RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization

Vivek Seshadri vseshadr@cs.cmu.edu
Yoongu Kim yoongukim@cmu.edu
Chris Fallin* cfallin@c1f.net
Donghyuk Lee donghyuk1@cmu.edu
Rachata Ausavarungnirun rachata@cmu.edu
Gennady Pekhimenko gpekhime@cs.cmu.edu
Yixin Luo yixinluo@andrew.cmu.edu
Onur Mutlu onur@cmu.edu
Phillip B. Gibbons† phillip.b.gibbons@intel.com
Michael A. Kozuch† michael.a.kozuch@intel.com
Todd C. Mowry tcm@cs.cmu.edu

Carnegie Mellon University †Intel Pittsburgh

ABSTRACT

Several system-level operations trigger bulk data copy or initialization. Even though these bulk data operations do not require any computation, current systems transfer a large quantity of data back and forth on the memory channel to perform such operations. As a result, bulk data operations consume high latency, bandwidth, and energy—degrading both system performance and energy efficiency.

In this work, we propose **RowClone**, a new and simple mechanism to perform bulk copy and initialization completely within DRAM — eliminating the need to transfer any data over the memory channel to perform such operations. Our key observation is that DRAM can internally and efficiently transfer a large quantity of data (multiple KBs) between a row of DRAM cells and the associated row buffer. Based on this, our primary mechanism can quickly copy an entire row of data from a source row to a destination row by first copying the data from the source row to the row buffer and then from the row buffer to the destination row, via two back-to-back *activate* commands. This mechanism, which we call the Fast Parallel Mode of RowClone, reduces the latency and energy consumption of a 4KB bulk copy operation by 11.6x and 74.4x, respectively, and a 4KB bulk zeroing operation by 6.0x and 41.5x, respectively. To efficiently copy data between rows that do not share a row buffer, we propose a second mode of RowClone, the Pipelined Serial Mode, which uses the shared internal bus of a DRAM chip to quickly copy data between two banks. RowClone requires only a 0.01% increase in DRAM chip area.

We quantitatively evaluate the benefits of RowClone by focusing on fork, one of the frequently invoked system calls, and five other copy and initialization intensive applications. Our results show that RowClone can significantly improve both single-core and multi-core system performance, while also significantly reducing main memory bandwidth and energy consumption.

*Currently at Intel Corporation.

MICRO -46, December 07-11 2013, Davis, CA, USA

Categories and Subject Descriptors

B.3.1 [**Memory Structures**]: Semiconductor Memories
D.4.2 [**Storage Management**]: Main Memory

Keywords

DRAM, Page Copy, Page Initialization, Memory Bandwidth, Performance, Energy, In-Memory Processing, Bulk Operations

1. INTRODUCTION

The main memory subsystem is an increasingly more significant limiter of system performance and energy efficiency for at least two reasons. First, the available memory bandwidth between the processor and main memory is not growing or expected to grow commensurately with the compute bandwidth available in modern multi-core processors [22, 25]. Second, a significant fraction (20% to 42%) of the energy required to access data from memory is consumed in driving the high-speed bus connecting the processor and memory [46] (calculated using [36]). Therefore, judicious use of the available memory bandwidth is critical to ensure both high system performance and energy efficiency.

In this work, we focus our attention on optimizing two important classes of bandwidth-intensive memory operations that frequently occur in modern systems: 1) *bulk data copy*—copying a large quantity of data from one location in physical memory to another, and 2) *bulk data initialization*—initializing a large quantity of data to a specific value. We refer to these two operations as *bulk data operations*. Prior research [41, 44] has shown that operating systems spend a significant portion of their time performing bulk data operations. Therefore, accelerating these operations will likely improve system performance. In fact, the x86 ISA has recently introduced instructions to provide enhanced performance for bulk copy and initialization (ERMSB [20]), highlighting the importance of bulk operations.

Bulk data operations degrade both system performance and energy efficiency due to three reasons. First, existing systems perform such operations one byte/word/cache line at a time. As a result, a bulk data operation incurs high latency, directly affecting the performance of the application performing the operation. For example, a typical system today (using DDR3-1066) takes roughly a microsecond (1046ns) to copy 4KB of data over the memory channel. Second, these operations require a large quantity of data to be transferred across the memory channel. Hence, they indirectly affect the performance of concurrently-running applica-

tions that share the memory bandwidth. Third, the data transfer across the memory channel represents a significant fraction of the energy required to perform a bulk data operation (40% for DDR3-1066). Clearly, all these problems result from the fact that existing systems must necessarily transfer a large quantity of data over the memory channel although neither copy nor initialization of bulk data requires any computation.

While bulk data operations can also affect performance by hogging the CPU and possibly polluting the on-chip caches, prior works [28, 58, 59] proposed simple solutions to address these problems. The proposed techniques include offloading bulk data operations to a separate engine to free up the CPU [28, 59] and/or providing hints to the processor caches to mitigate cache pollution [58]. However, none of these techniques eliminate the need to transfer data over the memory channel, and hence, all of them suffer from the three problems mentioned above. This is also true for the enhanced copy and move instructions (ERMSB) recently introduced in the x86 ISA [20].

Our goal is to design a mechanism that reduces the latency, bandwidth, and energy consumed by bulk data operations. Our approach is to provide low-cost hardware support for performing such operations completely within main memory.[1] This approach eliminates the need to transfer data over the memory channel to perform a bulk data operation, and hence can potentially mitigate the associated latency, bandwidth and energy problems.

We propose *RowClone*, a simple and low-cost mechanism to export bulk data copy and initialization operations to DRAM. RowClone exploits DRAM's internal architecture to efficiently perform these bulk data operations with low latency and energy consumption. At a high level, a modern DRAM chip consists of a hierarchy of banks and subarrays, as shown in Figure 1. Each chip is made up of multiple banks that share an internal bus for reading/writing data. Each bank in-turn contains multiple subarrays [31]. Each subarray consists of hundreds of rows of DRAM cells sharing a structure called a *row buffer*.

The key observation behind our approach is that DRAM internally transfers data from an entire row of DRAM cells to the corresponding row buffer (even to access a single byte from that row). Based on this observation, our primary mechanism efficiently copies all the data (multiple KBs) from one row of DRAM to another row within the same subarray in two steps: 1) copy the data from the source row to the row buffer, and 2) copy the data from the row buffer to the destination row. This mechanism, which we call the *Fast Parallel Mode* (FPM) of RowClone, requires very few modifications to DRAM, and these modifications are limited to the peripheral logic of the chip. FPM reduces the latency and energy consumption of a bulk copy operation by 11.6x and 74.4x (Section 7.1). For the system to accelerate a copy operation using FPM, the source and destination must be within the same subarray and the operation must span an entire row (Section 3.1). Despite these constraints, FPM can accelerate two widely used primitives in modern systems: 1) Copy-on-Write – a technique used by operating systems to postpone expensive copy operations [6, 8, 47, 50, 57], and 2) Bulk Zeroing – used for bulk memory initialization [7, 9, 45, 58] (Section 5.2 discusses several applications of Copy-on-Write and Bulk Zeroing in more detail).

To make RowClone more generally applicable to other copy operations between rows that do not share a row buffer, we propose a second mode of RowClone, called the *Pipelined Serial Mode* (PSM). PSM exploits the fact that the DRAM chip's internal bus

[1]In this work, we assume DRAM-based main memory, predominantly used by existing systems. Extending our mechanisms to other emerging memory technologies is part of our future work.

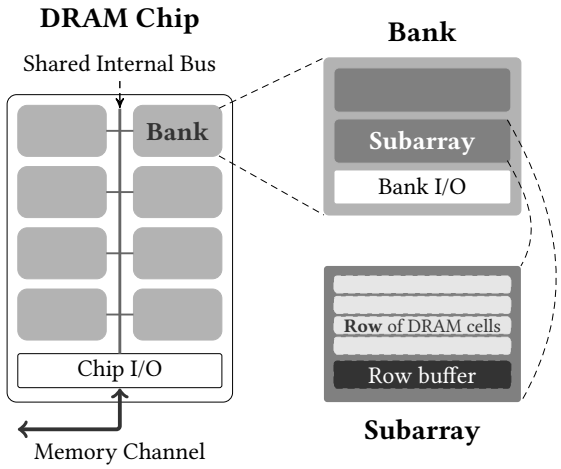


Figure 1: High-level organization of a DRAM chip

shared across all the banks (Figure 1) is used for both read and write operations. To copy data from one bank to another, PSM puts the source bank in read mode, the destination bank in write mode, and transfers the required cache lines over the internal bus from the source row to the destination row in a pipelined manner. By doing so, PSM overlaps the latency of the read and the write operations involved in the copy. We use PSM to efficiently copy data 1) from one bank to another and 2) from one subarray to another *within* the same bank (Section 3.3).[2] As we show in Section 7.1, PSM can reduce the latency and energy of an inter-bank copy operation by 1.9x and 3.2x, respectively.

In addition to the changes required to the DRAM chip (which cost only 0.01% additional chip area) to support bulk copy and initialization operations, RowClone requires changes to different layers of the system stack. In particular, we add two new instructions, `memcopy` and `meminit`, to the ISA to enable the software to specify occurrences of bulk copy and initialization operations to the hardware. We discuss the changes required to 1) the processor microarchitecture and the memory controller to implement the two new instructions, and 2) the system software to derive maximum benefits from RowClone. Section 4 describes these changes.

