to propose a main memory organization and a main memory control mechanism that decouples IO accesses from the processor and the memory channel (CPU channel) with the goal of reducing contention in the memory channel. In this section, we present related studies and compare them to our approach.

Separate Control for CPU and IO Accesses. Many past works propose techniques to manage IO accesses more efficiently. Tang et al. propose the DMA cache [85], a dedicated on-chip storage for IO, and sophisticated last-level cache partitioning mechanisms. Huggahalli et al. propose Direct Cache Access [24], a technique to directly inject IO data into on-chip caches. Iyer proposes dedicated data caching in the IO hub [32]. Many software mechanisms have been proposed to cache IO data in main memory to reduce IO access latency [38, 67, 75]. Although all these works deal with managing IO accesses, they do so using caching techniques. Our work, on the other hand, proposes a new memory system organization to effectively isolate CPU and IO accesses from each other by separating them onto different channels. Therefore, our approach is orthogonal to these previously proposed approaches and can be applied in conjunction with them, providing higher overall performance improvement.

Memories Having Multiple Independent Ports. Kim et al. propose OneDRAM [39], a DRAM architecture that has two completely independent data and control ports within a DRAM chip, developed to be shared by two separate processing units in a mobile system. Dual-Port SRAM (DP-SRAM) [13] is a buffer memory to connect two interfaces having different amounts of bandwidth. Some processors [3, 51] support DP-SRAM as part of DMA, however DP-SRAM only buffers data that would be migrated to the main memory through the memory channel. Video RAM (VRAM) [17, 73] was proposed for interfacing between the GPU and the display. All these memory architectures are conceptually similar to our DDP-DRAM in terms of employing multiple ports within a chip. However, DDP-DRAM is different in three major ways. First, all these previous architectures propose *multiple control and data ports* within an SRAM/DRAM chip. In contrast, our DDP-DRAM design adds *only an additional data port*, based on the observation that the data path and port (*not* the control port) are the main bottlenecks. As a result, the DRAM chip area overhead for DDP-DRAM (1.6%) is much less than other approaches (6.7% for OneDRAM; Section 7.4). Second, many of these architectures have been developed for very specific applications (DP-SRAM for embedded systems, VRAM for display), limiting their applicability. Third, these past approaches also add logic to support specialized functionalities (i.e., serial interface in VRAM), incurring higher area overhead and further limiting their applicability. Finally and most importantly, none of these previous works seek to isolate CPU and IO traffic as our DDMA proposal seeks to, by employing DDP-DRAM.

High Bandwidth DRAM Channel Organization. Decoupled DIMM [88] and Fully-Buffered DIMM [20] have been proposed to increase memory system bandwidth by adding a high speed channel interface. However, none of them tackle the specific problem of contention between CPU and IO accesses. Therefore, our approach is orthogonal to these previously proposed approaches.

3D-stacked memory systems [25, 49] have been proposed to increase memory bandwidth by exploiting a new chip-to-chip interface, through-silicon-vias (TSVs). While the TSV interface can increase the bandwidth between 3D-stacked chips, the propagation of heat from the processor layer to the 3D-stacked DRAM makes it difficult to directly integrate them with high performance processors, requiring off-chip connections, like a conventional memory channel. Therefore, adopting our proposal in the context of a 3D-stacked memory system can also provide performance improvements by making the memory channel less of a bottleneck.

Efficient In-Memory Communication. RowClone [76] accelerates in-memory communication (in particular, row-granularity bulk data copy/initialization) by utilizing the high internal DRAM bandwidth. However, it supports such data transfers only within a rank. Our mechanism, on the other hand, is more general and enables efficient data transfers across memories in different channels and across different compute units and memories.

Off-Loading I/O Management. I/O processor (IOP) [30] has been proposed to provide efficient interfaces between processors and IO devices. IOP is conceptually similar to our DDMA mechanism: it off-loads I/O management from the processor. However, IOP mainly focuses on the efficient management between different IO requests, while DDMA mitigates the interference between CPU and IO requests in main memory. Therefore, DDMA is orthogonal to IOP and can be applied in conjunction with it.

