

# 3D-Stacked Memory Architectures for Multi-Core Processors

Gabriel H. Loh

Georgia Institute of Technology  
College of Computing  
loh@cc.gatech.edu

## Abstract

*Three-dimensional integration enables stacking memory directly on top of a microprocessor, thereby significantly reducing wire delay between the two. Previous studies have examined the performance benefits of such an approach, but all of these works only consider commodity 2D DRAM organizations. In this work, we explore more aggressive 3D DRAM organizations that make better use of the additional die-to-die bandwidth provided by 3D stacking, as well as the additional transistor count. Our simulation results show that with a few simple changes to the 3D-DRAM organization, we can achieve a  $1.75\times$  speedup over previously proposed 3D-DRAM approaches on our memory-intensive multi-programmed workloads on a quad-core processor. The significant increase in memory system performance makes the L2 miss handling architecture (MHA) a new bottleneck, which we address by combining a novel data structure called the Vector Bloom Filter with dynamic MSHR capacity tuning. Our scalable L2 MHA yields an additional 17.8% performance improvement over our 3D-stacked memory architecture.*

## 1. Introduction

It is well known that DRAM access latencies have not decreased at the same rate as microprocessor cycle times. This leads to the situation where the relative memory access time (in CPU cycles) keeps increasing from one generation to the next. This problem is popularly referred to as the *Memory Wall* [41]. Three-dimensional (3D) die-stacking has received a great deal of recent attention in the computer architecture community [5, 20, 26, 27, 29, 32]. 3D stacking enables the construction of circuits using multiple layers of active silicon bonded with low-latency, high-bandwidth and very dense vertical interconnects [8, 14]. 3D stacking also enables mixing dissimilar process technologies such as high-speed CMOS with high-density DRAM. Stacking DRAM directly on top of a processor is a natural way to attack the Memory Wall problem.

A few prior studies have already started to investigate the potential of 3D-stacked memory. Liu et al. and Loi et al. independently researched the performance impact of placing a system's main memory on top of the processor using 3D fabrication technologies [24, 26]. While these studies report impressive performance speedups for 3D-stacked memories (92% performance gain for Liu et al.'s work, and 65% for Loi et al.'s), we will show that their results are really

only a starting point. In particular, these studies have only considered commodity DRAMs with traditional main memory architectures. In this paper, we investigate the performance benefit of 3D-stacked architectures that employ more aggressive, highly-parallel memory organizations not constrained by traditional memory topologies.

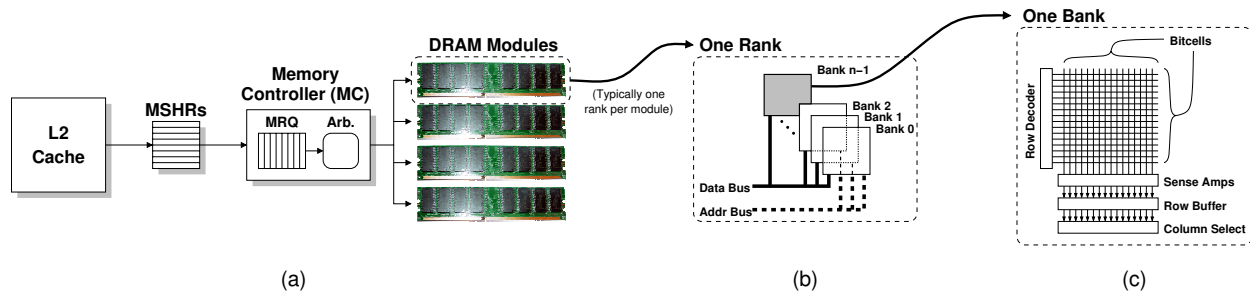
We detail our proposed 3D memory architecture in the remainder of this paper as follows. Section 2 reviews background material on 3D integration, DRAM, and 3D-stacked DRAMs. Section 3 replicates and validates the performance benefits of previously proposed 3D DRAM architectures and presents results for "true" 3D DRAMs (i.e., what if you split the actual DRAM bitcell arrays, wordlines and bitlines across multiple layers in 3D?). Section 4 presents our new 3D-DRAM organizations that better exploit the possibilities enabled by 3D technologies. While the proposed 3D-DRAM architectures provide a significant performance boost, we discover that the L2 MSHR architecture also requires scaling. Section 5 details a novel MSHR organization based on a new data structure called the Vector Bloom Filter which we also combine with dynamic MSHR resizing, and Section 6 draws some final conclusions.

