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

Vivek Seshadri Yoongu Kim[†] Chris Fallin[†] Donghyuk Lee[†]
Rachata Ausavarungnirun[†] Gennady Pekhimenko Yixin Luo
Onur Mutlu[†] Phillip B. Gibbons[*]
Michael A. Kozuch[*] Todd C. Mowry

April 2013
CMU-CS-13-108

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*]Intel Pittsburgh
[†]Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh

This research is supported by grants from the National Science Foundation (Awards CCF-0953246 and CCF-1212962), the Gigascale Systems Research Centre, Intel and Samsung.

Keywords: Memory bandwidth, Bulk data copy, Bulk data initialization, Performance

Abstract

Many programs initialize or copy large amounts of memory data. Initialization and copying are forms of memory operations that do not require computation in order to derive their data-values – they either deal with known data-values (e.g., initialize to zero) or simply move data-values that already exist elsewhere in memory (e.g., copy). Therefore, initialization/copying can potentially be performed entirely within the main memory subsystem without involving the processor or the DMA engines. Unfortunately, existing main memory subsystems are unable to take advantage of this fact. Instead, they unnecessarily transfer large amounts of data between main memory and the processor (or the DMA engine) – thereby consuming large amounts of latency, bandwidth, and energy.

In this paper, we make the key observation that DRAM chips – the predominant substrate for main memory – *already* have the capability to transfer large amounts of data within themselves. Internally, a DRAM chip consists of *rows* of bits and a *row-buffer*. To access data from any portion of a particular row, the DRAM chip transfers the entire row (e.g., 4 Kbits) into the equally-sized row-buffer, and vice versa. While this internal data-transfer between a row and the row-buffer occurs in bulk (i.e., all 4 Kbits at once), an external data-transfer (to/from the processor) is severely serialized due to the very narrow width of the DRAM chip's data-pins (e.g., 8 bits). Our key idea is to utilize and extend the row-granularity data-transfer in order to quickly initialize or move data one row at a time within a DRAM chip. We call this new mechanism ***RowClone***. By making several relatively unintrusive changes to DRAM chip design (0.026% die-size increase), we accelerate a one-page (4 KByte) copying operation by 11.5x, and a one-page zeroing operation by 5.8x, while also conserving memory bandwidth. In addition, we achieve large energy reductions – 41.5x/74.4x energy reductions for one-page zeroing/copying, respectively. We show that RowClone improves performance on an 8-core system by 27% averaged across 64 copy/initialization-intensive workloads.

1 Introduction

Modern main memory subsystems are designed to quickly and efficiently provide the data the processor requires to execute programs. In this work, we focus on an important class of memory accesses that *copy* bulk data (e.g., moving data between buffers) or *initialize* bulk data to zero (e.g., resetting program state at program startup). These operations have two characteristics that distinguish them from other types of memory accesses. First, these operations do not require any computation to derive their data-values, because they only move existing data-values or reset memory locations to known data-values. Second, they typically deal with large contiguous regions of memory which can be kilobytes or megabytes in size.

Due to these two characteristics, we observe that there is opportunity to perform bulk initialization/copying entirely within the main memory subsystem (e.g., DRAM chips) in a way that is self-contained and efficient. First, because these operations do not require computation, the processor need not be involved; the values to write to memory are either statically known (initialization to zero or other values) or already elsewhere in memory (copying). Second, because these operations involve large amounts of data, the main memory subsystem can perform them more efficiently by parallelizing the operations within itself.

Unfortunately, existing systems cannot offload bulk initialization/copying to the main memory subsystem. Instead, existing systems must unnecessarily shuttle large quantities of data back and forth between the processor and main memory to perform these operations. This causes three specific problems that degrade the performance and energy-efficiency of the system. First, large data-transfers over the memory bus (connecting the processor to main memory) incur a large latency that reduces the performance of the program performing the bulk initialization/copying. Second, large data-transfers over the memory bus consume a significant amount of memory bandwidth that could have been used to serve memory requests from other programs co-running on the same system. Third, the unnecessary movement of large data up and down the memory hierarchy wastes a large amount of energy.

In this work, we introduce *RowClone*, a new technique that can efficiently perform bulk initialization and bulk copying within the main memory subsystem. By making small changes to the existing main memory design, RowClone enables direct in-memory bulk data-transfer in a way that the data-transfer bypasses the memory bus and the processor. RowClone exposes the bulk data-transfer capability of the main memory subsystem as hardware primitives that can be taken advantage of by software programs in order to accelerate and conserve the energy of bulk initialization/copying.

RowClone is based on an observation we make about DRAM chips, which are the predominant building block of modern main memory subsystems. This observation –that data is already transferred in bulk at the row granularity within parts of the DRAM chip– leads to a mechanism that enables the offloading of bulk initialization/copying to a DRAM chip where they can be performed efficiently. To understand how we can accelerate bulk initialization/copying within DRAM, we briefly examine how a DRAM chip operates.

Brief DRAM Background. A DRAM chip comprises a two-dimensional array of single-bit storage units called DRAM *cells*, as shown in Figure 1.[1] Cells that are horizontally adjacent to each other have consecutive physical addresses. In other words, all cells in the horizontal direction (e.g., 4Kbit) – collectively referred to as a DRAM *row* – correspond to a contiguous 4Kbit chunk of the physical address space. A DRAM chip also has an auxiliary structure called the *row-buffer* (4Kbit) that acts as an I/O interface between the DRAM chip's rows and the external memory bus. When there is an access to a row (either read or write), the DRAM chip loads the *entire* row into the row-buffer from where data is accessed through the DRAM chip's narrow data-pins (8bit) and transferred onto the external memory bus.

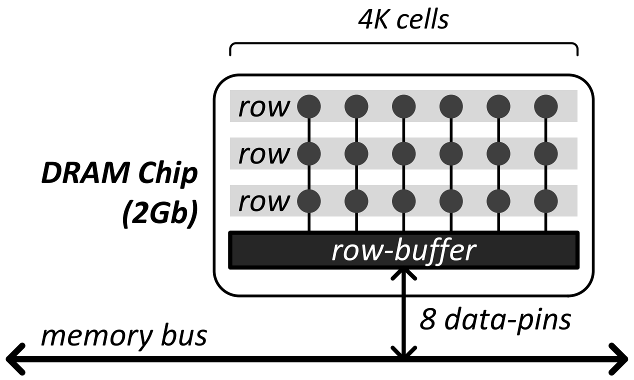


Figure 1: High-level abstract view:[1] 2Gbit DRAM chip

Key Observation. Our key observation is that a DRAM chip *already* implements bulk data-transfer between any of its rows and the row-buffer – i.e., the entire 4Kbit row is loaded into the 4Kbit row-buffer *all at once* in just a single-step (and vice versa). In fact, the DRAM chip *cannot* selectively load only a fraction of a row into the row-buffer (and vice versa), because all cells within a row operate in lockstep as a single unit. Therefore, in a DRAM chip, the full-width of a row (4Kbit) is the *only* granularity at which a row and the row-buffer can transfer data between each other. On the other hand, large data transfers between the row-buffer and the external memory bus experience severe serialization because the DRAM chip's data-pins are very narrow (8bit).

Key Idea. Our key idea is to enable bulk row-to-row transfer within a DRAM chip by leveraging the large internal bandwidth that already exists between the rows and the row-buffer. We propose an efficient two-step[2] mechanism to transfer data between two rows, i.e., the source and destination rows: *(i)* source-row to row-buffer, *(ii)* row-buffer to destination-row. While existing main memory subsystems already perform these two steps, they also perform two additional steps in-between that are very costly: *(i)* row-buffer to processor, *(ii)* processor to row-buffer. In contrast, our mechanism allows the main memory subsystem to omit the costly and unnecessary transfers between the DRAM chip and the processor while executing bulk initialization/copying.

[1]This figure abstracts away lower-level details of a DRAM chip, such as banks and subarrays, which we will explain in §3.3.

[2]A single-step mechanism – in which the source-row writes *directly* into the destination-row, as opposed to writing into the row-buffer – is not possible at all due to circuit-level reasons that we explain in § 4.1.

Overview of Mechanism. Our two-step mechanism builds on an already existing DRAM operation called *activation*. When the memory controller activates a row in a DRAM chip (e.g., source-row), the row is connected to the row-buffer and the row's data is captured in the row-buffer – thereby accomplishing the first step. At this point, we introduce a new DRAM operation called *de-activation* that disconnects a row from the row-buffer, *while still preserving the data captured in the row-buffer.* Using this new DRAM operation, the memory controller can then accomplish the second step by de-activating the source-row and subsequently activating the destination-row – thereby loading the row-buffer's data (i.e., the source-row's data) into the destination-row.

We make the following contributions.

- We show that modern DRAM chips internally transfer data at a large granularity (4Kb), presenting the opportunity to perform bulk data copying and initialization entirely in DRAM.

- We propose RowClone, a simple mechanism that allows the processor to export bulk data copy and initialization operations to the memory controller, which in turn performs the operation fully in DRAM. RowClone reduces the latency of a 4KB copy operation by 11.5x and a 4KB clear operation by 5.8x. RowClone requires minimal non-intrusive changes to the DRAM architecture.

- We analyze the benefits of using RowClone with a set of copy/initialization-intensive workloads. We show that on an 8-core system, RowClone improves performance by 29%, and reduces energy consumption by 37% and memory bandwidth by 28%, averaged across 64 workloads.