Applications from standard benchmark suites like SPEC CPU2006 [51] do not trigger many bulk copy or initialization operations. However, as mentioned before, several system-level functions invoke a significant fraction of copy and initialization operations. We discuss several such scenarios in which we can employ RowClone (Section 5). We quantitatively evaluate the benefits of RowClone using a `fork` benchmark (Section 7.2), and five other copy and initialization intensive applications: system bootup, compilation, memcached [2], mysql [3], and a shell script (Section 7.3). We also evaluate multi-programmed multi-core system workloads that consist of a mix of memory-intensive and copy/initialization-intensive applications. Our evaluations show that RowClone significantly improves both system performance and energy efficiency.

We make the following contributions:

- We show that the internal organization of DRAM presents an opportunity to perform bulk copy and initialization operations efficiently and quickly within DRAM.

[2]The DRAM hierarchy consists of higher levels, e.g., ranks and channels. However, as these transfers happen across DRAM chips, data copy across such levels must necessarily happen over the memory channel.

- We propose RowClone, a simple and low-cost mechanism to export bulk copy and initialization operations to DRAM via the memory controller. With only a 0.01% increase in DRAM chip area, RowClone can potentially reduce the latency/energy of a 4KB copy by 11.6x/74.4x.
- We analyze the benefits of RowClone by using it to accelerate two widely-used primitives in modern systems: Copy-on-Write and Bulk Zeroing. Our evaluations with a variety of workloads show that RowClone significantly improves the performance and energy efficiency of both single-core and multi-core systems and outperforms a memory-controller-based DMA approach [59].

2. DRAM BACKGROUND

In this section, we provide a brief background on DRAM organization and operation required to understand RowClone. As we will describe in Section 3, RowClone exploits the internal organization of modern DRAM chips as much as possible, and hence requires very few changes to existing DRAM chip designs.

As briefly mentioned in Section 1, a modern DRAM chip consists of multiple DRAM banks, which are further divided into multiple subarrays (Figure 1). Figure 2 shows the internal organization of a subarray. Each subarray consists of a 2-D array of DRAM *cells* connected to a single row of *sense amplifiers*, also referred to as the *row buffer*. A DRAM cell consists of two components: 1) a capacitor, which represents logical state in terms of charge, and 2) an access transistor that determines if the cell is currently being accessed. A sense amplifier is a component that is used to *sense* the state of the DRAM cell and *amplify* the state to a stable voltage level. The wire that connects the DRAM cell to the corresponding sense amplifier is called a *bitline* and the wire that controls the access transistor of a DRAM cell is called a *wordline*. Each subarray contains a row decoding logic that determines which row of cells (if any) is currently being accessed.

At a high level, accessing a cache line from a subarray involves three steps: 1) *activating* the row containing the cache line – this copies the data from the row of DRAM cells to the row buffer, 2) *reading/writing* the cache line – this transfers the data from/to the corresponding sense amplifiers within the row buffer to/from the processor through the internal bus shared by all the banks, and 3) *precharging* the subarray – this clears the sense amplifiers and prepares the subarray for a subsequent access.

Figure 3 pictorially shows the three steps of a subarray access. In the initial *precharged* state ❶, the sense amplifiers and the bitlines are maintained at an *equilibrium* voltage level of $\frac{1}{2}V_{DD}$ (half the maximum voltage, V_{DD}). The row decoder is fed with a sentinel input (—), such that all the wordlines within the subarray are lowered, i.e., no row is connected to the sense amplifiers.

To access data from a row A of a subarray, the DRAM controller first issues an `ACTIVATE` command with the row address. Upon receiving this command, the bank feeds the row decoder

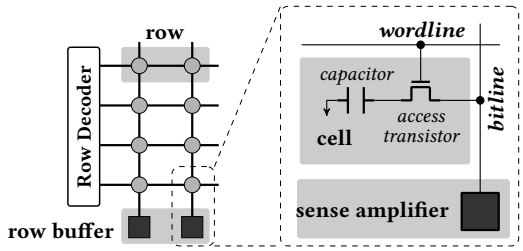


Figure 2: Internal organization of a DRAM subarray

of the corresponding subarray with the address A, which raises the wordline of the corresponding row. As a result, all cells of the row are connected to the corresponding bitlines and in turn to the sense amplifiers. Depending on the initial state of the cells, they lose/gain charge, thereby raising/lowering the voltage level on the corresponding bitlines (state ❷). In this state, the sense amplifiers detect the deviation from $\frac{1}{2}V_{DD}$ and begin to amplify the deviation (state ❸). In the process, they also drive the DRAM cells back to their original state. Once the amplification process is complete, all the sense amplifiers and bitlines are at a *stable* state (V_{DD} or 0) and the cells are fully restored to their initial state (state ❹).

The required data can be accessed from the sense amplifiers as soon as their voltage levels reach a threshold (state ❸). This is done by issuing a `READ` or `WRITE` command to the bank with the corresponding column address. Upon receiving the `READ` command, the bank I/O logic (Figure 1) accesses the required data from the corresponding sense amplifiers using a set of global bitlines that are shared across all subarrays [31] (not shown in the figure for clarity) and transfers the data to the shared internal bus outside the bank. From there, the data is transferred on to the memory channel connecting the DRAM and the memory controller. The `WRITE` command operates similarly except the data flows in the opposite direction. If the subsequent access is to the same row, then it can be performed by simply issuing a `READ`/`WRITE` command while the row is still activated.

When the DRAM controller wants to access data from a different row in the bank, it must first take the subarray back to the *precharged* state. This is done by issuing the `PRECHARGE` command to the bank. The precharge operation involves two steps. First, it feeds the row decoder with the sentinel input (—), lowering the wordline corresponding to the currently activated row. This essentially disconnects the cells from the corresponding bitlines. Second, it drives the sense amplifiers and the bitlines back to the *equilibrium* voltage level of $\frac{1}{2}V_{DD}$ (state ❻ – we intentionally ignore state ❺ for now as it is part of our mechanism to be described in Section 3.1).

In the following section, we describe the design and implementation of RowClone, which exploits the afore-described internal organization of DRAM to accelerate bulk data operations within DRAM while incurring low implementation cost.

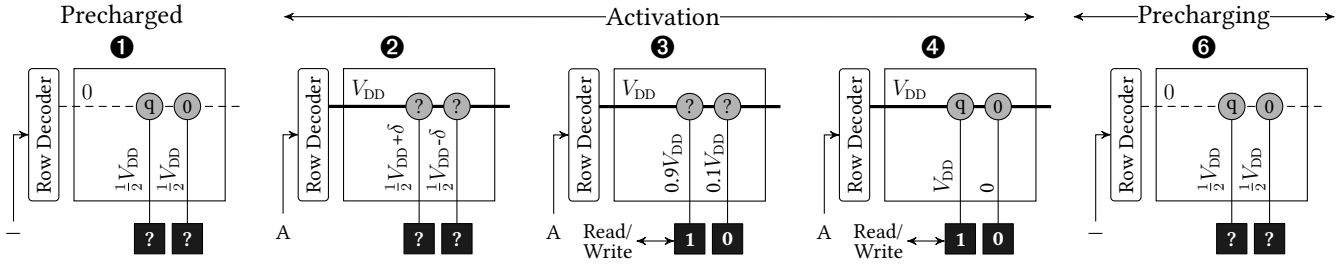


Figure 3: Steps involved in accessing a cache line from a subarray (figure adapted from [31])

3. ROWCLONE: DETAILED DESIGN

RowClone consists of two mechanisms based on two observations about DRAM operation. First, in order to access any data (even a single byte), DRAM internally transfers an entire row of data from the DRAM cells to the corresponding row buffer. Our first mechanism exploits this observation to efficiently copy data between two rows that share a row buffer. We call this the *Fast Parallel Mode* (FPM). Second, a single internal bus is shared across all banks within a chip to perform both read and write operations. Our second mechanism exploits this observation to copy cache lines between two banks in a pipelined manner. We call this the *Pipelined Serial Mode* (PSM). Although not as fast as FPM, PSM has fewer constraints and hence is more generally applicable. We now describe each of these modes in detail.

3.1 Fast Parallel Mode (FPM)

The Fast Parallel Mode is designed to copy data from a source row to a destination row within the same subarray. FPM exploits the fact that DRAM has the ability to transfer an entire row of data from the DRAM cells to the corresponding row buffer. FPM first copies the data from the source row to the row buffer and then copies the data from the row buffer to the destination row. The observation behind FPM's implementation is as follows:

> *If a DRAM cell is connected to a bitline that is at a stable state (either V_{DD} or 0) instead of the equilibrium state ($\frac{1}{2}V_{DD}$), then the data in the cell is overwritten with the data (voltage level) on the bitline.*

The reason for this behavior is that the cell induces only a small perturbation in the voltage level of the bitline. Therefore, when the bitline is at a stable state, any perturbation caused by the cell on the bitline is so small that the sense amplifier drives the bitline (and hence the cell) back to the original stable state of the bitline.