Efficient Memory Scheduling and Partitioning. Many previous works enable efficient memory channel utilization by employing more sophisticated memory scheduling [18, 31, 42, 43, 56, 57, 82–84], application-aware memory channel and bank partitioning [54], or vertical cache/DRAM bank partitioning [35, 48]. These works focus only on CPU traffic and do not handle IO traffic. On the other hand, our DDMA design provides a new data transfer mechanism for both CPU and IO traffic by employing the Dual-Data-Port DRAM architecture. Therefore, the DDMA framework is largely orthogonal to these scheduling and partitioning mechanisms, and can be applied in conjunction with them, providing even more efficient memory channel bandwidth utilization. Furthermore, the DDMA framework can enable the design of new hardware and software memory scheduling and partitioning mechanisms.

Balancing CPU and GPU Accesses. Several previous works [4, 34, 36] aim to reduce memory interference between CPU and GPU requests by better prioritization or throttling. Compared to these, our DDMA design not only reduces memory interference between CPU and GPU traffic, but more generally provides CPU and IO traffic isolation, thereby fundamentally eliminating the interference.

9. Conclusion

We introduce a new hardware-software cooperative data transfer mechanism, *Decoupled DMA*, that separates CPU and IO requests and provides independent data channels for them, with the goal of improving system performance. We observe that the main memory channel is a heavily contended resource, with CPU and IO accesses contending for the limited memory channel bandwidth. Our approach to mitigate this contention is to decouple IO accesses from CPU accesses. To this end, we propose a DDMA-based system organization using a Dual-Data-Port DRAM, with one data port connected to the CPU and the other port connected to the IO devices, enabling IO and CPU accesses to be performed on different ports concurrently.

We present different scenarios where our DDMA-based system can be leveraged to achieve high performance benefits, such as CPU-GPU communication, in-memory communication and storage-memory communication. We evaluate two scenarios across a variety of systems and workloads and show that our proposed mechanism significantly improves system performance. We also show that DDMA can potentially be leveraged by many speculative data transfer approaches, such as aggressive IO prefetching and speculative page migration over the IO channel, without interfering with user applications' requests that happen on the CPU channel. We conclude that Decoupled DMA is a substrate that can not only mitigate bandwidth contention at the main memory, but also provide opportunities for new system designs (for data-intensive computing). We hope our initial analyses will inspire other researchers to consider new use cases for DDMA and the Dual-Data-Port DRAM.

Acknowledgments

We thank the anonymous reviewers for their valuable suggestions. We thank the SAFARI group members for the feedback and the stimulating research environment they provide. We acknowledge the support of our industrial partners: Facebook, Google, IBM, Intel, Microsoft, Nvidia, Qualcomm, Samsung, and VMware. This research was partially supported by NSF (grants 0953246, 1212962, 1320531, 1409723), Semiconductor Research Corporation, and the Intel Science and Technology Center for Cloud Computing. Donghyuk Lee is supported in part by a Ph.D. scholarship from Samsung and the John and Claire Bertucci Graduate Fellowship.

REFERENCES

[1] "Bochs IA-32 emulator," http://bochs.sourceforge.net/.
[2] "STREAM Benchmark," http://www.streambench.org/.
[3] ARM, "ARM966E-S (Rev 1), Chapter 5," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0186a/DDI0186.pdf, 2000.
[4] R. Ausavarungnirun *et al.*, "Staged memory scheduling: achieving high performance and scalability in heterogeneous systems," in *ISCA*, 2012.
[5] M. Awasthi *et al.*, "Handling the Problems and Opportunities Posed by Multiple On-chip Memory Controllers," in *PACT*, 2010.
[6] S. Byna *et al.*, "Parallel I/O Prefetching Using MPI File Caching and I/O Signatures," in *SC*, 2008.
[7] R. Chandra *et al.*, "Scheduling and Page Migration for Multiprocessor Compute Servers," in *ASPLOS*, 1994.
[8] F. Chang and G. A. Gibson, "Automatic I/O Hint Generation Through Speculative Execution," in *OSDI*, 1999.