2 Motivation

In this section, we motivate the need for bulk initialization and copying (§2.1), demonstrate the prevalence of these operations quantitatively with two case studies (§2.2), discuss shortcomings of existing memory systems (§2.3), and state our goal of performing these operations quickly and efficiently (§2.4).

2.1 Cases for Bulk Initialization/Copying

Operating systems (OS) manage main memory at the granularity of a *page*, whose size is typically 4-8KB. There are a number of cases in which the OS must initialize a page or copy one page to another page. We present six different scenarios that lead to such page initialization/copying.

System/Application Startup. At bootup time, many pages are initialized as the operating system is loaded into memory from disk. Similarly, when an application starts to run and begins to fill large regions of memory, pages are demand-allocated by the operating system and must be zeroed [30].

Secure Memory Deallocation. For security reasons, the OS may clear (i.e., zero-out) an application's memory before deallocating it [4, 5, 35]. This is done to prevent a malicious application/user from obtaining sensitive data (e.g., passwords, cryptographic keys) stored in memory [7, 8].

Copy-on-Write and Process Cloning. The `fork` system call is used by the operating system to create new processes. The semantics of `fork` require the entire address space of the process being forked (parent process) to be copied over to the newly created process (child process). Modern operating systems avoid copying the entire address space by sharing the pages between the two processes until either one writes to a shared page. At this point, the operating system allocates a new page for the writing process and copies the contents of the shared page to the new page. This technique is referred to as *copy-on-write*, in which page copies are deferred for as long as possible [30]. In spite of such optimizations, a `fork` may still lead to a large number of page copies.

Memory Checkpointing/Cloning. Large-scale applications with long runtimes (e.g., days or weeks) regularly checkpoint all of their main memory data so that they can recover from a known state after a possible physical machine failure. Checkpointing requires the memory data to be either copied to another memory location or flushed out to disk. In both cases, the application may have to be temporarily to prevent it from making any further modifications to the memory data until the whole memory image is checkpointed.

Virtual Machine Deduplication. In many datacenter or server environments, multiple virtual machines (VM) running user applications are consolidated on the same server. Prior works have proposed to share pages that contain the same data across virtual machines in order to conserve physical memory [31]. In such scenarios, any write to such a shared page will trigger a copy operation to provide the writing application with a private version of the page [37, 38]. Similar to the *secure memory deallocation* case above, the hypervisor in such systems zeroes the data of a page after deallocation to protect the privacy of data of a VM [34].

Graphics Processing. Graphics applications perform z-buffer and frame buffer clearing in bulk before generating new frames by setting pages corresponding to the buffer to a particular value [24]. GPUs often employ special hardware to record these cleared pages in order to avoid writing this data to memory. GPU applications also often require bulk data copy: before starting a GPU application, the OS kernel has to copy data from the host to the device address spaces [11], even if the CPU and GPU are integrated and share a memory controller. In such an integrated system with shared physical DRAM, an in-DRAM copy operation could accelerate this data transfer.

2.2 System Workload Case Studies

Figure 2 shows the fraction of memory accesses that are triggered due to page initialization or page copying operations for four different system workloads (a full description of methodology is in §10). The first, **bootup**, is a trace of a portion of system boot for an ordinary Linux system. A non-negligible fraction (23%) of memory operations occur due to page zeroing, which the kernel performs whenever a new page is demand-allocated by a process. System bootup incurs many such operations because many programs are being loaded and are initializing their in-memory data. Another, **forkset**, is a trace of a system running a small microbenchmark which sets up a large (64MB) array with random data, and then repeatedly *(i)* forks a child process and *(ii)* randomly writes to many pages of this large array from the child process. This usage pattern causes many copy-on-write allocations of new data pages which are copied from the pages of the original (parent) process. Overall, 82% of memory operations are due to the in-kernel page copy routine in this workload. The **shell** workload (a Unix shell script executing many commands) exhibits a mix of both copies and page zeroing operations. Our results indicate that page initialization and copying occur frequently enough to motivate mechanisms that accelerate them and make them more energy-efficient.

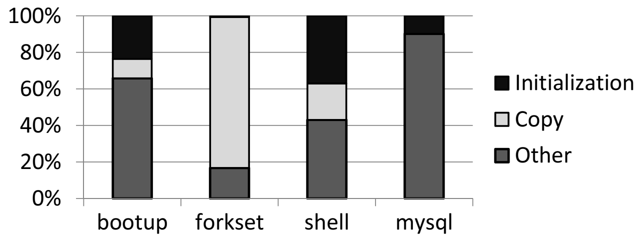
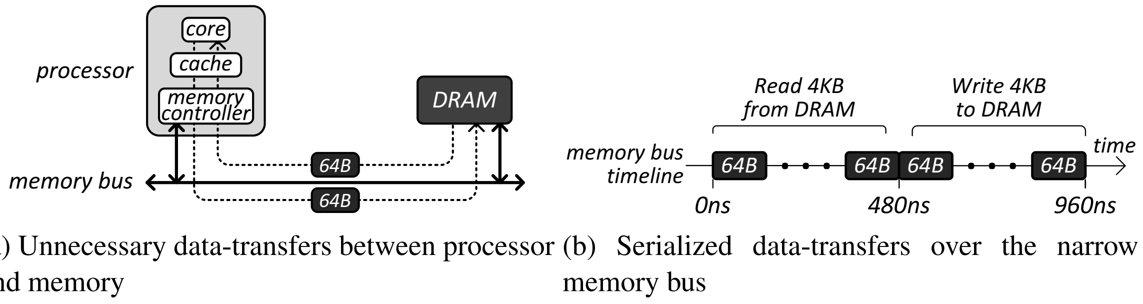


Figure 2: Memory access composition of 4 benchmarks

2.3 Shortcomings of Existing Systems

In existing systems, page initialization/copying are performed explicitly by the processor on behalf of the software. As Figure 3 shows, when copying data from one page to another, the processor reads the data from the source page one cache-line (64B) at a time and writes it to the destination page one cache-line (64B) at a time. This is very inefficient since data is unnecessarily transferred between the processor and DRAM through the memory bus and the processor's cache hierarchy (Figure 3a). Similarly, to initialize a page, the processor repeatedly writes zeros to every cache-line (64B) of the destination page. This has three shortcomings.



(a) Unnecessary data-transfers between processor and memory

(b) Serialized data-transfers over the narrow memory bus

Figure 3: Inefficient page copy in existing systems[3]

First, since a page is initialized/copied one cache-line at a time, the operation incurs a long latency. For a page copy, as shown in Figure 3b, the memory bus – just by itself – incurs a large latency of 480ns, even if we ignore other sources of latency, e.g., the DRAM chip itself. Furthermore, this latency becomes proportionally larger depending on the number of pages that are copied and the system's page size (which can be as large as 4MB [20] or even 1GB [9]). Such a

[3]For latency calculations, we assume DDR3-1066 timing parameters [13]. In addition, we assume that the memory bus is connected to eight DRAM chips in parallel (i.e., eight chips per *rank*), such that the memory bus width is 64 bit (= 8 chips$\times$8 bit/chip). Therefore, it takes eight transfers over the memory bus to send/receive a single 64 Byte cache-line (64 Byte = 8 transfers$\times$64 bit/transfer).

large latency directly affects the performance of the application that is waiting for the initialization/copying to complete. In fact, this is the primary reason why current systems try to avoid such operations as much as possible. Second, since all the cache-lines are transferred from main memory to the processor, initialization/copying consumes a large amount of memory bandwidth on the memory bus (Figure 3b). This indirectly degrades the system performance by stealing bandwidth away from other applications that are co-running in the same system. Third, since both initialization/copying do not require any computation to derive their data-values, the movement of data between the processor and memory is unnecessary. This extraneous data movement in turn leads to wasted energy at the memory bus, the memory controller, the on-chip cache hierarchy, and the processor (Figure 3a).

2.4 Our Goal

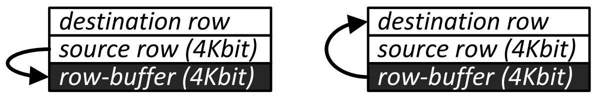
Our goal in this work is to design mechanisms to perform page initialization/copying in a manner that avoids the above shortcomings. To this end, we develop a mechanism to export such operations to main memory (DRAM chips), where bulk initialization/copying can be performed at the large granularity of a DRAM chip's row. We discuss the size-alignment issues between a page (4KByte) and a DRAM chip's row (4Kbit) in §7.1. But, until then, we will assume that a single page is mapped to a single DRAM chip's row (which is not always the case) and focus primarily on developing mechanisms for row-sized bulk initialization/copying.

2.5 Overview of Our Mechanisms

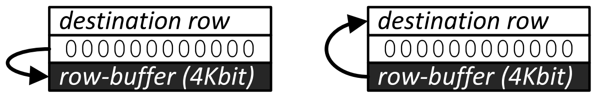
We develop two mechanisms, row-copying and row-initialization, which are summarized at a high-level in Figure 4. First, row-copying consists of two-steps as shown in Figure 4a: *(i)* loading the source row into the row-buffer and *(ii)* loading the row-buffer into the destination row. Second, row-initialization is similar to row-copying and relies on the same framework of utilizing the row-buffer to transfer data between two rows. As Figure 4b shows, row-initialization also consists of two-steps: *(i)* loading a zeroed row into the row-buffer and *(ii)* loading the row-buffer into the destination row. (For now, we assume that a DRAM chip sets aside one row that is zeroed during system bootup and always remains zeroed.) In the next section, we provide background on a DRAM chip's internal operation that is required to understand our mechanisms.