To copy data from a source row (`src`) to a destination row (`dst`) within the same subarray, FPM first activates the source row. At the end of the activation, the sense amplifiers and the bitlines are in a stable state corresponding to the data of the source row. The cells of the source row are fully restored to their original state. This is depicted by state ❹ in Figure 4. In this state, simply lowering the wordline of `src` and raising the wordline corresponding to `dst` connects the cells of the destination row with the stable bitlines. Based on the observation made above, doing so overwrites the data on the cells of the destination row with the data on the bitlines, as depicted by state ❺ in Figure 4.[3]

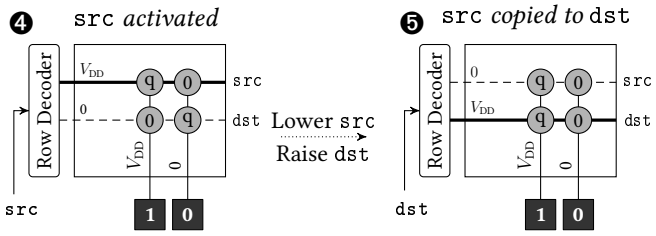


Figure 4: Fast Parallel Mode of RowClone

Implementation. Lowering the wordline corresponding to `src` and raising the wordline corresponding to `dst` can be achieved by simply changing the input to the subarray row decoder from `src` to `dst`. While issuing an `ACTIVATE` to the `dst` row will achieve this effect, modern DRAM chips do not allow another `ACTIVATE` to an already activated bank – the expected result of such an action is undefined. This is because a modern DRAM chip allows at most one row (subarray) within each bank to be activated. If a bank that already has a row (subarray) activated receives an `ACTIVATE` to a different subarray, the currently activated subarray must first be precharged [31].[4]

To support FPM, we propose the following change to the DRAM chip in the way it handles back-to-back `ACTIVATE`s. When an already activated bank receives an `ACTIVATE` to a row, the chip processes the command similar to any other `ACTIVATE` if and only if the command is to a row that belongs to the currently activated subarray. If the row does not belong to the currently activated subarray, then the chip takes the action it normally does with back-to-back `ACTIVATE`s—e.g., drop it. Since the logic to determine the subarray corresponding to a row address is already present in today's chips, implementing FPM only requires a comparison to check if the row address of an `ACTIVATE` belongs to the currently activated subarray, the cost of which is almost negligible.

Summary. To copy data from `src` to `dst` within the same subarray, FPM first issues an `ACTIVATE` to `src`. This copies the data from `src` to the subarray row buffer. FPM then issues an `ACTIVATE` to `dst`. This modifies the input to the subarray row-decoder from `src` to `dst` and connects the cells of `dst` row to the row buffer. This, in effect, copies the data from the sense amplifiers to the destination row. As we show in Section 7.1, with these two steps, FPM copies a 4KB page of data 11.6x faster and with 74.4x less energy than an existing system.

Limitations. FPM has two constraints that limit its general applicability. First, it requires the source and destination rows to be within the same subarray. Second, it cannot partially copy data from one row to another. Despite these limitations, we show that FPM can be immediately applied to today's systems to accelerate two commonly used primitives in modern systems – Copy-on-Write and Bulk Zeroing (Section 5). In the following section, we describe the second mode of RowClone – the Pipelined Serial Mode (PSM). Although not as fast or energy-efficient as FPM, PSM addresses these two limitations of FPM.

3.2 Pipelined Serial Mode (PSM)

The Pipelined Serial Mode efficiently copies data from a source row in one bank to a destination row in a *different* bank. PSM exploits the fact that a single internal bus that is shared across all the banks is used for both read and write operations. This enables the opportunity to copy an arbitrary quantity of data one cache line at a time from one bank to another in a pipelined manner.

More specifically, to copy data from a source row in one bank to a destination row in a different bank, PSM first activates the corresponding rows in both banks. It then puts the source bank in the read mode, the destination bank in the write mode, and transfers data one cache line (corresponding to a column of data—64 bytes) at a time. For this purpose, we propose a new DRAM command called `TRANSFER`. The `TRANSFER` command takes four parameters: 1) source bank index, 2) source column index, 3) destination bank index, and 4) destination column index. It copies the cache line corresponding to the source column index in the activated row of the source bank to the cache line corresponding to the destination column index in the activated row of the destination bank.

Unlike `READ`/`WRITE` which interact with the memory channel connecting the processor and main memory, `TRANSFER` does not transfer data outside the chip. Figure 5 pictorially compares the

[3] A recent prior work, TL-DRAM [34], uses this observation to move data between two segments within the same subarray.

[4] Some DRAM manufacturers design their chips to drop back-to-back `ACTIVATE`s to the same bank.

operation of the TRANSFER command with that of READ and WRITE. The dashed lines indicate the data flow corresponding to the three commands. As shown in the figure, in contrast to the READ or WRITE commands, TRANSFER does not transfer data from or to the memory channel.

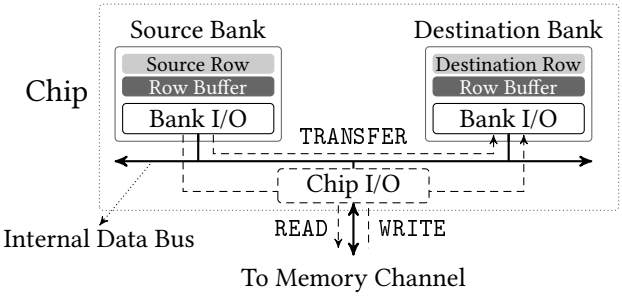


Figure 5: Pipelined Serial Mode of RowClone

Implementation. Implementing TRANSFER requires changes to 1) the DRAM chip and 2) the DRAM interface. First, the chip I/O logic should be augmented with logic to disconnect the internal DRAM bus from the memory channel during a TRANSFER and logic to transfer data appropriately between banks. Our calculations using a publicly available 2Gb 55nm DDR3 DRAM technology model [43] show that these changes (additional muxes) incur only a 0.01% die area cost. Second, the memory controller needs to provide four parameters along with TRANSFER. To minimize the changes to the existing DRAM interface, we make TRANSFER a two-cycle command. In the first cycle, the controller sends TRANSFER with the source bank ID and the source column index, and in the second cycle, it sends the destination bank ID and the destination column index.

Summary. To copy data from a source row in one bank to a destination row in a different bank, PSM first activates the corresponding rows in each bank. For each cache line to be copied between the two rows, PSM issues TRANSFER with the appropriate parameters. By overlapping the latency of the read and write operations involved in the copy, PSM results in a 1.9x reduction in latency and 3.2x reduction in energy of a 4KB bulk copy operation compared to performing the required operations over the memory channel.

3.3 Mechanism for Bulk Data Copy

When the data from a source row (src) needs to be copied to a destination row (dst), there are three possible cases depending on the location of src and dst: 1) src and dst are within the same subarray, 2) src and dst are in different banks, 3) src and dst are in different subarrays within the same bank. For case 1 and case 2, RowClone uses FPM and PSM, respectively, to complete the operation (as described in Sections 3.1 and 3.2).

For the third case, when src and dst are in different subarrays within the same bank, one can imagine a mechanism that uses the global bitlines (shared across all subarrays within a bank – described in [31]) to copy data across the two rows in different subarrays. However, we do not employ such a mechanism for two reasons. First, it is not possible in today's DRAM chips to activate multiple subarrays within the same bank simultaneously. Second, even if we enable simultaneous activation of multiple subarrays, as in [31], transferring data from one row buffer to another using the global bitlines requires the bank I/O circuitry to switch between read and write modes for each cache line transfer. This switching incurs significant latency overhead. To keep our design simple, for such an intra-bank copy operation, our mechanism uses PSM to first copy the data from src to a temporary row (tmp) in a different bank. It then uses PSM again to copy the data back from tmp to dst. The capacity lost due to reserving one row within each bank is negligible (0.0015% for a bank with 64k rows).

3.4 Mechanism for Bulk Data Initialization

Bulk data initialization sets a large block of memory to a specific value. To perform this operation efficiently, our mechanism first initializes a single DRAM row with the corresponding value. It then uses the appropriate copy mechanism (from Section 3.3) to copy the data to the other rows to be initialized.

Bulk Zeroing (or BuZ), a special case of bulk initialization, is a frequently occurring operation in today's systems [28, 58]. To accelerate BuZ, one can reserve one row in each subarray that is always initialized to zero. By doing so, our mechanism can use FPM to efficiently BuZ any row in DRAM by copying data from the reserved zero row of the corresponding subarray into the destination row. The capacity loss of reserving one row out of 512 rows in each subarray is very modest (0.2%).

Although the reserved zero rows can potentially lead to gaps in the physical address space, we can use an appropriate memory interleaving technique that maps consecutive rows to different subarrays. Such a technique ensures that the reserved zero rows are contiguously located in the physical address space. Note that interleaving techniques commonly used in today's systems (e.g., row or cache line interleaving [39]) have this property.

4. END-TO-END SYSTEM DESIGN

So far, we described RowClone, a DRAM substrate that can efficiently perform bulk data copy and initialization. In this section, we describe the changes to the ISA, the processor microarchitecture and the operating system that will enable the system to efficiently exploit the RowClone DRAM substrate.

4.1 ISA Support

To enable the software to communicate occurrences of bulk copy and initialization operations to the hardware, we introduce two new instructions to the ISA: memcopy and meminit. Table 1 describes the semantics of these two new instructions. We deliberately keep the semantics of the instructions simple in order to relieve the software from worrying about microarchitectural aspects of RowClone such as row size, alignment, etc. (discussed in Section 4.2.1). Note that such instructions are already present in some of the instructions sets in modern processors – e.g., rep movsd, rep stosb, ermsb in x86 [20] and mvcl in IBM S/390 [19].

Instruction	Operands	Semantics
memcopy	*src, dst, size*	Copy *size* bytes from *src* to *dst*
meminit	*dst, size, val*	Set *size* bytes to *val* at *dst*

Table 1: Semantics of the memcopy and meminit instructions

There are three points to note regarding the execution semantics of these operations. First, the processor does not guarantee atomicity for both memcopy and meminit, but note that existing systems also do not guarantee atomicity for such operations. Therefore, the software must take care of atomicity requirements using explicit synchronization. However, the microarchitectural implementation will ensure that any data in the on-chip caches is kept consistent during the execution of these operations (Section 4.2.2). Second, the processor will handle any page faults during the execution of these operations. Third, the processor can take interrupts during the execution of these operations.