[9] S. Che *et al.*, "Accelerating Compute-Intensive Applications with GPUs and FPGAs," in *SASP*, 2008.
[10] Y. Chen *et al.*, "Hiding I/O Latency with Pre-Execution Prefetching for Parallel Applications," in *SC*, 2008.
[11] Y. Chou *et al.*, "Microarchitecture Optimizations for Exploiting Memory-Level Parallelism," in *ISCA*, 2004.
[12] J. Corbalan *et al.*, "Evaluation of the Memory Page Migration Influence in the System Performance: The Case of the SGI O2000," in *ICS*, 2003.
[13] Cypress, "Dual-Port SRAMs," http://www.cypress.com/?iD=82.
[14] R. Das *et al.*, "Application-to-core Mapping Policies to Reduce Memory System Interference in Multi-core Systems," in *HPCA*, 2013.
[15] H. David *et al.*, "Memory Power Management via Dynamic Voltage/Frequency Scaling," in *ICAC*, 2011.
[16] X. Ding *et al.*, "DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch," in *ATC*, 2007.
[17] T. J. Ebbers *et al.*, "Video RAM with External Select of Active Serial Access Register," in *US Patent 5,001,672*, 1991.
[18] E. Ebrahimi *et al.*, "Parallel Application Memory Scheduling," in *MICRO*, 2011.
[19] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," in *IEEE Micro*, 2008.
[20] B. Ganesh *et al.*, "Fully-buffered DIMM memory architectures: Understanding mechanisms, overheads and scaling," in *HPCA*, 2007.
[21] S. Grauer-Gray *et al.*, "PolyBench/GPU: Implementation of PolyBench codes for GPU processing," http://www.cse.ohio-state.edu/~pouchet/software/polybench/GPU/index.html.
[22] S. Grauer-Gray *et al.*, "Auto-tuning a high-level language targeted to GPU codes," in *InPar*, 2012.
[23] M. Greenberg, "LPDDR3 and LPDDR4: How Low-Power DRAM Can Be Used in High-Bandwidth Applications," http://www.jedec.org/sites/default/files/M_Greenberg_Mobile%20Forum_May_%202014_Final.pdf, 2013.
[24] R. Huggahalli *et al.*, "Direct Cache Access for High Bandwidth Network I/O," in *ISCA*, 2005.
[25] Hybrid Memory Cube Consortium, "Hybrid Memory Cube," http://www.hybridmemorycube.org/.
[26] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual."
[27] Intel, "Intel Core i7-800 and i5-700 Processor Series Datasheet," http://download.intel.com/design/processor/datashts/322164.pdf.
[28] Intel, "Intel Data Direct I/O Technology (Intel DDIO)," http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf.
[29] Intel, "Mobile Intel 4 Series Express Chipset Family," http://www.intel.com/Assets/PDF/datasheet/320122.pdf.
[30] Intel, "IOP333 I/O Processor with Intel XScale Microarchitecture," http://download.intel.com/design/iio/prodbref/30658301.pdf, 2011.
[31] E. Ipek *et al.*, "Self optimizing memory controllers: A reinforcement learning approach," in *ISCA*, 2008.
[32] R. Iyer, "Performance implications of chipset caches in web servers," in *ISPASS*, 2003.
[33] JEDEC, "DDR3 SDRAM, JESD79-3F," 2012.
[34] M. K. Jeong *et al.*, "A QoS-aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC," in *DAC*, 2012.
[35] M. K. Jeong *et al.*, "Balancing DRAM locality and parallelism in shared memory CMP systems," in *HPCA*, 2012.
[36] O. Kayiran *et al.*, "Managing GPU Concurrency in Heterogeneous Architectures," in *MICRO*, 2014.
[37] C. Keltcher *et al.*, "The AMD Opteron processor for multiprocessor servers," in *IEEE Micro*, 2003.
[38] H. Kim *et al.*, "Increasing Web Server Throughput with Network Interface Data Caching," in *ASPLOS*, 2002.
[39] J.-S. Kim *et al.*, "A 512 Mb Two-Channel Mobile DRAM (OneDRAM) with Shared Memory Array," in *JSSC*, 2008.
[40] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," in *IEEE CAL*, 2015.
[41] Y. Kim *et al.*, "Ramulator source code," https://github.com/CMU-SAFARI/ramulator.
[42] Y. Kim *et al.*, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
[43] Y. Kim *et al.*, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *MICRO*, 2010.
[44] Y. Kim *et al.*, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
[45] D. Lee *et al.*, "Tiered-latency DRAM: A low latency and low cost DRAM architecture," in *HPCA*, 2013.
[46] D. Lee *et al.*, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," in *HPCA*, 2015.
[47] J. Liu *et al.*, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.