## 2. Background

This section provides a brief overview of DRAM technology and 3D integration. This section also reviews previous work on 3D-stacked DRAMs and then details our simulation methodology.

### 2.1. DRAM Architectures

A DRAM is simply an array of single-transistor bitcells with some accompanying logic to access those bits. There are, however, many ways to organize those bits and logic blocks. Figure 1(a) illustrates a generic memory system organization. Starting with the L2 cache, a miss here requires an access to main memory to satisfy the memory request. A miss status handling register (MSHR) must be allocated which stores bookkeeping information regarding the missed request [22]. The L2 control logic forwards the request to the memory controller (MC) which is responsible for interfacing with the actual DRAM chips. The request may be buffered in a Memory Request Queue (MRQ), and the actual order of requests sent to main memory will be determined by an arbiter or scheduler. The MC forwards the physical address to be read or written as well as manages the timing of other signals corresponding to the different phases of



**Figure 1. (a) Organization of the memory hierarchy starting from the L2 cache. (b) Details of one memory rank. (c) Details of one memory bank.**

the memory access protocol. Different MC implementations may have different levels of complexity; some may use simple first-in-first-out (FIFO) processing of memory requests while others may reorder requests to improve access locality [18, 34].

At the highest level, the DRAM arrays are divided into *ranks*, typically with one or two ranks per DRAM module, as shown in Figure 1(b). Within each rank, the memory is subdivided into *banks*. Each bank consists of a 2D array of bitcells (c). On a DRAM read, bits from the physical address select the rank, bank and row; a set of sense amplifiers read the contents of that row and then latch the results in a row buffer. Any subsequent accesses to the same row can bypass the array-read and access the data directly from the row buffer. A few remaining bits from the physical address select the column to be read. The data are sent back to the MC, which then sends the data to the processor. It is also important to note that reading the contents of a DRAM row destroys those contents. After *any* read, the row must eventually be written back to the bitcell array. Furthermore, the contents of the bitcells slowly leak or decay, and so they must be periodically read and written back (“refreshed”). The timing and interfaces of different DRAM standards (e.g., SDRAM, DDR, Rambus, etc.) vary, but the high level organization is similar across families. For more information on the organization of modern DRAM architectures, we refer the reader to the overview provided in Cuppa et al.’s study of DRAM performance [7].

In general, increasing the number of ranks or banks increases the amount of parallelism that the memory system can support in terms of the number of simultaneously open pages/rows. Increasing the number of row buffer entries per bank (sometimes called a row buffer cache or a DRAM cache [17]) also helps improve performance by increasing the frequency of lower-latency row-buffer hits. Increasing the width or clock speed of buses increases the bandwidth for transferring data between components. In typical systems, such as that illustrated in Figure 1, these parameters usually take on relatively small values. Having too many ranks is undesirable because it increases the total number of DRAM modules (chip count), the number of memory sock-

ets, and the length and loading of the memory bus. The number of banks is limited by the area required to implement the additional sets of row decoders, sense amplifiers, column muxes, and row buffers. Increasing the bus widths between the MC and the DRAM modules is primarily limited by the pin counts on both ends, as well as the area requirements on the motherboard for the printed circuit board (PCB) traces. Similarly, the width of the front-side bus (FSB) between the processor and an off-chip MC cannot be easily increased due to the pin limitations of the processor and MC packages. Increasing clocks speeds of off-chip buses is also very difficult due to the long, heavily capacitive PCB traces. On-chip memory controllers, however, could implement wider and/or faster interfaces between the processor and the MC, but this does not address the connection between the MC and main memory.

## 2.2. 3D Integration

There are many candidate technologies for 3D die stacking, but wafer-to-wafer bonding appears to be the leading contender in industry [5, 13, 30]. Many recent academic studies have assumed this type of 3D stacking technology [20, 26, 27, 29, 32]. Figure 2 shows a 3D stack. Each layer consists of a die manufactured in a typical 2D technology. The individual layers each consist of bulk silicon, active devices, and multiple layers of metal routing. After fabrication, each wafer is thinned to only 10-100 $\mu$ m in thickness [5, 8], and then Through Silicon Vias (TSVs) are etched through the bulk silicon. Thermocompression is then used to bond the individual layers together to form the 3D stack [25].