4.2 Processor Microarchitecture Support

The microarchitectural implementation of the new instructions, `memcopy` and `meminit`, has two parts. The first part determines if a particular instance of `memcopy` or `meminit` can be fully/partially accelerated by RowClone. The second part involves the changes required to the cache coherence protocol to ensure coherence of data in the on-chip caches. We discuss these parts in this section.

4.2.1 Source/Destination Alignment and Size

For the processor to accelerate a copy/initialization operation using RowClone, the operation must satisfy certain alignment and size constraints. Specifically, for an operation to be accelerated by FPM, 1) the source and destination regions should be within the same subarray, 2) the source and destination regions should be row-aligned, and 3) the operation should span an entire row. On the other hand, for an operation to be accelerated by PSM, the source and destination regions should be cache line-aligned and the operation must span a full cache line.

Upon encountering a `memcopy`/`meminit` instruction, the processor divides the region to be copied/initialized into three portions: 1) row-aligned row-sized portions that can be accelerated using FPM, 2) cache line-aligned cache line-sized portions that can be accelerated using PSM,[5] and 3) the remaining portions that can be performed by the processor. For the first two regions, the processor sends appropriate requests to the memory controller which completes the operations and sends an acknowledgment back to the processor. The processor completes the operation for the third region similarly to how it is done in today's systems.[6]

4.2.2 Managing On-Chip Cache Coherence

RowClone allows the memory controller to directly read/modify data in memory without going through the on-chip caches. Therefore, to ensure cache coherence, the memory controller appropriately handles cache lines from the source and destination regions that may be cached in the on-chip caches before issuing the copy/initialization operations to memory.

First, the memory controller writes back any dirty cache line from the source region. This is because the main memory version of such a cache line is likely stale. Copying the data in-memory before flushing such cache lines will lead to stale data being copied to the destination region. Second, the memory controller invalidates any cache line (clean or dirty) from the destination region that is cached in the on-chip caches. This is because after performing the copy operation, the cached version of these blocks may contain stale data. The memory controller already has the ability to perform such flushes and invalidations to support Direct Memory Access (DMA) [21]. After performing the necessary flushes and invalidations, the memory controller performs the copy/initialization operation. To ensure that cache lines of the destination region are not cached again by the processor in the meantime, the memory controller blocks all requests (including prefetches) to the destination region until the copy or initialization operation is complete.

While simply performing the flushes and invalidates as mentioned above will ensure coherence, we propose a slightly modified solution to handle dirty cache lines of the source region to reduce memory bandwidth consumption. When the memory controller identifies a dirty cache line belonging to the source region while performing a copy, it creates an in-cache copy of the source cache line with the tag corresponding to the destination cache line. This has two benefits. First, it avoids the additional memory flush required for the dirty source cache line. Second and more importantly, the controller does not have to wait for all the dirty source cache lines to be flushed before it can perform the copy.[7]

Although RowClone requires the memory controller to manage cache coherence, it does not affect memory consistency — i.e., concurrent readers or writers to the source or destination regions involved in the copy/initialization. As mentioned before, a bulk copy/initialization operation is not guaranteed to be atomic even in current systems, and the software needs to explicitly perform the operation within a critical section to ensure atomicity.

4.3 Software Support

The minimum support required from the system software is the use of the proposed `memcopy` and `meminit` instructions to indicate bulk data operations to the processor. Although one can have a working system with just this support, maximum latency and energy benefits can be obtained if the hardware is able to accelerate most copy operations using FPM rather than PSM. Increasing the likelihood of the use of the FPM mode requires further support from the operating system (OS) on two aspects: 1) page mapping, and 2) granularity of copy/initialization.

4.3.1 Subarray-Aware Page Mapping

The use of FPM requires the source row and the destination row of a copy operation to be within the same subarray. Therefore, to maximize the use of FPM, the OS page mapping algorithm should be aware of subarrays so that it can allocate a destination page of a copy operation in the same subarray as the source page. More specifically, the OS should have knowledge of which pages map to the same subarray in DRAM. We propose that DRAM expose this information to software using the small EEPROM that already exists in today's DRAM modules. This EEPROM, called the Serial Presence Detect (SPD) [26], stores information about the DRAM chips that is read by the memory controller at system bootup. Exposing the subarray mapping information will require only a few additional bytes to communicate the bits of the physical address that map to the subarray index.[8]

Once the OS has the mapping information between physical pages and subarrays, it can maintain multiple pools of free pages, one pool for each subarray. When the OS allocates the destination page for a copy operation (e.g., for a *Copy-on-Write* operation), it chooses the destination page from the same pool (subarray) as the source page. Note that this approach does not require contiguous pages to be placed within the same subarray. As mentioned before, commonly used memory interleaving techniques spread out contiguous pages across as many banks/subarrays as possible to improve parallelism. Therefore, both the source and destination of a bulk copy operation can be spread out across many subarrays.

[5] Since `TRANSFER` copies only a single cache line, a bulk copy using PSM can be interleaved with other commands to memory.

[6] Note that the CPU can offload all these operations to the memory controller. In such a design, the CPU need not be made aware of the DRAM organization (e.g., row size, subarray mapping, etc.).

[7] In Section 7.3, we will consider another optimization, called RowClone-Zero-Insert, which inserts clean zero cache lines into the cache to further optimize Bulk Zeroing. This optimization does not require further changes to our proposed modifications to the cache coherence protocol.

[8] To increase DRAM yield, DRAM manufacturers design chips with spare rows that can be mapped to faulty rows [18]. Our mechanism can work with this technique by either requiring that each faulty row is remapped to a spare row within the same subarray, or exposing the location of all faulty rows to the memory controller so that it can use PSM to copy data across such rows.

4.3.2 Granularity of Copy/Initialization

The second aspect that affects the use of FPM is the granularity at which data is copied/initialized. FPM has a minimum granularity at which it can copy/initialize data. There are two factors that affect this minimum granularity: 1) the size of each DRAM row, and 2) the memory interleaving employed by the controller.

First, FPM copies *all* the data of the source row to the destination row (across the entire DIMM). Therefore, the minimum granularity of copy using FPM is at least the size of the row. Second, to extract maximum bandwidth, some memory interleaving techniques, like cache line interleaving, map consecutive cache lines to different memory channels in the system. Therefore, to copy/initialize a contiguous region of data with such interleaving strategies, FPM must perform the copy operation in each channel. The minimum amount of data copied by FPM in such a scenario is the product of the row size and the number of channels.

To maximize the likelihood of using FPM, the system or application software must ensure that the region of data copied (initialized) using the `memcopy` (`meminit`) instructions is at least as large as this minimum granularity. For this purpose, we propose to expose this minimum granularity to the software through a special register, which we call the *Minimum Copy Granularity Register* (MCGR). On system bootup, the memory controller initializes the MCGR based on the row size and the memory interleaving strategy, which can later be used by the OS for effectively exploiting RowClone. Note that some previously proposed techniques such as sub-wordline activation [54] or mini-rank [56, 60] can be combined with RowClone to reduce the minimum copy granularity, further increasing the opportunity to use FPM.

5. APPLICATIONS

RowClone can be used to accelerate any bulk copy and initialization operation to improve both system performance and energy efficiency. In this paper, we quantitatively evaluate the efficacy of RowClone by using it to accelerate two primitives widely used by modern system software: 1) Copy-on-Write and 2) Bulk Zeroing. We now describe these primitives followed by several applications that frequently trigger them.

5.1 Primitives Accelerated by RowClone

Copy-on-Write (CoW) is a technique used by most modern operating systems (OS) to postpone an expensive copy operation until it is actually needed. When data of one virtual page needs to be copied to another, instead of creating a copy, the OS points both virtual pages to the same physical page (source) and marks the page as read-only. In the future, when one of the sharers attempts to write to the page, the OS allocates a new physical page (destination) for the writer and copies the contents of the source page to the newly allocated page. Fortunately, prior to allocating the destination page, the OS already knows the location of the source physical page. Therefore, it can ensure that the destination is allocated in the same subarray as the source, thereby enabling the processor to use FPM to perform the copy.

Bulk Zeroing (BuZ) is an operation where a large block of memory is zeroed out. As mentioned in Section 3.4, our mechanism maintains a reserved row that is fully initialized to zero in each subarray. For each row in the destination region to be zeroed out, the processor uses FPM to copy the data from the reserved zero-row of the corresponding subarray to the destination row.

5.2 Applications that Use CoW/BuZ

We now describe seven example applications or use-cases that extensively use the CoW or BuZ operations. Note that these are just a small number of example scenarios that incur a large number of copy and initialization operations.

Process Forking. `fork` is a frequently-used system call in modern operating systems (OS). When a process (parent) calls `fork`, it creates a new process (child) with the exact same memory image and execution state as the parent. This semantics of `fork` makes it useful for different scenarios. Common uses of the `fork` system call are to 1) create new processes, and 2) create stateful threads from a single parent thread in multi-threaded programs. One main limitation of `fork` is that it results in a CoW operation whenever the child/parent updates a shared page. Hence, despite its wide usage, as a result of the large number of copy operations triggered by `fork`, it remains one of the most expensive system calls in terms of memory performance [47].

Initializing Large Data Structures. Initializing large data structures often triggers Bulk Zeroing. In fact, many managed languages (e.g., C#, Java, PHP) require zero initialization of variables to ensure memory safety [58]. In such cases, to reduce the overhead of zeroing, memory is zeroed-out in bulk.