3 DRAM Background

In this section, we provide a bottom-up view of how a DRAM chip operates. We first take a closer look at the structure of the rows and the row-buffer. Then we will provide a step-by-step operational example of how a row's data is accessed through the row-buffer.



(a) Row-copying: source row to row-buffer, row-buffer to dest. row

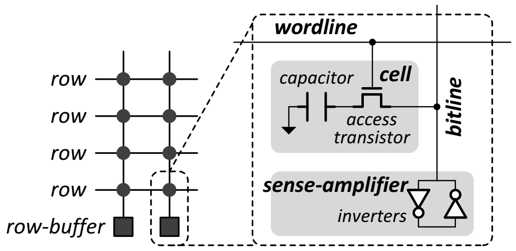


(b) Row-initialize: zeroed row to row-buffer, row-buffer to dest. row

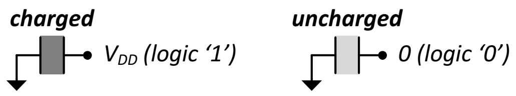
Figure 4: Overview of our mechanisms

3.1 A Closer Look: Row & Row-Buffer

As we have explained, a DRAM chip consists of many rows and a row-buffer. Similar to how a row is a horizontal collection of *cells*, the row-buffer is a horizontal collection of *sense-amplifiers*, as shown in Figure 5a. Therefore, loading a row into the row-buffer is equivalent to loading all cells of the row into their respective sense-amplifier in the row-buffer. In addition, a DRAM chip has two sets of wires that are involved in loading a cell into a sense-amplifier – *bitlines* and *wordlines* – as shown in Figure 5a. The bitline is a vertical wire that connects a cell to the sense-amplifier. On the other hand, each row has a horizontal wire called the wordline that determines whether or not the cells of the row are connected to the bitlines. When the DRAM chip turns on a wordline (i.e., raises the wordline voltage to V_{DD}), all cells of the corresponding row are connected to the bitlines and is loaded into the sense-amplifiers.



(a) Low-level components of a DRAM chip



(b) Two states of a cell's capacitor

Figure 5: A closer look at a row and a row-buffer

Figure 5a also shows the internals of a cell and a sense-amplifier. A cell stores data in the form of charge in its *capacitor*. Depending on the data stored in the cell, the capacitor can either be charged or uncharged. Without loss of generality, we will assume that a charged capacitor maps to binary value of '1' and depict it using a dark capacitor (Figure 5b) throughout the rest of this paper. By raising the cell's wordline voltage to V_{DD}, we can switch on the cell's *access transistor* (Figure 5a) that then connects the cell's capacitor to the bitline. At this point, the sense-amplifier detects the capacitor's charge through the bitline and latches the data in its cross-coupled inverters

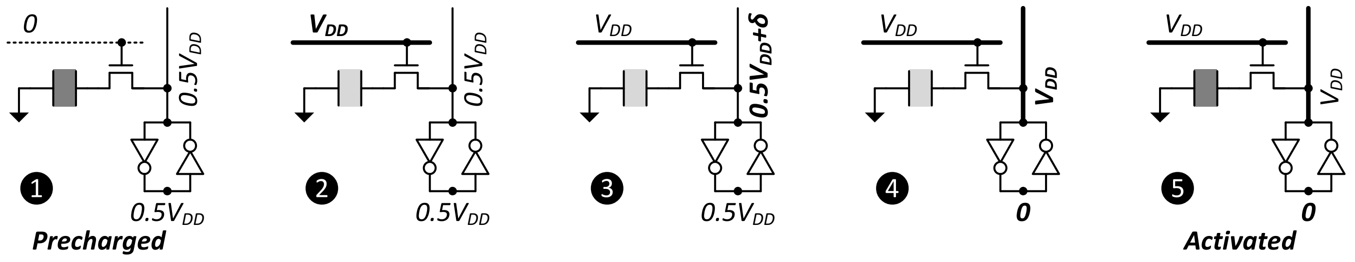


Figure 6: Activation (37.5ns): step-by-step operation of loading a cell into the sense-amplifier

(Figure 5a). Since the cell's capacitor can store only a very small amount of charge, the data it represents is fragile. That is why the sense-amplifier exists; it is a specialized piece of circuitry that can reliably detect and latch the data from a cell. Subsequently, all accesses to the cell's data (both reads and writes) are served by the sense-amplifier on behalf of the cell.

3.2 Operational Example: Accessing a Cell

Starting with a cell that initially stores a logic value of '1', we will now walk through an operational example of how a cell's data is accessed. This consists of three steps. First, the cell's data is loaded into the sense-amplifier in a step that is called *activation*. Second, depending on whether the access is a *read* or a *write*, the DRAM chip either retrieves the sense-amplifier's data or overwrites it with new data. In the case of a write, the sense-amplifier in turn updates the cell with the new data. Third, after serving the access from the sense-amplifier, the DRAM chip clears (i.e., empties) the sense-amplifier in a step that is called *precharging*.

The memory controller guides a DRAM chip through the three steps (activation, read/write, precharging) by issuing DRAM *commands* [13] to the DRAM chip: ACTIVATE, READ/WRITE, PRECHARGE. In the following, we discuss the three steps in detail.

Activation (Latency: 37.5ns [13]). The ACTIVATE command (issued along with the row's address) triggers the activation of a particular row of cells within a DRAM chip. Figure 6 shows the step-transitions of a cell and a sense-amplifier during activation. Initially, the cell is disconnected from the bitline and the bitline is at a voltage of $\frac{1}{2}V_{\text{DD}}$. This state is called the *precharged* state (❶). To access the cell, the wordline is raised, connecting the cell to the bitline (❷). Assuming that the cell is in the charged state, the cell is at a higher voltage (V_{DD}) than the bitline ($\frac{1}{2}V_{\text{DD}}$). This voltage difference causes the cell to share some of its charge with the bitline, slightly raising the bitline voltage to $\frac{1}{2}V_{\text{DD}}+\delta$ (❸). After detecting this small voltage increase on the bitline, the sense-amplifier starts to amplify it. Eventually, the sense-amplifier drives the bitline voltage all the way up to V_{DD} (❹). Since the DRAM cell is still connected to the bitline, it is also fully charged back to its original state (❺). The sense-amplifier drives the bitline voltage in the opposite direction (to 0) if the cell was initially in an uncharged state. In that case, the bitline shares some of its charge with the cell and the sense-amplifier eventually drives the bitline to a voltage of 0. The latency of activation is $\sim$37.5ns in modern DRAM chips.

Read/Write (Latency: 15ns [13]). The READ command retrieves the sense-amplifier's data and transfers it out of the DRAM chip. On the other hand, the WRITE command (issued along with the write data) overwrites the data latched in the sense-amplifier by flipping the bitline voltage (from V_{DD} to 0, or vice versa) through the DRAM chip's I/O circuitry (which we explain later in §3.3). This, in turn, overwrites the data stored in the cell by flipping its charge.

Precharging (Latency: 15ns [13]). After all accesses to the cell's data are served from the sense-amplifier, the PRECHARGE command triggers the precharging of a row. Figure 7 shows the step-transitions of a cell and a sense-amplifier during precharging. First, the wordline is lowered, disconnecting the cells from the bitlines. Second, the sense-amplifier drives the bitline to a voltage of $\frac{1}{2}V_{\text{DD}}$. The latency of precharging is $\sim$15ns in modern DRAM chips.

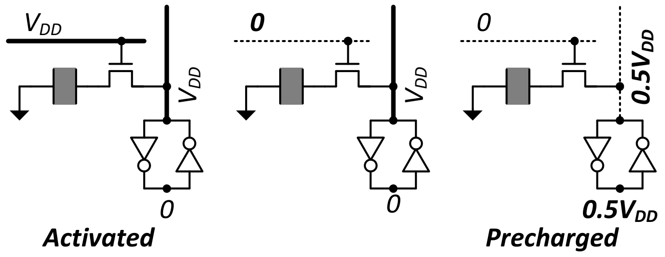


Figure 7: Precharging (15ns): step-by-step operation of disconnecting a cell from the sense-amplifier and emptying the sense-amplifier

3.3 Hierarchical Organization of a DRAM Chip

Until now, we have abstracted a DRAM chip as having a single row-buffer that is shared by all rows (e.g., 512K rows for a 4Gb DDR3 DRAM chip [13]). In practice, however, a DRAM chip has *many* row-buffers, each of which is shared by only a few rows (Figure 8), for reasons we soon explain. In addition, the 512K rows are grouped at different granularities to form a hierarchy within a DRAM chip. At the lowest level of the hierarchy is a *subarray* [17, 36], followed by a *bank*, then finally the DRAM chip itself. Therefore, given two rows within a DRAM chip, the rows can belong to *(i)* the same subarray, *(ii)* different subarrays of the same bank, or *(iii)* different banks altogether. For each of these three scenarios, the wiring between the rows is different. As a result, the data-transfer mechanism between the rows must also be different as well. In the following, we explain the reasoning behind the hierarchical organization of a DRAM chip and how different rows within that hierarchy are wired together, which directly relates to our goals and mechanisms of bulk initialization/copying within a DRAM chip.