For the computer architect, the main 3D attributes of interest are (1) the ability to combine different process technologies, (2) the latency of the TSVs, and (3) the size of the TSVs. The first point is obviously necessary to stack DRAM on top of a processor. The second point is important because if the delay required to communicate between stacked layers is of the same order of magnitude as off-chip communication, then there will be very little potential for improving performance. Fortunately, it has been reported that the vertical latency for traversing the height of a 20-layer stack is only 12 ps [26]. Even though the TSVs are wider than

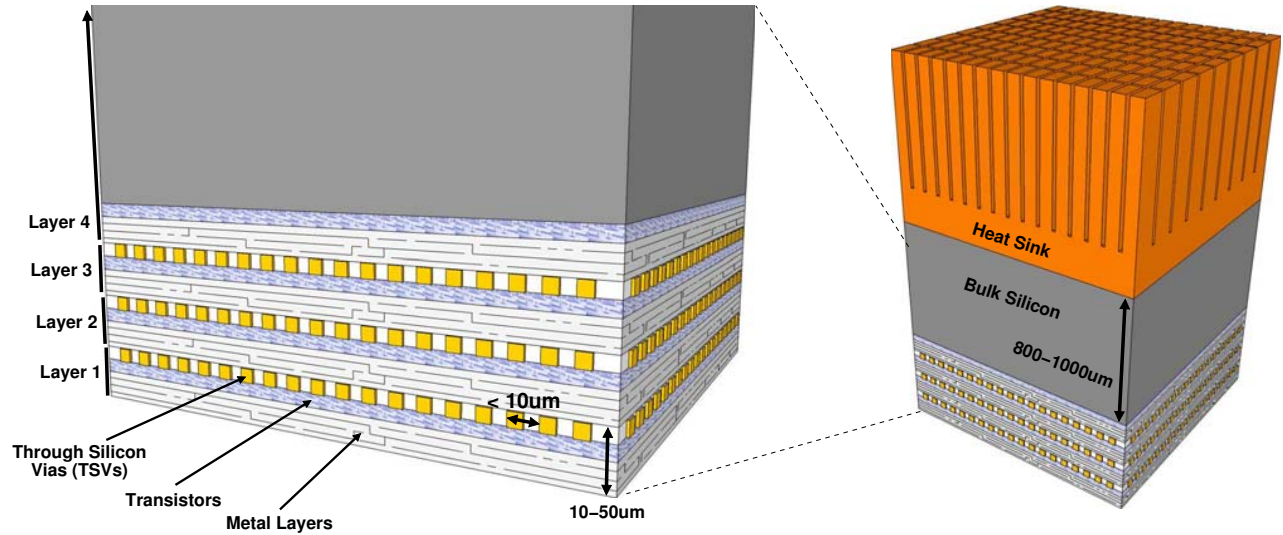


Figure 2. Cross-sectional view of 3D-stacked dies with a face-to-back topology (*not drawn to scale*).

typical metal routes, they are very short since each wafer is thinned to only tens of microns. The TSVs have been reported to have pitches of only 4-10 $\mu\text{m}$  [14]. Even at the high-end with a 10 $\mu\text{m}$  TSV-pitch, a 1024-bit bus would only require an area of 0.32mm<sup>2</sup>. To put that in perspective, a 1cm<sup>2</sup> chip could support over three hundred of these 1Kb buses. For the purposes of stacking DRAM, the TSV size and latency will likely not be a limiting factor for several generations.

### 2.3. 3D-Stacked DRAM

Previous studies have already started exploring the performance benefits of using 3D integration to stack main memory on top of a processor. Figure 3(a) shows a traditional 2D processor core with multiple layers of 3D-stacked DRAM [20, 24, 26]. The 3D TSVs implement a vertical bus across the layers to connect the DRAM layers to the processor core. Independent of the required latencies, the *topology* and overall architecture of these processor-memory 3D organizations are no different than that used in traditional off-chip memories. Liu et al. [24] and Kgil et al. [20] do consider using the dense TSV interface to implement wider buses, which are traditionally limited by pin-count, but they do not observe a large performance benefit due to the small sizes of their workloads and critical-word-first delivery of data from memory.