Secure Deallocation. Most operating systems (e.g., Linux [7], Windows [45], Mac OS X [48]) zero out pages newly allocated to a process. This is done to prevent malicious processes from gaining access to the data that previously belonged to other processes or the kernel itself. Not doing so can potentially lead to security vulnerabilities, as shown by prior works [9, 13, 16, 17].

Process Checkpointing. Checkpointing is an operation during which a consistent version of a process state is backed-up, so that the process can be restored from that state in the future. This checkpoint-restore primitive is useful in many cases including high-performance computing servers [6], software debugging with reduced overhead [50], hardware-level fault and bug tolerance mechanisms [10, 11], and speculative OS optimizations to improve performance [8, 57]. However, to ensure that the checkpoint is consistent (i.e., the original process does not update data while the checkpointing is in progress), the pages of the process are marked with copy-on-write. As a result, checkpointing often results in a large number of CoW operations.

Virtual Machine Cloning/Deduplication. Virtual machine (VM) cloning [33] is a technique to significantly reduce the startup cost of VMs in a cloud computing server. Similarly, deduplication is a technique employed by modern hypervisors [55] to reduce the overall memory capacity requirements of VMs. With this technique, different VMs share physical pages that contain the same data. Similar to forking, both these operations likely result in a large number of CoW operations for pages shared across VMs.

Page Migration. Bank conflicts, i.e., concurrent requests to different rows within the same bank, typically result in reduced row buffer hit rate and hence degrade both system performance and energy efficiency. Prior work [53] proposed techniques to mitigate bank conflicts using page migration. The PSM mode of RowClone can be used in conjunction with such techniques to 1) significantly reduce the migration latency and 2) make the migrations more energy-efficient.

CPU-GPU Communication. In many current and future processors, the GPU is or is expected to be integrated on the same chip with the CPU. Even in such systems where the CPU and GPU share the same off-chip memory, the off-chip memory is partitioned between the two devices. As a consequence, whenever a CPU program wants to offload some computation to the GPU, it has to copy all the necessary data from the CPU address space to the GPU address space [23]. When the GPU computation is finished, all the data needs to be copied back to the CPU address space. This copying involves a significant overhead. By spreading

out the GPU address space over all subarrays and mapping the application data appropriately, RowClone can significantly speed up these copy operations. Note that communication between different processors and accelerators in a heterogeneous System-on-a-chip (SoC) is done similarly to the CPU-GPU communication and can also be accelerated by RowClone.

In the following sections, we quantitatively compare RowClone to existing systems and show that RowClone significantly improves both system performance and energy efficiency.

6. METHODOLOGY

Simulation. Our evaluations use an in-house cycle-level multi-core simulator along with a cycle-accurate command-level DDR3 DRAM simulator. The multi-core simulator models out-of-order cores, each with a private last-level cache.[9] We integrate RowClone into the simulator at the command-level. We use DDR3 DRAM timing constraints [27] to calculate the latency of different operations. Since `TRANSFER` operates similarly to `READ`/`WRITE`, we assume `TRANSFER` to have the same latency as `READ`/`WRITE`. For our energy evaluations, we use DRAM energy/power models from Rambus [43] and Micron [36]. Although, in DDR3 DRAM, a row corresponds to 8KB across a rank, we assume a minimum in-DRAM copy granularity (Section 4.3.2) of 4KB – same as the page size used by the operating system (Debian Linux) in our evaluations. For this purpose, we model a DRAM module with 512-byte rows per chip (4KB across a rank). Table 2 specifies the major parameters used for our simulations.

Component	Parameters
Processor	1–8 cores, OoO 128-entry window, 3-wide issue, 8 MSHRs/core
Last-level Cache	1MB per core, private, 64-byte cache line, 16-way associative
Memory Controller	One per channel, 64-entry read queue, 64-entry write queue
Memory System	DDR3-1066 (8-8-8) [27], 2 channels, 1 rank per channel, 8 banks per rank,

Table 2: Configuration of the simulated system

Workloads. We evaluate the benefits of RowClone using 1) a case study of the `fork` system call, an important operation used by modern operating systems, 2) six copy/initialization intensive benchmarks: *bootup*, *compile*, *forkbench*, *memcached* [2], *mysql* [3], and *shell* (Section 7.3 describes these benchmarks), and 3) a wide variety of multi-core workloads comprising the copy/initialization intensive applications running alongside memory-intensive applications from the SPEC CPU2006 benchmark suite [51]. Note that benchmarks such as SPEC CPU2006, which predominantly stress the CPU, typically use a small number of page copy and initialization operations and therefore would serve as poor individual evaluation benchmarks for RowClone.

We collected instruction traces for our workloads using Bochs [1], a full-system x86-64 functional emulator, running a GNU/Linux system. We modify the kernel's implementation of page copy/initialization to use the `memcopy` and `meminit` instructions and mark these instructions in our traces.[10] We collect 1-billion instruction traces of the representative portions of these workloads. We use the instruction throughput (IPC) metric to measure single-core performance. We evaluate multi-core runs using the weighted speedup metric, a widely-used measure of system throughput for multi-programmed workloads [14], as well as five other performance/fairness/bandwidth/energy metrics, as shown in Table 7.

[9]Since our mechanism primarily affects off-chip memory traffic, we expect our results and conclusions to be similar with shared caches as well.

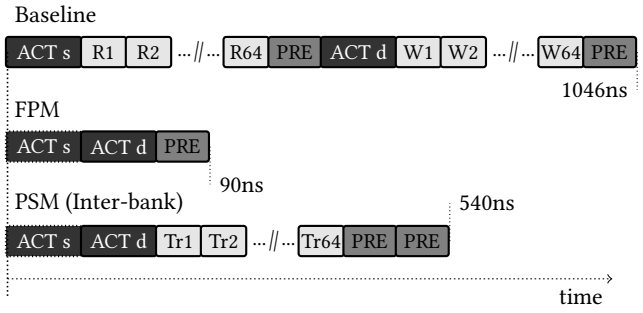
[10]For our `fork` benchmark (described in Section 7.2), we used the Wind River Simics full system simulator [4] to collect the traces.

7. EVALUATIONS

In this section, we quantitatively evaluate the benefits of RowClone. We first analyze the raw latency and energy improvement enabled by the DRAM substrate to accelerate a single 4KB copy and 4KB zeroing operation (Section 7.1). We then discuss the results of our evaluation of RowClone using `fork` (Section 7.2) and six copy/initialization intensive applications (Section 7.3). Section 7.4 presents our analysis of RowClone on multi-core systems and Section 7.5 provides quantitative comparisons to memory controller based DMA engines.

7.1 Latency and Energy Analysis

Figure 6 shows the sequence of DRAM commands issued by the baseline, FPM and PSM (inter-bank) to perform a 4KB copy operation. The figure also shows the overall latency incurred by each of these mechanisms, assuming DDR3-1066 timing constraints. Note that a 4KB copy involves copying 64 64B cache lines. For ease of analysis, only for this section, we assume that no cache line from the source or the destination region are cached in the on-chip caches. While the baseline serially reads each cache line individually from the source page and writes it back individually to the destination page, FPM parallelizes the copy operation of all the cache lines by using the large internal bandwidth available within a subarray. PSM, on the other hand, uses the new `TRANSFER` command to overlap the latency of the read and write operations involved in the page copy.



ACT s — `ACTIVATE` source, ACT d — `ACTIVATE` destination
R — `READ`, W — `WRITE`, Tr — `TRANSFER`, PRE — `PRECHARGE`

Figure 6: Command sequence and latency for Baseline, FPM, and Inter-bank PSM for a 4KB copy operation. Intra-bank PSM simply repeats the operations for Inter-bank PSM twice (source row to temporary row and temporary row to destination row). The figure is not drawn to scale.

Table 3 shows the reduction in latency and energy consumption due to our mechanisms for different cases of 4KB copy and zeroing operations. To be fair to the baseline, the results include only the energy consumed by the DRAM and the DRAM channel. We draw two conclusions from our results.

First, FPM significantly improves both the latency and the energy consumed by bulk data operations — 11.6x and 6x reduction in latency of 4KB copy and zeroing, and 74.4x and 41.5x reduction

	Mechanism	Absolute Latency (ns)	Absolute Memory Energy (μJ)	Reduction Latency	Reduction Memory Energy
Copy	Baseline	1046	3.6	1.00x	1.0x
	FPM	90	0.04	**11.62x**	**74.4x**
	Inter-Bank - PSM	540	1.1	1.93x	3.2x
	Intra-Bank - PSM	1050	2.5	0.99x	1.5x
Zero	Baseline	546	2.0	1.00x	1.0x
	FPM	90	0.05	**6.06x**	**41.5x**

Table 3: DRAM latency and memory energy reductions due to RowClone

in memory energy of 4KB copy and zeroing. Second, although PSM does not provide as much benefit as FPM, it still reduces the latency and energy of a 4KB inter-bank copy by 1.9x and 3.2x, while providing a more generally applicable mechanism.

When an on-chip cache is employed, any line cached from the source or destination page can be served at a lower latency than accessing main memory. As a result, in such systems, the baseline will incur a lower latency to perform a bulk copy or initialization compared to a system without on-chip caches. However, as we show in the following sections (7.2–7.4), *even in the presence of on-chip caching*, the raw latency/energy improvement due to RowClone translates to significant improvements in both overall system performance and energy efficiency.

7.2 The `fork` System Call

As mentioned in Section 5.2, `fork` is one of the most expensive yet frequently-used system calls in modern systems [47]. Since `fork` triggers a large number of CoW operations (as a result of updates to shared pages from the parent or child process), RowClone can significantly improve the performance of `fork`.