Subarray (512 rows). If there were just a single row-buffer in the entire DRAM chip, then it would require very long bitlines – spanning the entire chip – that connect the row-buffer to all 512K rows. But, very long bitlines have a large propagation delay that significantly increases the DRAM chip's latency. To mitigate this problem, a DRAM chip adopts shorter bitlines by having many row-buffers, each of which is connected to a smaller number of rows (e.g., 512 [17, 36]). As Figure 8 (left) shows, a group of 512 rows and their communal row-buffer are collectively referred to as a *subarray*. Each subarray also has its own *multiplexer* that selectively connects a small portion of the wide row-buffer (4Kb) to the narrow *I/O bus* (64b) – the I/O bus is purposely

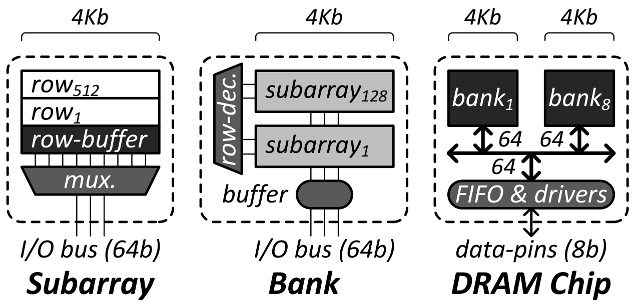


Figure 8: Bottom-up hierarchical organization of a DRAM Chip

designed to be narrow in order to allow fine-grained reads and writes to a row-buffer. As we will explain below, the I/O bus is connected to the DRAM chip's data-pins that are even narrower (8b).

Bank (128 subarrays). Many subarrays (e.g., 128) are grouped into a larger entity called a *bank*, as shown in Figure 8 (middle). Each bank has its own *row-decoder* that activates a row by raising only its wordline among all wordlines within the entire bank. Therefore, if all rows of a DRAM chip belonged to the same bank, only one row can be accessed at a time. To allow parallelized accesses to rows more than one row, a modern DDR3 DRAM chip has eight banks [13]. In addition, each bank has an *I/O buffer* [10, 16, 23][4] (64b) that reads and writes from the bank's I/O bus that is connected to the row-buffers of all subarrays within the bank.

DRAM Chip (8 banks). As Figure 8 (right) shows, the I/O bus (64b) from all the eight banks are fed into the DRAM chip's I/O driver which directly connects to the 8 data-pins. To reduce chip-packaging and board-routing costs, a DRAM chip has only a very small number of data-pins. However, note that the I/O bus (64b) is 8x wider than the data-pins (8b). This is because the I/O bus operates at 8x lower frequency than the data-pins and, hence, the I/O bus must be 8x wider in order to sustain the maximum data throughput of the data-pins.[5]

Wiring between two rows. As we have just explained, given two rows within a DRAM chip, the two rows are connected differently depending on where they lie in the DRAM chip's hierarchy of banks and subarrays. There are three possible cases that we list below.

1. *Same subarray:* The two rows are directly connected to each other through the wide bitlines (4Kb), creating the opportunity for low-latency row-granularity data-transfer between them.

2. *Different subarrays (same bank):* The two rows are indirectly connected through the narrow I/O bus (64b).

3. *Different subarrays (different banks):* Similar to the second scenario, the two rows are indirectly connected through the narrow I/O bus (64b).

[4]Similar to a subarray's row-buffer (4Kb), a bank's I/O buffer (64b) has 64 sense-amplifiers [23] that read from the I/O bus (64b). Unlike a row-buffer, the I/O buffer writes to the I/O bus using its separate 64 write-drivers [14]. Due to this division of labor, the I/O buffer incurs a delay when switching between performing reads and writes – this delay will become relevant in §4.2.

[5]This is why in a modern DDR3 DRAM chip, the minimum number of consecutive transfers over the data-pins (i.e., "burst-length") is eight [13].

For the second and third cases, while row-granularity data-transfer between the rows is not possible, there is still opportunity for conserving energy by performing the data-transfer within the DRAM chip in self-contained manner.

4 Mechanism: Row-Copying

In this section, we discuss how to perform row-copy operations in DRAM between a source row and a destination row. As we have just explained, depending on the location of the source and destination rows, there are three variants of our mechanism: *(i) intra-subarray* copying between two rows in the same subarray, *(ii) inter-subarray* copying between two rows in different subarrays of the same bank, *(iii) inter-bank* copying between two rows in different subarrays of different banks. Our first mechanism, intra-subarray copying, provides the largest latency and energy benefits due to the wide bitlines (8Kb) and the wide row-buffer (8Kb) shared by rows in the same subarray. On the other hand, our second and third mechanisms provide only energy benefits since their transfers are serialized over the narrow I/O bus (64b). In the following, we discuss the three mechanisms in detail.

4.1 Same Subarray: Intra-Subarray Row-Copy

When the source and the destination rows are in the same subarray, they share the same row-buffer. Our *intra-subarray* row-copy mechanism consists of two steps: *(i)* loading the source row into the row-buffer and *(ii)* loading the row-buffer into the destination row. In contrast, a single-step mechanism, in which the source row writes *directly* into the destination row, is not possible for reasons we explain later in this section.

We can easily accomplish the first step by activating the source row. By doing so, we load the source row's data into the row-buffer which, in turn, drives the bitline voltages to the source row's data (either V_{DD} or 0). At this point, if we could somehow connect the bitlines to the destination row as well, the bitline voltage would be copied into the destination row, thereby accomplishing the second step. This is because the row-buffer is *already* driving the bitline voltages with the data of the source row (either V_{DD} or 0) – hence, connecting the bitlines to the destination row will result in the row-buffer *overwriting* the data that was originally in the destination row.[6]

Unfortunately, we cannot accomplish the second step in the manner we have described. This is because the source row and the destination row *cannot* both be connected to the bitlines at the same time, which requires both of their wordlines to be raised. Two raised wordline are not possible due to the bank's row-decoder that determines the unique row whose wordline can be raised among all rows within the bank (to which the subarray belongs). A naive solution to this problem is to precharge the source row (thereby lowering its wordline) so that the destination row's wordline can then be raised. However, precharging the source row not only lowers its wordline, but also clears

[6]Note that this is different from the conventional case where the bitline voltages are at $\frac{1}{2}V_{\text{DD}}$ (not V_{DD} or 0) before the bitlines are connected to a row, in which case the row "overwrites" the bitlines (❸, Figure 6), not the other way around.

the row-buffer such that the data it stores is lost (i.e., the bitline voltages are precharged to $\frac{1}{2}V_{\text{DD}}$) as explained in §3.2, Figure 7.

To solve this problem, we introduce a new DRAM command called *DEACTIVATE* that only lowers a wordline *without* clearing the row-buffer – in this sense, the DEACTIVATE command is a limited form of the PRECHARGE command. After issuing a DEACTIVATE to the source row, an ACTIVATE can be issued to the destination row, thereby accomplishing the second step of loading the row-buffer into the destination row.

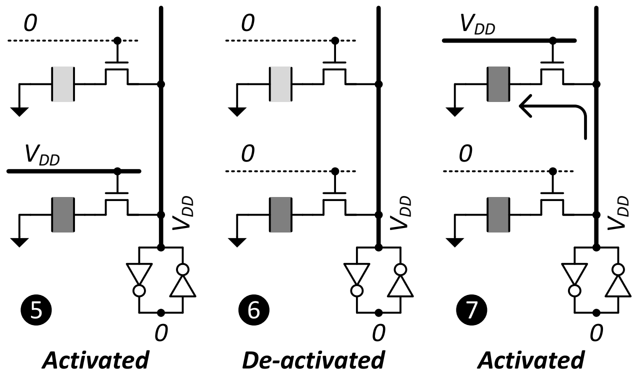


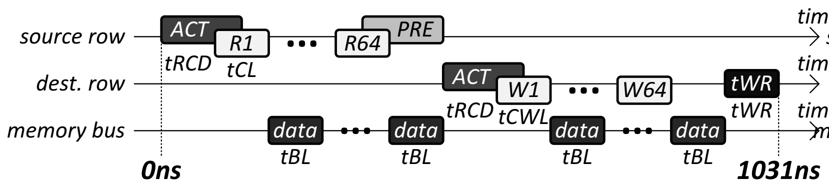
Figure 9: Row-copy: source (bottom), destination (top)

Figure 9 depicts our intra-subarray row-copy mechanism. Initially, we assume that the source row (bottom row) has already been activated (❺) in the manner of Figure 6. We then lower the wordline of the source row by issuing a DEACTIVATE command (❻). Note that the voltage of the bitline remains unchanged at V_{DD} during this process. Now we are free to issue an ACTIVATE command to the destination row (top row) to raise its wordline. As soon as this happens, the destination row is connected to the bitlines. Since the bitline voltage is already being driven to V_{DD} by the sense-amplifier, the bitline fully charges the destination row (❼), thereby copying the data into the destination row.[7]

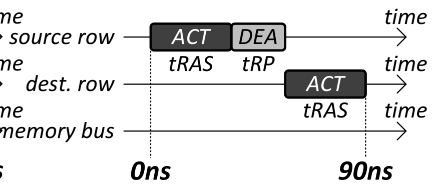
Now that we have described our two-step mechanism, we briefly explain why a single-step mechanism is not possible. The source row *cannot* write directly into the destination row due to two reasons. First, their wordlines cannot be both raised at the same time due to the row-decoder's restrictions. Second, even if both wordlines were raised somehow, data would not not be copied from one row to another. Instead, both rows would only lose their data irretrievably – if the rows store opposite charges, then the charges cancel each other when the rows are connected through the bitlines (§3.2, Figure 6).