The previous approaches do not fully exploit 3D stacking technology because the individual structures are all still inherently two-dimensional. Tezzaron Corporation has announced “true” 3D DRAMs where the individual bitcell arrays are stacked in a 3D fashion [38, 39]. Figure 3(b) shows the overall organization. The top N layers consist of the stacked DRAM bitcells; this stacked organization reduces the lengths of internal buses, wordlines and bitlines, which

in turn reduces the access latency of the memory. The bottom layer implements the various control and access circuits, such as the row decoder, sense amplifiers, row buffers and output drivers. The advantage of isolating the peripheral circuitry to a separate, dedicated layer is that different process technologies can be incorporated. The DRAM bitcells are all implemented in a traditional NMOS technology optimized for density, whereas the peripheral circuits are implemented on a CMOS layer optimized for speed. The combination of reducing bitline capacitance and using high-speed logic provides a 32% improvement in memory access time ( $t_{\text{RAS}}$ ) for a five-layer DRAM (four layers of memory plus one layer of logic). Note that this latency reduction is for the memory array access itself, and this is additive with any benefits due to placing the memory closer to the processor.

### 2.4. Methodology and Assumptions

In this section, we briefly describe the processor microarchitecture, memory system and 3D integration assumptions that we use. We base our baseline processor on the Intel 45nm “Penryn” model [19], detailed in Table 1, extended to a quad-core configuration. We use the SimpleScalar toolset for the x86 ISA for performance evaluation [3], and we extended it to perform cycle-level modeling of a multi-core processor. The simulator models the contention for cache/memory buses, MSHR capacity, traffic due to write-back and prefetches, and memory controller request queue capacity. For the DRAM, we model the low-level timing including precharge delays (including when it can be overlapped with other commands), different timings for row buffer hits and misses, and the effects of periodic DRAM refreshing. We assume a memory controller implementation that attempts to schedule accesses to the same row together to increase row buffer hit rates [34]. For the off-chip DRAM, we assume a refresh period of 64ms, and the on-chip ver-

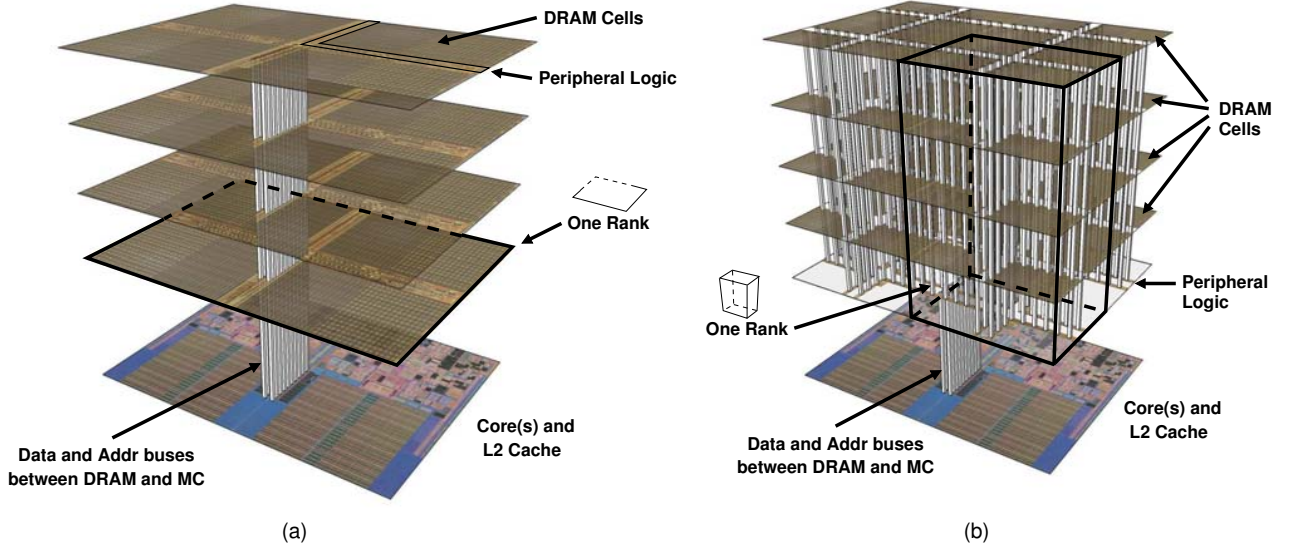


Figure 3. 3D-stacked DRAM on CPU with (a) one rank per layer and (b) ranks split across multiple layers.