To analyze this, we use a simple benchmark, `forkbench`. The performance of `fork` depends on two parameters: 1) the size of the address space used by the parent—which determines how much data may potentially have to be copied, and 2) the number of pages updated after the `fork` operation by either the parent or the child—which determines how much data are actually copied. To exercise these two parameters, `forkbench` first creates an array of size $\mathcal{S}$ and initializes the array with random values. It then forks itself. The child process updates N random pages (by updating a cache line within each page) and exits; the parent process waits for the child process to complete before exiting itself.

As such, we expect the number of copy operations to depend on N—the number of pages copied. Therefore, one may expect RowClone's performance benefits to be proportional to N. However, an application's performance typically depends on the *overall memory access rate* [52], and RowClone can only improve performance by reducing the *memory access rate due to copy operations*. As a result, we expect the performance improvement due to RowClone to primarily depend on the *fraction* of memory traffic (total bytes transferred over the memory channel) generated by copy operations. We refer to this fraction as FMTC—Fraction of Memory Traffic due to Copies.

Figure 7 plots FMTC of `forkbench` for different values of $\mathcal{S}$ (64MB and 128MB) and N (2 to 16k) in the baseline system. As the figure shows, for both values of $\mathcal{S}$, FMTC increases with increasing N. This is expected as a higher N (more pages updated by the child) leads to more CoW operations. However, because of the presence of other read/write operations (e.g., during the initialization phase of the parent), for a given value of N, FMTC is larger for $\mathcal{S}$ = 64MB compared to $\mathcal{S}$ = 128MB. Depending on the value of $\mathcal{S}$ and N, anywhere between 14% to 66% of the memory traffic arises from copy operations. This shows that accelerating copy operations using RowClone has the potential to significantly improve the performance of the `fork` operation.

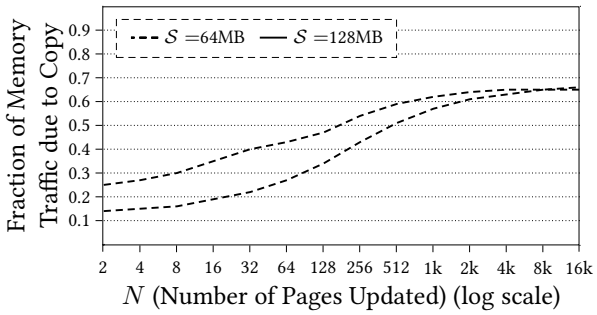


Figure 7: FMTC of `forkbench` for varying $\mathcal{S}$ and N

Figure 8 plots the performance (IPC) of FPM and PSM for `forkbench`, normalized to that of the baseline system. We draw two conclusions from the figure. First, FPM significantly improves the performance of `forkbench` for both values of $\mathcal{S}$ and most values of N. The peak performance improvement is 2.2x for N = 16k (30% on average across all data points). As expected, the performance improvement of FPM increases as the number of pages updated increases. The trend in performance improvement of FPM is similar to that of FMTC (Figure 7), confirming our hypothesis that FPM's performance improvement primarily depends on FMTC. Second, PSM does not provide considerable performance improvement over the baseline. This is because the large on-chip cache in the baseline system buffers the writebacks generated by the copy operations. These writebacks are flushed to memory at a later point without further delaying the copy operation. As a result, PSM, which just overlaps the read and write operations involved in the copy, does not improve latency significantly in the presence of a large on-chip cache. On the other hand, FPM, by copying all cache lines from the source row to destination in parallel, significantly reduces the latency compared to the baseline (which still needs to read the source blocks from main memory), resulting in high performance improvement.

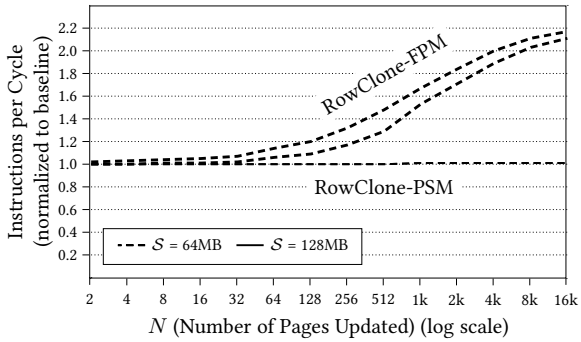


Figure 8: Performance improvement due to RowClone for `forkbench` with different values of $\mathcal{S}$ and N

Figure 9 shows the reduction in DRAM energy consumption (considering both the DRAM and the memory channel) of FPM and PSM modes of RowClone compared to that of the baseline for `forkbench` with $\mathcal{S}$ = 64MB. Similar to performance, the overall DRAM energy consumption also depends on the total memory access rate. As a result, RowClone's potential to reduce DRAM

energy depends on the fraction of memory traffic generated by copy operations. In fact, our results also show that the DRAM energy reduction due to FPM and PSM correlate well with FMTC (Figure 7). By efficiently performing the copy operations, FPM reduces DRAM energy consumption by up to 80% (average 50%, across all data points). Similar to FPM, the energy reduction of PSM also increases with increasing N with a maximum reduction of 9% for N=16k.

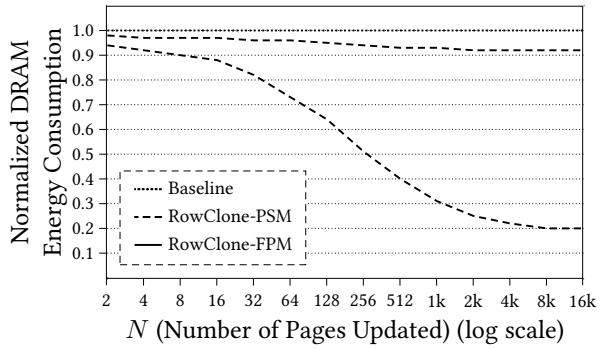


Figure 9: Comparison of DRAM energy consumption of different mechanisms for `forkbench` ($\mathcal{S}$ = 64MB)

In a system that is agnostic to RowClone (i.e., one that does not consider the relative performance benefits of FPM and PSM while allocating pages), we expect the performance improvement and energy reduction of RowClone to be in between that of FPM and PSM. By making the system software aware of RowClone (Section 4.3), we can approximate the maximum performance and energy benefits by increasing the likelihood of the use of FPM.

7.3 Copy/Initialization Intensive Applications

In this section, we analyze the benefits of RowClone on six copy/initialization intensive applications, including one instance of the `forkbench` described in the previous section. Table 4 describes these applications.

Name	Description
bootup	A phase booting up the Debian operating system.
compile	The compilation phase from the GNU C compiler (while running *cc1*).
forkbench	An instance of the `forkbench` described in Section 7.2 with $\mathcal{S}$ = 64MB and N = 1k.
mcached	Memcached [2], a memory object caching system, a phase inserting many key-value pairs into the memcache.
mysql	MySQL [3], an on-disk database system, a phase loading the sample *employeedb* database.
shell	A Unix shell script running 'find' on a directory tree with 'ls' on each sub-directory (involves filesystem accesses and spawning new processes).

Table 4: Copy/Initialization-intensive benchmarks

Figure 10 plots the fraction of memory traffic due to copy, initialization, and regular read/write operations for the six applications. For these applications, between 10% and 80% of the memory traffic is generated by copy and initialization operations.

Figure 11 compares the IPC of the baseline with that of RowClone and a variant of RowClone, RowClone-ZI (described shortly). As can be seen from the figure, the RowClone-based initialization mechanism slightly degrades performance for the

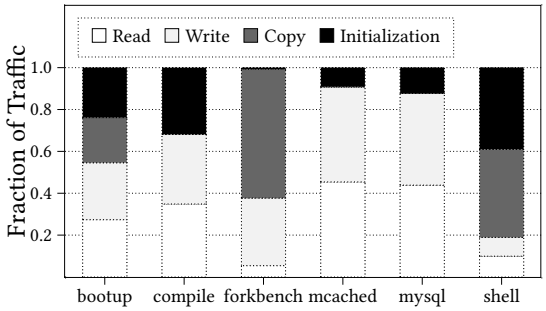


Figure 10: Fraction of memory traffic due to read, write, copy and initialization operations

applications which have a negligible number of copy operations (*mcached*, *compile*, and *mysql*).

Further analysis indicated that, for these applications, although the operating system zeroes out any newly allocated page, the application typically accesses almost all cache lines of a page immediately after the page is zeroed out. There are two phases: 1) the phase when the OS zeroes out the page, and 2) the phase when the application accesses the cache lines of the page. While the baseline incurs cache misses during phase 1, RowClone, as a result of performing the zeroing operation completely in memory, incurs cache misses in phase 2. However, the baseline zeroing operation is heavily optimized for memory-level parallelism (MLP) [40]. In contrast, the cache misses in phase 2 have low MLP. As a result, incurring the same misses in Phase 2 (as with RowClone) causes higher overall stall time for the application (because the latencies for the misses are serialized) than incurring them in Phase 1 (as in the baseline), resulting in RowClone's performance degradation compared to the baseline.

To address this problem, we introduce a variant of RowClone, RowClone-Zero-Insert (RowClone-ZI). RowClone-ZI not only zeroes out a page in DRAM but it also inserts a zero cache line into the processor cache corresponding to each cache line in the page that is zeroed out. By doing so, RowClone-ZI avoids the cache misses during both phase 1 (zeroing operation) and phase 2 (when the application accesses the cache lines of the zeroed page). As a result, it improves performance for all benchmarks, notably `forkbench` (by 66%) and *shell* (by 40%), compared to the baseline.