DRAM Support for DEACTIVATE. Implementing the DEACTIVATE command requires minimal changes to the DRAM chip (0.016% die-size overhead, §6). As we explained, DEACTIVATE performs only a subset of the operations performed by PRECHARGE. While PRECHARGE performs two operations (lower wordline and drive the bitlines to $\frac{1}{2}V_{\text{DD}}$), DEACTIVATE performs just the first operation and leaves the bitline voltages unchanged. Therefore, DEACTIVATE can

[7]Between ❻ and ❼, the bitline voltage slightly drops to V_{DD}-δ since the bitline shares some of its charge with the uncharged cell, but it soon recovers back to V_{DD} since the sense-amplifier is driving it to V_{DD}.



(a) Baseline†



(b) Intra-subarray row-copy†

† DRAM commands: *ACT* (Activation), *PRE* (Precharging), *DEA* (Deactivation), *R* (Read), *W* (Write)

Figure 10: Intra-subarray row-copy: latency savings on a 4KB page copy (DDR3-1066 timing parameters [13])

be implemented by decoupling the two steps involved in PRECHARGE. Fortunately, in an existing DRAM chip, the two steps of PRECHARGE are already decoupled and controlled by two different control-signals within a DRAM chip. The first control-signal brings down the currently activated wordline by feeding the bank's row-decoder with an appropriate input. After some delay to ensure that the wordline is completely lowered, the second control-signal notifies the sense-amplifiers to drive the bitlines to $\frac{1}{2}V_{\text{DD}}$. In order to support DEACTIVATE, the DRAM chip's internal control-logic (that processes DRAM commands) should be slightly modified to trigger only the first control-signal when a DEACTIVATE is issued. This incurs a negligible DRAM die-size overhead of only 0.016%, as we show in §6.

Latency and Energy Benefits. Figure 10 shows detailed command-by-command timelines[8] of copying a 4KB page in the baseline (Figure 10a) and in our intra-subarray row-copy mechanism (Figure 10b). Calculated using DDR3 DRAM command timing parameters [13], the latency of the baseline is 1031ns, whereas the latency of our mechanism is 90ns – a 11.5x reduction. Additionally, our intra-subarray mechanism is self-contained within the DRAM chip and does not consume energy or bandwidth over the external memory bus. Specifically, for copying a 4KB page, our mechanism reduces the energy consumption by a very large factor, 74.4x, as we discuss in §6.

4.2 Different Subarrays: Inter-Subarray & Inter-Bank

When the source and destination rows are in different subarrays, they do not share the same bitlines/row-buffer. Therefore, we cannot utilize the wide bitlines/row-buffer to transfer data between the two rows – regardless of whether the two rows are in different subarrays in the same bank or different banks. Instead, for both cases, data must be transferred using the narrow I/O bus (64bit) that connects to all row-buffers within a bank as well as to all row-buffers in a different bank (Figure 11). In that regard, our second and third row-copy mechanisms (*inter-subarray* and *inter-bank*) are similar to each other. In the following, we first describe inter-bank row-copy, since it is easier to understand, then describe inter-subarray row-copy.

Inter-Bank Row-Copy. When the source and destination rows are in *different banks* altogether, their wordlines are controlled by separate row-decoders (one in each bank). Therefore, we

[8]Although DEACTIVATE is a subset of PRECHARGE, we conservatively assume that DEACTIVATE incurs the same latency as a PRECHARGE, i.e., *tRP* [13] (row-precharge latency).

can simultaneously activate both rows and load them into their respective row-buffers (i.e., source and destination row-buffers). Subsequently, we transfer data between the source and destination row-buffers by utilizing the I/O bus that connects them, as shown in Figure 11. This involves two operations – *(i)* reading 64bits from the source row-buffer into the I/O bus, *(ii)* writing 64bits from the I/O bus into the destination row-buffer – both of which must be performed repeatedly since the I/O bus (64bit) is narrower than the row-buffer (4Kbit). We observe that these two operations are identical to issuing a conventional READ or WRITE command to each of the two row-buffers, respectively – except for one major difference. Unlike READ/WRITE, these two operations do *not* transfer the data on the I/O bus to/from the DRAM chip's data-pins. Therefore, we introduce a new DRAM command called *TRANSFER* that reads a 64bit data from the source row-buffer and steers it over the I/O bus and writes it into the destination row-buffer (as shown in Figure 11), *without* transferring the data to the DRAM chip's data pins. In order to copy all of the data in the source row-buffer (4Kbit), the TRANSFER command must be issued by the memory controller to the DRAM chip 64 times (= 4Kbit/64bit).

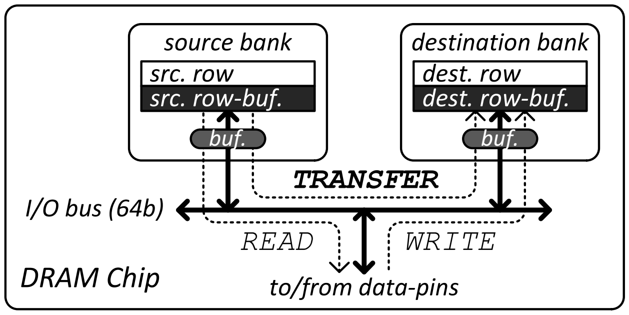


Figure 11: Inter-bank row-copy

Inter-Subarray Row-Copy. When the source and destination rows are in different subarrays within the *same bank*, it would seem natural for the the two rows to transfer data directly between them without the data ever having to leave the bank – however, our inter-subarray row-copy mechanism does *not* operate in such a manner. Instead, our mechanism utilizes the TRANSFER commands to copy the source row *entirely out of* the original bank into a *different bank*, after which the row is then copied into the destination row in the original bank. In this regard, the inter-subarray mechanism is an extension of the inter-bank mechanism that we have already described. The reasoning behind our inter-subarray mechanism has to do with the bank's I/O buffer (§3.3) that performs reads and writes on the bank's I/O bus. If we transfer data directly from the source row-buffer into the destination row-buffer, then the bank's I/O buffer has to switch repeatedly between reading 64bits (from the source row-buffer) and writing 64bits (into the destination row-buffer). Unfortunately, there is a latency penalty associated with the I/O buffer for such switches between reads and writes. This in contrast to an inter-bank row-copy, where the source bank's I/O buffer performs only reads while the destination bank's I/O buffer performs only writes. This is why our inter-subarray mechanism first copies the source row into a "temporary" row in a different bank. In order to implement our mechanism, it is sufficient to reserve one row in each of the eight banks to serve as the "temporary" row for another bank. Given the hundreds of thousands of rows within a DRAM chip, the capacity loss of the eight "temporary" rows is negligible (0.0015%).

DRAM Support for TRANSFER. Implementing the TRANSFER command requires additional control-logic to disconnect the I/O bus from the data-pins and to steer the data on the I/O bus appropriately between different banks. This incurs a negligible 0.01% die-size increase (§6).

Latency and Energy Benefits. When copying two rows in different banks, the baseline takes 1016ns (summed latency of requisite DRAM commands) whereas our inter-bank row-copy mechanism takes only 525ns – a 1.9x reduction. In addition, our inter-bank mechanism reduces the energy consumption by a factor of 3.2x (§6). When copying two rows in different subarrays in the same bank, however, our inter-subarray mechanism takes slightly longer (1035ns) than the baseline (1031ns), due to the additional latency for transferring data from the source row to the "temporary" row and from the "temporary" row to the destination row. However, our inter-subarray mechanism provides energy savings – reducing the consumption by a factor of 1.5x (§6).

5 Mechanism: Row-Initialization

Bulk initialization sets a block of memory to a static value, which can be either zero or any non-zero arbitrary value. We distinguish these two cases and develop two separate mechanisms for row-initialization, respectively.

Initialization to Zero. Since memory is often initialized to zero, we treat it as a special case that we refer to as a "clear." Our mechanism has two techniques of clearing a target row: *(i) cleared row* – our technique of choice for the remainder of the paper – and *(ii) cleared row-buffer* – discussed only briefly in this section. In the first technique, we dedicate a single row in each subarray that always stores zeros – i.e., a *cleared row*. When clearing a particular row, the cleared row within the same subarray can simply be copied into the target row using our intra-subarray row-copy mechanism. However, the memory controller must ensure that the cleared rows are never modified by not exposing them to the processor. As a result of this, there is a slight loss in the memory capacity. Depending on the number of rows in a subarray (512–1024 [36, 19]), dedicating one row as a cleared row reduces the capacity by 0.2% (=1/512) or 0.1% (=1/1024). Also, the memory controller must perform a one-time initialization of the cleared rows during system bootup, by writing all zeros into them. In the second technique, *cleared row-buffer*, the row-buffer drives the bitlines to zero voltages using an internal voltage-source of 0 (i.e., ground). Existing row-buffers already have an internal $\frac{1}{2}V_{\text{DD}}$ voltage-source so that they can precharge the bitlines precisely to $\frac{1}{2}V_{\text{DD}}$. By adding a single transistor at next to the row-buffer, we allow the voltage-source to selectively switch between $\frac{1}{2}V_{\text{DD}}$ and 0. The die-size overhead of such a transistor at each subarray is a modest 0.16% (calculated using methods explained in §6).