[48] L. Liu *et al.*, "Going Vertical in Memory Management: Handling Multiplicity by Multi-policy," in *ISCA*, 2014.
[49] G. H. Loh, "3D-stacked memory architectures for multi-core processors," in *ISCA*, 2008.
[50] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.
[51] Microchip, "Direct Mermoy Access (DMA) (Part III)," http://ww1.microchip.com/downloads/en/DeviceDoc/70215C.pdf, 2008.
[52] Micron, "2Gb: x4, x8, x16, DDR3 SDRAM," 2012.
[53] T. C. Mowry *et al.*, "Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications," in *OSDI*, 1996.
[54] S. P. Muralidhara *et al.*, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in *MICRO*, 2011.
[55] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," in *IMW*, 2013.
[56] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.
[57] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.
[58] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," in *SUPERFRI*, 2015.
[59] J. Nickolls *et al.*, "Scalable parallel programming with CUDA," in *ACM Queue*, 2008.
[60] D. S. Nikolopoulos *et al.*, "A Case for User-level Dynamic Page Migration," in *ICS*, 2000.
[61] Nvidia, "CUDA parallel computing platform and programming model," https://developer.nvidia.com/category/zone/cuda-zone.
[62] Nvidia, "CUDA Toolkit Documentation v7.0," http://docs.nvidia.com/cuda/index.html.
[63] Nvidia, "Nvidia Visual Profiler," https://developer.nvidia.com/nvidia-visual-profiler.
[64] Nvidia, "Tesla C2050 C2070 GPU computing processor," http://www.nvidia.com/tesla.
[65] Nvidia, "Compute unified device architecture programming guide," 2007.
[66] Nvidia, "Peer-to-Peer & Unified Virtual Addressing," http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_GPUDirect_uva.pdf, 2011.
[67] V. S. Pai *et al.*, "IO-lite: A Unified I/O Buffering and Caching System," in *OSDI*, 1999.
[68] A. E. Papathanasiou and M. L. Scott, "Aggressive Prefetching: An Idea Whose Time Has Come," in *HoT-OS*, 2005.
[69] H. Patil *et al.*, "Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation," in *MICRO*, 2004.
[70] R. H. Patterson *et al.*, "Informed Prefetching and Caching," in *SOSP*, 1995.
[71] PCI-SIG, "PCI Express Base Specification Revision 3.0," https://www.pcisig.com/specifications/pciexpress/base3/, 2010.
[72] PCI-SIG, "PCI Express 4.0 evolution to 16GT/s, twice the throughput of PCI Express 3.0 technology," https://www.pcisig.com/news_room/Press_Releases/November_29_2011_Press_Release_/, 2011.
[73] R. Pinkham *et al.*, "A 128 K$\times$ 8 70-MHz multiport video RAM with auto register reload and 8$\times$ 4 block WRITE feature," in *JSSC*, 1988.
[74] J. Power *et al.*, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *MICRO*, 2013.
[75] S. Seelam *et al.*, "Application level I/O caching on Blue Gene/P systems," in *IPDPS*, 2009.
[76] V. Seshadri *et al.*, "RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
[77] V. Seshadri *et al.*, "Page Overlays: An Enhanced Virtual Memory Framework to Enable Fine-grained Memory Management," in *ISCA*, 2015.
[78] B. Sinharoy *et al.*, "IBM POWER7 multicore server processor," in *IBM Journal of R&D*, 2011.
[79] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *ASPLOS*, 2000.
[80] Standard Performance Evaluation Corporation, "SPEC CPU2006."
[81] J. Stuecheli *et al.*, "The virtual write queue: coordinating DRAM and last-level cache policies," in *ISCA*, 2010.
[82] L. Subramanian *et al.*, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *HPCA*, 2013.
[83] L. Subramanian *et al.*, "The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost," in *ICCD*, 2014.
[84] L. Subramanian *et al.*, "The Blacklisting Memory Scheduler: Balancing Performance, Fairness and Complexity," in *CoRR abs/1505.07502*, 2015.
[85] D. Tang *et al.*, "DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance," in *HPCA*, 2010.
[86] Transaction Processing Performance Council, "TPC Benchmark," http://www.tpc.org/.
[87] T. Vogelsang, "Understanding the Energy Consumption of Dynamic Random Access Memories," in *MICRO*, 2010.
[88] H. Zheng *et al.*, "Decoupled DIMM: building high-bandwidth memory system using low-speed DRAM devices," in *ISCA*, 2009.