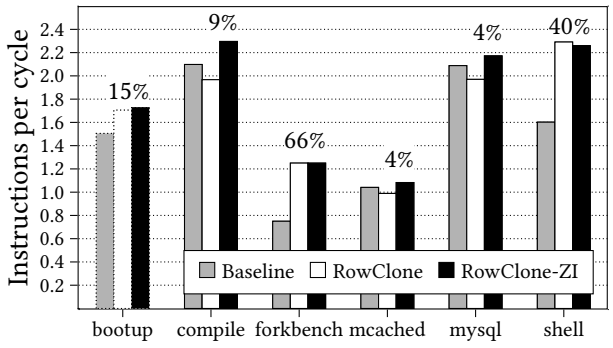


Figure 11: Performance improvement of RowClone and RowClone-ZI. Value on top indicates percentage improvement of RowClone-ZI over baseline.

Table 5 shows the percentage reduction in DRAM energy and memory bandwidth consumption with RowClone and RowClone-ZI compared to the baseline. While RowClone significantly reduces both energy and memory bandwidth consumption for *bootup*, *forkbench* and *shell*, it has negligible impact on both met-

rics for the remaining three benchmarks. The lack of energy and bandwidth benefits in these three applications is due to serial execution caused by the the cache misses incurred when the processor accesses the zeroed out pages (i.e., *phase 2*, as described above), which also leads to performance degradation in these workloads (as also described above). RowClone-ZI, which eliminates the cache misses in *phase 2*, significantly reduces energy consumption (between 15% to 69%) and memory bandwidth consumption (between 16% and 81%) for all benchmarks compared to the baseline. We conclude that RowClone-ZI can effectively improve performance, memory energy, and memory bandwidth efficiency in page copy and initialization intensive single-core workloads.

Application	Energy Reduction		Bandwidth Reduction	
	RowClone	+ZI	RowClone	+ZI
bootup	39%	40%	49%	52%
compile	-2%	32%	2%	47%
forkbench	69%	69%	60%	60%
mcached	0%	15%	0%	16%
mysql	-1%	17%	0%	21%
shell	68%	67%	81%	81%

Table 5: DRAM energy and bandwidth reduction due to RowClone and RowClone-ZI (indicated as +ZI)

7.4 Multi-core Evaluations

As RowClone performs bulk data operations completely within DRAM, it significantly reduces the memory bandwidth consumed by these operations. As a result, RowClone can benefit other applications running concurrently on the same system. We evaluate this benefit of RowClone by running our copy/initialization-intensive applications alongside memory-intensive applications from the SPEC CPU2006 benchmark suite [51] (i.e., those applications with last-level cache MPKI greater than 1). Table 6 lists the set of applications used for our multi-programmed workloads.

Copy/Initialization-intensive benchmarks
bootup, *compile*, *forkbench*, *mcached*, *mysql*, *shell*
Memory-intensive benchmarks from SPEC CPU2006
bzip2, *gcc*, *mcf*, *milc*, *zeusmp*, *gromacs*, *cactusADM*, *leslie3d*, *namd*, *gobmk*, *dealII*, *soplex*, *hmmer*, *sjeng*, *GemsFDTD*, *libquantum*, *h264ref*, *lbm*, *omnetpp*, *astar*, *wrf*, *sphinx3*, *xalancbmk*

Table 6: List of benchmarks used for multi-core evaluation

We generate multi-programmed workloads for 2-core, 4-core and 8-core systems. In each workload, half of the cores run copy/initialization-intensive benchmarks and the remaining cores run memory-intensive SPEC benchmarks. Benchmarks from each category are chosen at random.

Figure 12 plots the performance improvement due to RowClone and RowClone-ZI for the 50 4-core workloads we evaluated (sorted based on the performance improvement due to RowClone-ZI). Two conclusions are in order. First, although RowClone degrades performance of certain 4-core workloads (with *compile*, *mcached* or *mysql* benchmarks), it significantly improves performance for all other workloads (by 10% across all workloads). Second, like in our single-core evaluations (Section 7.3), RowClone-ZI eliminates the performance degradation due to RowClone and consistently outperforms both the baseline and RowClone for all workloads (20% on average).

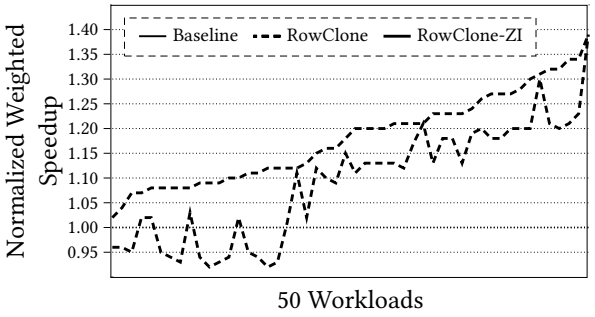


Figure 12: System performance improvement of RowClone for 4-core workloads

Table 7 shows the number of workloads and six metrics that evaluate the performance, fairness, memory bandwidth and energy efficiency improvement due to RowClone compared to the baseline for systems with 2, 4, and 8 cores. For all three systems, RowClone significantly outperforms the baseline on all metrics.

Number of Cores	2	4	8
Number of Workloads	138	50	40
Weighted Speedup [14] Improvement	15%	20%	27%
Instruction Throughput Improvement	14%	15%	25%
Harmonic Speedup [35] Improvement	13%	16%	29%
Maximum Slowdown [12, 29, 30] Reduction	6%	12%	23%
Memory Bandwidth/Instruction [49] Reduction	29%	27%	28%
Memory Energy/Instruction Reduction	19%	17%	17%

Table 7: Effect of RowClone on multi-core performance, fairness, bandwidth, and energy

To provide more insight into the benefits of RowClone on multi-core systems, we classify our copy/initialization-intensive benchmarks into two categories: 1) Moderately copy/initialization-intensive (*compile*, *mcached*, and *mysql*) and highly copy/initialization-intensive (*bootup*, *forkbench*, and *shell*). Figure 13 shows the average improvement in weighted speedup for the different multi-core workloads, categorized based on the number of highly copy/initialization-intensive benchmarks. As the trends indicate, the performance improvement increases with increasing number of such benchmarks for all three multi-core systems, indicating the effectiveness of RowClone in accelerating bulk copy/initialization operations.

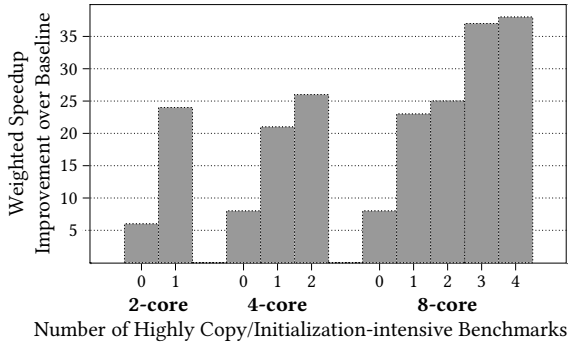


Figure 13: Effect of increasing copy/initialization intensity

We conclude that RowClone is an effective mechanism to improve system performance, energy efficiency and bandwidth efficiency of future, bandwidth-constrained multi-core systems.

7.5 Memory-Controller-based DMA

One alternative way to perform a bulk data operation is to use the memory controller to complete the operation using the regular DRAM interface (similar to some prior approaches [28, 59]). We refer to this approach as the memory-controller-based DMA (MC-DMA). MC-DMA can potentially avoid the cache pollution caused by inserting blocks (involved in the copy/initialization) unnecessarily into the caches. However, it still requires data to be transferred over the memory bus. Hence, it suffers from the large latency, bandwidth, and energy consumption associated with the data transfer. Because the applications used in our evaluations do not suffer from cache pollution, we expect the MC-DMA to perform comparably or worse than the baseline. In fact, our evaluations show that MC-DMA degrades performance compared to our baseline by 2% on average for the six copy/initialization intensive applications (16% compared to RowClone). In addition, the MC-DMA does not conserve any DRAM energy, unlike RowClone.

8. RELATED WORK

To our knowledge, this is the first paper to propose a concrete mechanism to perform bulk data copy and initialization operations completely in DRAM. In this section, we discuss related work and qualitatively compare them to RowClone.

Patents on Data Copy in DRAM. Several patents [5, 15, 37, 38] propose the abstract notion that the row buffer can be used to copy data from one row to another. These patents have several drawbacks. First, they do not provide any concrete mechanism used to perform the copy operation. Second, while using the row buffer to copy data between two rows is possible only when the two rows are within the same subarray, these patents make no such distinction. Third, these patents do not discuss the support required from the other layers of the system to realize a working system. Finally, these patents do not provide any concrete evaluation to show the benefits of performing copy operations in-DRAM. In contrast, RowClone is more generally applicable, and we discuss concrete changes required to all layers of the system stack from the DRAM architecture to the system software.

Offloading Copy/Initialization Operations. Prior works [28, 59] have proposed mechanisms to 1) offload bulk data copy/initialization operations to a separate engine, and 2) reduce the impact of pipeline stalls (by waking up instructions dependent on a copy operation as soon as the necessary blocks are copied without waiting for the entire copy operation to complete), and 3) reduce cache pollution by using hints from software to decide whether to cache blocks involved in the copy or initialization. While we have already shown the effectiveness of RowClone compared to offloading bulk data operations to a separate engine (Section 7.5), the techniques to reduce pipeline stalls and cache pollution [28] can be naturally combined with RowClone to further improve performance.