Initialization to Non-Zero. For initialization to non-zero values, the processor must first initialize one row using the baseline mechanism of repeatedly writing zeros, one cache-line at a time. Once one row has been initialized, this row can be copied to many other rows (using our row-copy mechanisms) to initialize a large number of rows.

Latency and Energy Benefits. When initializing a row to zero, the baseline takes 521ns (summed latency of requisite DRAM commands) whereas our cleared row mechanism takes 90ns

– a 5.8x reduction. The mechanism also reduces energy consumption by a very large factor of 41.5x (§6).

6 Hardware Considerations & Analysis

Exposing Subarrays to Memory Controller. All our mechanisms require the memory controller to distinguish whether any two given rows belong to the same subarray or not. In order to expose the DRAM chip's subarray organization to the memory controller, we utilize a small 256Byte EEPROM that exists on every DRAM module (i.e., an assembly of multiple DRAM chips). This EEPROM, called the *serial presence detect* (SPD) [12], stores various information about the DRAM chips that is read by the memory controller at system bootup. DRAM manufacturers can utilize the SPD in order to convey how different rows (and their addresses) are internally mapped to subarrays within a DRAM chip.

DRAM Die-Size Analysis. We calculate the DRAM die-size overhead of RowClone using a publicly available 2Gb 55nm DDR3 DRAM technology model [29] and find that the area overhead is only 0.026% (0.016% for DEACTIVATE and 0.01% for TRANSFER). As § 4 explains, to implement DEACTIVATE, we modify the existing control-logic for PRECHARGE such that it can selectively mask the control-signal that triggers the row-buffer to drive the bitlines to $\frac{1}{2}V_{\text{DD}}$. To implement TRANSFER, we need additional muxing-/control-logic to steer the I/O bus data to the destination bank and to selectively disconnect the I/O bus from the data-pins. None of these changes are intrusive to the DRAM chip's dense core-array.

DRAM Energy Analysis. We calculate the main memory energy savings of our mechanisms using DRAM energy/power models from Rambus [29] and Micron [21]. We find that all of our mechanisms significantly reduce energy consumption due to eliminating unnecessary and costly transfers over the memory bus. For a 4KB copy, our mechanisms conserve energy by factors of 74.4x (intra-subarray), 1.5x (inter-subarray), and 3.2x (inter-bank) compared to the baseline. For a 4KB clear, our mechanism conserves energy by a factor of 41.5x (cleared row). Our energy savings calculations are only for the main memory subsystem and they likely provide conservative estimates since they do not consider additional energy savings at the processor, cache hierarchy and the on-chip interconnect.

7 Exposing Page Copy/Clear to Software

Exposing fast and efficient page copies/clears to software requires two basic modifications to the hardware-software interface. First, the hardware should *recognize copies and clears* performed by the software, and make use of DRAM row-granularity operations to accelerate these operations when possible. In order to communicate the copy and clear operations to hardware, we add `MemCopy` and `MemClear` instructions to the ISA (which handle any granularity of copy or clear, and use DRAM row-granularity operations for the aligned portions). Some ISAs (e.g., x86) already have such instructions (e.g., `REP MOVSB` and `REP STOSB`); in these ISAs, one could modify the implementation of these existing instructions to use RowClone copying and initialization mechanisms. Second, the software should provide as much *opportunity* as possible by performing copies and clears with alignments and sizes that are multiples of the DRAM row-size. We expose the *minimum accelerated block size* (MABS) as a hardware property that the software can read at runtime (e.g., as a special processor register) in order to adapt its memory layout and copy/clear operations where possible.

7.1 Instruction Set Architecture (ISA) Modifications

`MemCopy` & `MemClear` Instructions. A data copy or clear is an *architectural* operation because it modifies main memory, which is software-visible state. However, our mechanisms are fundamentally *microarchitectural*: accelerated copies and clears can only be performed in multiples of the DRAM row-size. This row-size can vary with the memory technology, specific memory chip, or individual system design. In order to avoid exposing this microarchitectural detail in an architectural interface, we introduce general-purpose `MemCopy` and `MemClear` instructions, defined in Table 1. These instructions either copy (`MemCopy`) or set to zero (`MemClear`) an arbitrary range of virtual addresses, specified at the *byte* granularity.

Instruction	Parameters	Description
`MemCopy`	*dest, source, size*	Copy *size* bytes of data from *source* to *dest*
`MemClear`	*dest, size*	Set *size* bytes to zero at *dest*

Table 1: New instructions: in-DRAM copy/clear

The hardware can use the accelerated implementation when any portion of the source or destination range is appropriately aligned (occupies an entire DRAM row), and otherwise, fall back to a conventional copy or clear. If an operation starts or ends at an address which is unaligned with respect to DRAM row boundaries, an accelerated row-buffer-based operation may still be possible in the *middle* of the source or destination range, if at least one whole DRAM row is spanned by that range (and for copies, if the starting row offset is the same for the source and destination). The `MemCopy` and `MemClear` operations use *virtual* addresses, and fully obey memory protection. The hardware performs address translation for each virtual page in the source and destination ranges, and sends the resulting physical addresses to DRAM in the copy or clear commands.

Minimum Accelerated Block Size. Copy and clear operations can only make use of whole-row accelerations if the source and destination ranges consist of *whole DRAM rows*. In order to allow software to align its copy and clear operations whenever possible, RowClone exposes the *minimum accelerated block size* (MABS) as a value that software can read from the hardware. MABS is the smallest-granularity chunk of physical memory which is fully aligned to DRAM row boundaries.

MABS is influenced by two factors. First, MABS depends on the number of DRAM chips connected in parallel to a memory bus – i.e., the number of DRAM chips in a *rank* (§2.3). Since

DRAM chips within the same rank operate in lockstep, their *effective row-size* is the product of a single chip's row-size and the number of chips in a rank – for a 64 bit memory bus and 8 bit DRAM chips, a rank consists of 8 chips (=64/8). Second, MABS depends on the memory controller's address-mapping policy which converts physical addresses into DRAM addresses (channel[9], rank, bank, row). In the simplest case, when the physical address is *row-interleaved*, contiguous physical addresses span an entire row before traversing to the next row. In this case, MABS is simply the effective row-size of a rank. However, other interleaving schemes can increase the MABS by interleaving contiguous addresses across multiple rows before spanning any single row completely. For example, in a *cache-line-interleaved* system, adjacent cache-lines (e.g., 64 bytes) in the physical address space typically alternate between channels. In this case, MABS is the product of a single rank's effective row-size and the number of memory channels in the system.

In the system which we evaluate, which is row-interleaved, the minimum accelerated block size is 4 KB. We explicitly design the system so that this size is equal to the architectural page size so that the operating system kernel has maximum flexibility to make use of accelerated copies and clears. Below, we discuss operating system techniques which could be used to allocate and manage memory at a coarser granularity and enable efficient use of a larger MABS.

7.2 Software Modifications

In this section, we describe how user and system software can be modified to take advantage of RowClone-accelerated data copy and clear operations. No modifications to system software are necessary for user programs to benefit from the new instructions. However, additional performance/energy improvements can be achieved if the system software (e.g., operating system) is also modified, as we describe below.

User Programs. User programs can use the `MemCopy` and `MemClear` instructions to directly implement `memcpy()` and `memset()` (when the initialization value for memory is zero), respectively. If such a change is made in the system C library, for example, then all programs which perform bulk data copying or initialization will immediately benefit.

Kernel Page Operations. As we demonstrated in §2, kernel page zeroing and copying (for demand-allocated user data pages, and copy-on-write pages, respectively), can consume a significant fraction of the system's available memory bandwidth. Instead, the kernel can use `MemCopy` and `MemClear` to implement these operations.

DRAM-Aware Page Allocation. Finally, note that system software can determine the destination page of a copy/clear by controlling which new page is allocated for this purpose. As we described in §4, data copies are more efficient (incur lower latency, bandwidth, energy) when the data is copied within the same DRAM subarray (§4, intra-subarray row-copy). To enable intra-subarray copies, the operating system can keep track of as-of-yet unallocated (i.e., free) physical frames in a *subarray-aware manner* – i.e., the operating system can inspect certain bits from a free physical frame's address to determine which subarray it belongs to (§6). Subsequently, whenever a new page is required for a copies/clears, the operating system can allocate a physical frame for the destination page that is in the same subarray as the physical frame for the source page. If

[9]We use the terms "memory bus" and "memory channel" interchangeably.

MABS happens to be larger than the architectural page-size, in order to provide more opportunity for DRAM-assisted copies/clears, the kernel can allocate contiguous physical frames (that map to the same DRAM row) to contiguous pages. While the evaluation of such DRAM-aware page allocation policies are out of the scope of this paper, we believe such policies are promising to examine in future research.