Bulk Memory Initialization. Jarrod et al. [24] propose a mechanism for avoiding the memory access required to fetch uninitialized blocks on a store miss by using a specialized cache to keep track of uninitialized regions of memory. RowClone can potentially be combined with this mechanism. While Jarrod et al.'s approach can be used to reduce bandwidth consumption for irregular initialization (initializing different pages with different values), RowClone can be used to push regular initialization (e.g., initializing multiple pages with the same values) to DRAM, thereby freeing up the CPU to perform other useful operations.

Yang et al. [58] propose to reduce the cost of zero initialization by 1) using non-temporal store instructions to avoid cache pollution, and 2) using idle cores/threads to perform zeroing ahead of time. While the proposed optimizations reduce the negative performance impact of zeroing, their mechanism does not reduce memory bandwidth consumption of the bulk zeroing operations. In contrast, RowClone significantly reduces the memory bandwidth consumption and the associated energy overhead.

Compute-in-Memory. Prior works (e.g., [32, 42]) have investigated mechanisms to add logic closer to memory to perform bandwidth-intensive computations (e.g., SIMD vector operations) more efficiently. The main limitation of such approaches is that adding logic to DRAM significantly increases the cost of DRAM. In contrast, RowClone exploits DRAM's internal organization and operation to perform bandwidth-intensive copy and initialization operations quickly and efficiently in DRAM with low cost.

9. CONCLUSION

We introduced RowClone, a new technique for exporting bulk data copy and initialization operations to DRAM. Based on the key observation that DRAM can internally transfer multiple kilo-bytes of data between the DRAM cells and the row buffer, our fastest mechanism copies an entire row of data between rows that share a row buffer, with very few changes to the DRAM architecture, while leading to significant reduction in the latency and energy of performing bulk copy/initialization. We also propose a more flexible mechanism that uses the internal data bus of a chip to efficiently copy data between different banks within a chip. Our evaluations using copy and initialization intensive applications show that RowClone can significantly reduce memory bandwidth consumption for both single-core and multi-core systems (by 28% on average for 8-core systems), resulting in significant system performance improvement and memory energy reduction (27% and 17%, on average, for 8-core systems).

We conclude that our approach of performing bulk copy and initialization completely in DRAM is effective in improving both system performance and energy efficiency for future, bandwidth-constrained, multi-core systems. We hope that greatly reducing the bandwidth, energy and performance cost of bulk data copy and initialization can lead to new and easier ways of writing data-intensive applications that would otherwise need to be designed to avoid bulk data copy and initialization operations.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable feedback and suggestions, which improved the paper's quality. We thank Brian Hirano from Oracle and Uksong Kang, Churoo Park, Jung-Bae Lee, and Joo Sun Choi from Samsung for their helpful comments at various stages of this project. We acknowledge the members of the SAFARI and LBA research groups for their feedback and the stimulating research environment they provide. We acknowledge the support of AMD, IBM, Intel, Oracle, Qualcomm, and Samsung. This research was partially supported by NSF (CCF-0953246, CCF-1147397, CCF-1212962), Intel University Research Office Memory Hierarchy Program, Intel Science and Technology Center for Cloud Computing, and Semiconductor Research Corporation.

REFERENCES

[1] Bochs IA-32 emulator project. `http://bochs.sourceforge.net/`.

[2] Memcached: A high performance, distributed memory object caching system. `http://memcached.org`.

[3] MySQL: An open source database. `http://www.mysql.com`.

[4] Wind River Simics full system simulation. http://www.windriver.com/products/simics/.

[5] J. Ahn. Memory device having page copy mode. U.S. patent 5886944, 1999.

[6] J. Bent et al. PLFS: A checkpoint filesystem for parallel applications. In *SC*, 2009.

[7] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*, page 388. O'Reilly Media, 2005.

[8] F. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. In *OSDI*, 1999.

[9] J. Chow et al. Shredding Your Garbage: Reducing data lifetime through secure deallocation. In *USENIX SS*, 2005.

[10] K. Constantinides et al. Software-Based Online Detection of Hardware Defects: Mechanisms, architectural support, and evaluation. In *MICRO*, 2007.

[11] K. Constantinides et al. Online Design Bug Detection: RTL analysis, flexible mechanisms, and evaluation. In *MICRO*, 2008.

[12] Reetuparna Das et al. Application-aware prioritization mechanisms for on-chip networks. In *MICRO*, 2009.

[13] A. M. Dunn et al. Eternal Sunshine of the Spotless Machine: Protecting privacy with ephemeral channels. In *OSDI*, 2012.

[14] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, (3), 2008.

[15] P. B. Gillingham and R. Torrance. DRAM page copy method. U.S. patent 5625601, 1997.

[16] J. A. Halderman et al. Lest We Remember: Cold boot attacks on encryption keys. In *USENIX SS*, 2008.

[17] K. Harrison and S. Xu. Protecting cryptographic keys from memory disclosure attacks. In *DSN*, 2007.

[18] M. Horiguchi and K. Itoh. *Nanoscale Memory Repair*. Springer, 2011.

[19] IBM Corporation. Enterprise Systems Architecture/390 Principles of Operation, 2001.

[20] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. April 2012.

[21] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3A, chapter 11, page 12. April 2012.

[22] E. Ipek et al. Self Optimizing Memory Controllers: A reinforcement learning approach. In *ISCA*, 2008.

[23] T. B. Jablin et al. Automatic CPU-GPU communication management and optimization. In *PLDI*, 2011.

[24] L. A. Jarrod et al. Avoiding Initialization Misses to the Heap. In *ISCA*, 2002.

[25] JEDEC. Server memory roadmap. http://www.jedec.org/sites/default/files/Ricki_Dee_Williams.pdf.

[26] JEDEC. Standard No. 21-C. Annex K: Serial Presence Detect (SPD) for DDR3 SDRAM Modules, 2011.

[27] JEDEC. DDR3 SDRAM, JESD79-3F, 2012.

[28] X. Jiang et al. Architecture support for improving bulk memory copying and initialization performance. In *PACT*, 2009.

[29] Y. Kim et al. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA*, 2010.

[30] Y. Kim et al. Thread Cluster Memory Scheduling: Exploiting differences in memory access behavior. In *MICRO*, 2010.

[31] Y. Kim et al. A case for exploiting subarray-level parallelism (SALP) in DRAM. In *ISCA*, 2012.

[32] P. M. Kogge. EXECUBE - A new architecture for scaleable MPPs. In *ICPP*, 1994.

[33] H. A. Lagar-Cavilla et al. SnowFlock: Rapid virtual machine cloning for cloud computing. In *EuroSys*, 2009.

[34] D. Lee et al. Tiered-Latency DRAM: A low-latency and low-cost DRAM architecture. In *HPCA*, 2013.

[35] Kun Luo et al. Balancing thoughput and fairness in SMT processors. In *ISPASS*, 2001.

[36] Micron. DDR3 SDRAM system-power calculator, 2011.

[37] D. M. Morgan and M. A. Shore. DRAMs having on-chip row copy circuits for use in testing and video imaging and method for operating same. U.S. patent 5440517, 1995.

[38] Kaori Mori. Semiconductor memory device including copy circuit. U.S. patent 5854771, 1998.

[39] S. P. Muralidhara et al. Reducing memory interference in multi-core systems via application-aware memory channel partitioning. In *MICRO*, 2011.

[40] O. Mutlu et al. Efficient Runahead Execution: Power-efficient memory latency tolerance. *IEEE Micro*, 26(1), 2006.

[41] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware. In *USENIX SC*, 1990.

[42] D. Patterson et al. A case for Intelligent RAM. *IEEE Micro*, 17(2), 1997.

[43] Rambus. DRAM power model, 2010.

[44] M. Rosenblum et al. The impact of architectural trends on operating system performance. In *SOSP*, 1995.

[45] M. E. Russinovich et al. *Windows Internals*, page 701. Microsoft Press, 2009.

[46] G. Sandhu. DRAM scaling and bandwidth challenges. In *WETI*, 2012.

[47] R. F. Sauers et al. *HP-UX 11i Tuning and Performance*, chapter 8. Memory Bottlenecks. Prentice Hall, 2004.

[48] A. Singh. *Mac OS X Internals: A Systems Approach*. Addison-Wesley Professional, 2006.

[49] S. Srinath et al. Feedback Directed Prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, 2007.

[50] S. M. Srinivasan et al. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX ATC*, 2004.

[51] Standard Performance Evaluation Corporation. SPEC CPU2006. http://www.spec.org/cpu2006.

[52] L. Subramanian et al. MISE: Providing performance predictability and improving fairness in shared main memory systems. In *HPCA*, 2013.

[53] K. Sudan et al. Micro-pages: Increasing DRAM efficiency with locality-aware data placement. In *ASPLOS*, 2010.

[54] A. N. Udipi et al. Rethinking DRAM design and organization for energy-constrained multi-cores. In *ISCA*, 2010.

[55] C. A. Waldspurger. Memory resource management in VMware ESX server. In *OSDI*, 2002.

[56] F.A. Ware and C. Hampel. Improving power and data efficiency with threaded memory modules. In *ICCD*, 2006.

[57] B. Wester et al. Operating system support for application-specific speculation. In *EuroSys*, 2011.

[58] X. Yang et al. Why Nothing Matters: The impact of zeroing. In *OOPSLA*, 2011.

[59] L. Zhao et al. Hardware support for bulk data movement in server platforms. In *ICCD*, 2005.

[60] H. Zheng et al. Mini-rank: Adaptive DRAM architecture for improving memory power efficiency. In *MICRO*, 2008.