8 Microarchitectural Implications

In this section, we will briefly outline the hardware changes which are necessary *in the core and memory hierarchy* in order to support RowClone. These minimal changes are in addition to the support for the new DRAM commands which we have already described.

Memory Copy/Clear Instructions. As we introduced in §7, the processor must support two new instructions, `MemCopy` and `MemClear`. If the processor's ISA already includes memory copy and clear instructions (e.g., x86), then no new instructions are required and these existing instructions can be used. At the microarchitectural level, these instructions will likely be implemented in microcode in order to implement the cache operations and send the appropriate DRAM commands.

Interactions with Cache Coherence. Because our DRAM operations move user data directly in memory, without interacting with any caches in the system, special care must be taken to ensure that cache coherence is maintained. Our key idea is to *(i)* ensure that any cached source data is written to DRAM before an in-DRAM copy, and *(ii)* ensure that any cached destination data is invalidated so that DRAM has the authoritative data after an in-DRAM copy/clear. In order to implement these operations and ensure coherence, we rely on the fact that most existing systems already provide *back-invalidations* and *back-snoops* which memory agents in the system (such as IO devices or DMA engines) can use to peek into the processors' coherent caches [3]. At a high level, in-DRAM copy/clear can be made correct if the memory controller performs the following three operations in that order: *(i)* back-snoops for source data, *(ii)* issues DRAM copy/clear commands, and *(iii)* back-invalidates for destinations – where *(ii)* and *(iii)* can potentially be done in parallel. A more detailed design depends on the specifics of the cache coherence protocol and the memory hierarchy design and is beyond the scope of this work.

9 Related Work

To our knowledge, RowClone is the first proposal that enables in-DRAM bulk data copy *and* initialization, and explores the full stack (DRAM design, cache coherence, microarchitecture, ISA, and software) implications of this new mechanism. Several past works have either proposed row-to-row copies [2, 6, 15, 22, 25, 26, 27], or proposed optimizing data initialization with specialized caches [18] or software modifications [39]. However, no previous work addresses both copying/initialization using general-purpose hardware-mechanisms and software-interfaces.

DRAM Copy. At a high-level, several patents [2, 6, 25, 26, 27] observe that a row-buffer is shared by multiple rows and propose the abstract notion that the row-buffer can be used to copy rows, but do not provide specific details. While this is possible *only* when the two rows are in the same subarray (§ 4.1, intra-subarray row-copy), the patents make no such distinction. In contrast, our proposal is more general: RowClone is capable of inter-subarray and inter-bank row-copy, and also supports row-initialization. In addition, the patents discuss only the DRAM chip changes, and do not address the software (user and operating system), architectural, or cache coherence implications of row-copy. Nor do these patents provide evaluations.

SRAM Copy. Other prior works [15, 22] propose efficient copy mechanisms in SRAM. Miyoshi et al. [22] propose a write-back cache design that implements a word-copy operation (in an SRAM cache) by adding extra sense-amplifiers that incur a large die-size overhead. In contrast, RowClone has little hardware overhead, since it only slightly modifies the DRAM chip's existing control-logic. Kadota et al. [15] propose a content-addressable memory that can copy words that have the same word address. However, this mechanism requires complex addressing logic to activate multiple rows simultaneously. In contrast, RowClone does not need complex logic since it copies data by simply activating only one row at a time.

Memory Initialization. Prior works [18, 39] have proposed different ways of minimizing the memory initialization latency. Lewis et al. [18] investigate the problem of initialization misses to newly allocated memory, in which invalid (i.e., uninitialized) data is unnecessarily loaded from memory. The paper's proposal avoids these unnecessary data-transfers by allocating and initializing a block into the cache when an uninitialized store miss is identified. RowClone is orthogonal and can likely be combined with it for further gains. Yang et al. [39] observe that the cost of zero initialization is significant even when several software optimizations are applied. Their work proposes two modifications to existing software bulk-zeroing implementations: *(i)* the use of non-temporal store instructions, which bypass the cache and thereby avoids cache pollution, and *(ii)* using a separate software thread to zero memory ahead of time. While such optimizations can sometimes potentially hide software zeroing latency, they do not reduce the DRAM bandwidth or energy that fundamentally must be spent when the processor is responsible for zeroing memory. RowClone, in contrast, avoids this bandwidth and energy overhead.

10 Methodology

In this study, we evaluate RowClone on both single-core systems and with multi-programmed workloads on multicore systems, consisting of two categories of applications: copying/initialization-intensive applications, and SPEC CPU2006.

Workloads. We evaluate RowClone with multiprogrammed workloads which consist of two categories of applications: copy/initialization benchmarks and SPEC CPU2006 [33] applications. Each workload has half of its cores occupied by each type of application (e.g., 8-core workloads have four copy/initialization applications and four SPEC CPU2006 applications). Our primary multicore results derive from 240 workloads (80 each on 2-, 4-, and 8-core configurations). We classify SPEC CPU2006 applications into "memory intensive" and "memory non-intensive" categories based on last-level cache miss rate: applications that experience more than 5 misses per thousand instructions (MPKI) are classified as memory intensive. We classify the copy/initialization benchmarks as "copy intensive" or "copy non-intensive" based on whether they contain a non-negligible number of page copies and initializations. We then form workloads in four categories: Copy-High/Mem-High (CHMH), Copy-High/Mem-Low (CHML), Copy-Low/Mem-High (CLMH), and Copy-Low/Mem-Low (CLML).

Copying/initialization benchmarks. Our copying and initialization benchmarks are: **apache2** (web server); **bootup** (system bootup); **compile** (gcc, make); **filecopy** (`cp -R`); **forkset** (the fork microbenchmark described in §2); **memcached** (an in-memory key-value store); **mysql** (a conventional on-disk database); and **shell** (a Unix shell script). We chose these eight benchmarks to represent behaviors such as process creation and memory initialization.

Collecting instruction traces. Instruction traces for our copying/initialization benchmarks are collected with a modified version of Bochs [1], a full-system x86-64 functional emulator, running a GNU/Linux system with kernel 3.6.6. We modify the kernel's implementation of page copies and clears to use the `MemCopy` and `MemClear` instructions and mark these instructions in our traces. We collect 1-billion instruction traces of representative portions of each workload. Our SPEC CPU2006 instruction traces are derived from a Pin-based frontend (uses PinPoints [28] to sample representative phases of execution).

Simulation. Instruction traces from both categories of benchmarks (copying/initialization and SPEC CPU2006) are simulated on a cycle-level multicore simulator (which models the cores, caches, and memory system) with a full command-level model of DDR3 DRAM. The simulator models RowClone at the DRAM command level, and models its cache effects (e.g., invalidations). Each workload runs until the last application has retired 100 million instructions (2-, 4-, 8-core workloads) or 1 billion instructions (1-core workloads). Applications continue to exert pressure on the memory system after completion. We report instruction throughput (IPC) for single-core workloads and weighted speedup [32] for multicore configurations. System parameters are given in Table 2.

RowClone. We model all page copies as *intra-subarray* row-copies (§4). The system can ensure this because, in this evaluation, *(i)* we perform copies only when a new page is allocated (for copy-on-write), and *(ii)* the kernel can allocate a destination page such that it belongs to the same subarray (§7.2). We model all page clears using the *cleared row* mechanism (§5).

11 Evaluation

RowClone improves system performance in two major ways. Accelerated copy and initialization operations *(i)* speed up the applications that use these operations, and *(ii)* reduce the bandwidth utilization of these applications, thereby allowing other applications to make faster progress. We evaluate performance, bandwidth and energy usage in multicore systems, then also show single-core performance for copy-intensive applications.

Multi-Core Performance. Fig. 12a shows performance (weighted speedup) geometric means for the four workload categories on an 8-core system, normalized to a system without RowClone. We draw two conclusions. First, in workloads which include copy/initialization-intensive applications, performance improves significantly both because RowClone reduces the latency of the copy and initialization operations, and because it reduces the bandwidth of these operations and hence their performance impact on the other applications running on the system. In the 8-core

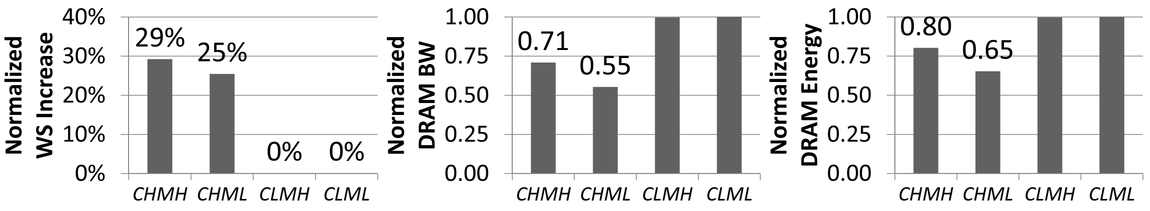
Component	Parameters
Processor	1–8 cores, 5.3GHz, 3-wide issue, 8 MSHRs, 128-entry instruction window
Last-level cache	64B cache-line, 16-way associative, 512kB private cache-slice per core
Memory controller	64/64-entry read/write queues per controller, FR-FCFS scheduler
Memory system	DDR3-1066 (8–8–8) [13], 2 channels, 2 ranks-per-channel, 8 banks-per-rank

Table 2: Configuration of simulated system

system, RowClone improves performance by 29% for workloads with copy-intensive applications and memory-intensive applications (i.e., CHMH category). Even when the applications which do not perform copies and initializations are *not* memory-intensive (CHML category), RowClone improves performance by 25% (largely because of RowClone's copy latency reduction). Second, as expected, RowClone does not negatively impact performance when copy/initialization operations are not present, because it does not alter any other aspect of the memory system.

Bandwidth and Energy Reduction. Figs. 12c and 12b show DRAM bandwidth and energy, respectively, for all workloads on the 8-core system. We draw two conclusions. First, RowClone reduces memory bandwidth (by eliding the data transfers that would have been required to copy or initialize data). This effect is significant: bandwidth is reduced by 45% in the best category (CHML). In the CHMH category, bandwidth is reduced by 29% (this is less than in the CHML category because the non-copy/initialization benchmarks consume significant bandwidth as well, hence copies and initializations are a smaller fraction of total bandwidth; the absolute reduction is still large). Second, as a consequence of reducing memory bandwidth, RowClone also reduces DRAM energy. Although some energy is expended by the copy or initialization operation inside the DRAM chips, this energy is significantly lower than the energy required to read, transfer, and write the data (as we showed in §6). Overall, energy is reduced by 35% in the best category (CHML).

Single-Core Performance: Copy-Intensive. Fig. 13 shows the instruction throughput (measured as IPC, or instructions per cycle) using RowClone for the eight copy/initialization-friendly benchmarks which we evaluate, normalized to IPC in the baseline system. These benchmarks are evaluated running alone on a single-core configuration. We draw two conclusions. First, for some benchmarks, the reduced latency of in-DRAM copy and initialization significantly improves performance (by up to 104% for the **forkset** microbenchmark, 40% for the **shell** workload, and 14% for **bootup**). However, for other workloads, such as **mysql** and **memcached**, performance is not improved, and can actually degrade slightly (by about 5% in both of these cases). This degradation



(a) Performance improvement (b) Bandwidth reduction (c) Energy reduction

Figure 12: 8-core system: performance, energy, bandwidth (averaged results for 32 workloads in each category)

occurs because of the *cache effects* of in-DRAM copy and initialization: the in-DRAM operations must invalidate the copied or initialized blocks from the cache hierarchy, but if these blocks are used soon afterward by the application, they must be brought back to the processor. These applications achieve higher performance if the copy or initialization is done in the conventional way (at the processor), because the cache contains the data when the operation is complete. Nevertheless, the performance impact is not large (because the in-DRAM operation is fast). If this performance impact becomes a significant problem, the hardware could track cache reuse of copied or initialized data following the in-DRAM operation, and disable RowClone dynamically if this reuse is high. Alternatively, software could provide hints by using alternate versions of `memset()` and `memcpy()` if this reuse behavior is known statically.

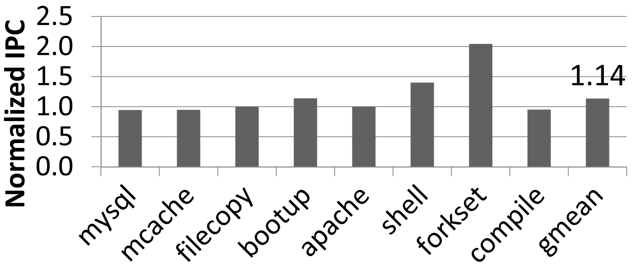


Figure 13: IPC increase: copy/init.-intensive benchmarks

Scalability with Core-Count. Fig. 14 shows performance for the two categories which include copy/initialization-intensive applications across a varying number of cores (2-, 4-, and 8-core systems). As this figure shows, RowClone's benefit increases as the number of cores increases. This is because RowClone alleviates the higher pressure exerted on memory bandwidth by eliminating the unnecessary bandwidth consumption due to copies and initializations – thereby enabling all applications to make faster progress.

12 Conclusion

We introduced RowClone, a new technique for bulk copying and initializing of data in DRAM. By making use of the large internal bandwidth of a DRAM chip, RowClone is able to copy one row (e.g., a 4 KB page) to another, or zero a row's data, with a simple sequence of commands that do

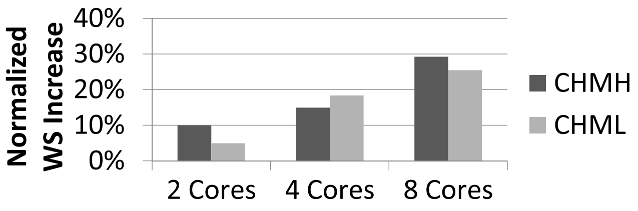


Figure 14: Weighted speedup increase for 2, 4, 8 cores

not read or write the row's data over the memory bus. We show that these copying and initialization mechanisms accelerate a one-page copy by 11.5x, and a one-page zero by 5.8x, with 74.4x and 41.5x energy reductions, respectively. We show that RowClone improves system application performance by 27% on average for a set of 64 data-copy-intensive workloads on an 8-core system while also reducing memory bus bandwidth and DRAM energy. These benefits come at the cost of only 0.026% DRAM area overhead. We conclude that accelerating bulk copying and initialization through the use of in-memory operations provides significant performance benefits and reduces bandwidth and energy, yielding a more efficient main memory system.

References

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

[2] Jin-hong Ahn. Memory device having page copy mode. U.S. patent number 5886944, 1999.

[3] T Berg. Maintaining I/O data coherence in embedded multicore systems. *IEEE Micro*, May–June 2009.

[4] J Chow et al. Shredding Your Garbage: Reducing Data Lifetime through Secure Deallocation. In *USENIX Security Symposium*, 2005.

[5] AM Dunn et al. Eternal Sunshine of the Spotless Machine: Protecting Privacy with Ephemeral Channels. In *OSDI*, 2012.

[6] Peter B. Gillingham et al. Dram page copy method. U.S. patent number 5625601, 1997.

[7] J Alex Halderman et al. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *USENIX Security*, 2008.

[8] Keith Harrison and Shouhuai Xu. Protecting Cryptographic Keys from Memory Disclosure Attacks. In *DSN*, 2007.

[9] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual (Volume 3A), August 2012.

[10] Kiyoo Itoh. *VLSI Memory Chip Design*. Springer, 2001.

[11] TB Jablin et al. Automatic cpu-gpu communication management and optimization. In *PLDI*, 2011.

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

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

[14] BongHwa Jeong et al. A 1.35V 4.3GB/s 1Gb LPDDR2 DRAM with Controllable Repeater and On-the-Fly Power-Cut Scheme for Low-Power and High-Speed Mobile Application. In *ISSCC*, 2009.

[15] H Kadota et al. An 8-kbit content-addressable and reentrant memory. In *JSSC*, 1985.

[16] R. Kho et al. 75nm 7Gb/s/pin 1Gb GDDR5 graphics memory device with bandwidth-improvement techniques. In *ISSCC*, 2009.

[17] Yoongu Kim et al. A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM. In *ISCA*, 2012.

[18] JA Lewis et al. Avoiding Initialization Misses to the Heap. In *ISCA*, 2002.

[19] KN Lim et al. A 1.2V 23nm $6F^2$ 4Gb DDR3 SDRAM with Local-Bitline Sense Amplifier, Hybrid LIO Sense Amplifier and Dummy-Less Array Architecture. In *ISSCC*, 2012.

[20] Mel Gorman. Huge Pages. `http://lwn.net/Articles/374424`, 2012.

[21] Micron. DDR3 SDRAM System-Power Calculator, 2010.

[22] A Miyoshi et al. A write-back cache memory using bit-line steal technique. In *VLSI*, 1998.

[23] Yongsam Moon et al. 1.2V 1.6Gb/s 56nm $6F^2$ 4Gb DDR3 SDRAM with hybrid-I/O sense amplifier and segmented sub-array architecture. In *ISSCC*, 2009.

[24] S. Morein. ATI Radeon HyperZ Technology. In *Graphics Hardware*, 2000.

[25] Donald M. Morgan et al. Drams having on-chip row copy circuits for use in testing and video imaging and method for operating same. U.S. patent number 5440517, 1995.

[26] Donald M. Morgan et al. Hardware implemented row copy enable mode for drams to create repetitive backgrounds for video images or dram testing. U.S. patent number 5381368, 1995.

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

[28] Harish Patil et al. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation*MICRO-37*, 2004.

[29] Rambus. DRAM Power Model, 2010.

[30] A Silberschatz et al. *Operating System Concepts, 7th Ed.* Wiley, 2005.

[31] M Sindelar et al. Sharing-aware algorithms for virtual machine colocation. In *SPAA*, 2011.

[32] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. *ASPLOS-9*, 2000.

[33] Standard Performance Evaluation Corporation. SPEC CPU2006.

[34] Jakub Szefer and Ruby B Lee. Architectural support for hypervisor-secure virtualization. In *ASPLOS*, 2012.

[35] The PaX Team. `http://pax.grsecurity.net`.

[36] Thomas Vogelsang. Understanding the Energy Consumption of Dynamic Random Access Memories. In *MICRO*, 2010.

[37] M Vrable et al. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *SOSP*, 2005.

[38] Carl A. Waldspurger. Memory resource management in vmware esx server. In *OSDI*, 2002.

[39] X Yang et al. Why Nothing Matters: the Impact of Zeroing. In *OOPSLA*, 2011.