Cores	4
Clock Speed	3.333 GHz
Fetch	16 bytes per cycle
Br. Pred.	TAGE (4KB, 5 tables) [35]
Fetch→Exec	14 stages min. mispred. penalty
Decoders	1 Complex, 3 Simple: 4/1/1/1
Dispatch Width	4 $\mu$ ops/cycle
RS/LDQ/STQ/ROB Sizes	32/32/20/96 entries
Exec Width	6 $\mu$ ops/cycle
Integer ALUs	3 Simple, 1 Branch, 2 Shift, 1 Mult, 1 Div
F.P. ALUs	1 Add, 1 Mult, 1 Div (shared with Int)
Ld/St Exec	1 Load, 1 Store Addr, 1 Store Data
Commit Width	4 $\mu$ ops/cycle
IL1/DL1	24KB, 12-way, 64-byte line, 2-cycle (+1 for addr. comp.), 8 MSHR
ITLB	32-entry, 4-way set-associative, 2-cycle
DTLB	64-entry, 4-way
Prefetchers	Nextline (IL1/DL1) IP-based Stride (DL1) [9]
DL2	12MB, 24-way, 64-byte line, 16 banks, 9-cycle, 8 MSHR
Prefetchers	Nextline, IP-based Stride
FSB	64-bit, 833.3 MHz ( $\rightarrow$ 1.66 GHz DDR)
Memory (2D)	8GB, 8 ranks, 8 banks, 64-bit FSB $t_{RAS}=36ns$ , $t_{RCD}$ , $t_{CAS}$ , $t_{WR}$ , $t_{RP}=12ns$ each
Memory (true-3D)	$t_{RAS}=24.3ns$ , $t_{RCD}$ , $t_{CAS}$ , $t_{WR}$ , $t_{RP}=8.1ns$ each

Table 1. Baseline quad-core processor parameters.

sions use a refresh period of 32ms to account for the higher leakage rates due to higher temperature operation [11].<sup>1</sup> We assume eight banks per rank and a memory page size of 4096 bytes.

Our simulator does not support the system calls required for cooperative multi-threaded applications, so we instead use multi-programmed workloads. We first observed the L2

<sup>1</sup>These refresh rates are consistent with those specified in the datasheets for the Samsung K4T51083QE DDR2 SDRAM part on which we base our memory parameters.

cache miss behaviors for a wide variety of applications and selected those that exhibited the highest cache miss rates when running alone on a single core with a 6MB L2 cache. The 24 applications listed in Table 2(a) had the highest L2 miss rates; these include programs from SPECcpu’s integer and floating point suites for both the 2000 and 2006 editions, bioinformatics workloads [1], MediaBench-I and II [10, 23], and one application from MiBench [15]. We also include the Stream workload which is particularly taxing on the memory system [28]. In addition to the original Stream workload, we also used decomposed versions of Stream where each version performs only one of the four basic operations used in the original benchmark (scale, copy, add and triad). In all cases, we use the SimPoint 3.2 toolset to choose representative execution samples [16].

We group our applications based on cache miss rates, and then we form several multi-programmed workloads as listed in Table 2(b). In this work, we are primarily concerned with workloads that exhibit relatively high memory demands, otherwise there would not be much point in designing faster memory architectures. These workloads are those with all high-miss (H) and very-high-miss (VH) applications. Nonetheless, we also consider workloads consisting of moderate-miss applications (M) and a mix of both high and moderate (HM) to ensure that our optimizations do not inadvertently degrade performance for these cases, as well as to provide an idea of how the benefits reduce when memory is not the major bottleneck. For all experiments, our primary results report the geometric mean speedups across the H and VH workloads. While we also provide supplementary speedup results across all four workload classes (H/VH/HM/M), most discussions will be focused only on

Benchmark Name and Suite			MPKI (6MB)
S.copy	Stream		326.9
S.add	Stream		313.2
S.all	Stream		282.2
S.triad	Stream		254.0
S.scale	Stream		252.1
tigr	BioBench		170.6
qsort	MiBench		153.6
libquantum	I'06		134.5
soplex	F'06		80.2
milc	F'06		52.6
wupwise	F'00		40.4
equake	F'00		37.3
lbm	F'06		36.5
mcf	I'06		35.1

(a)

Benchmark Name and Suite			MPKI (6MB)
mummer	BioBench		29.2
swim	F'00		18.7
omnetpp	I'06		14.6
applu	F'06		12.2
mgrid	F'06		9.2
apsi	F'06		3.9
h264	Media-II		2.9
mesa	Media-I		2.4
gzip	I'00		1.4
astar	I'06		1.4
zeusmp	F'06		1.4
bzip2	I'06		1.4
vortex	I'00		1.3
namd	F'06		1.0

(b)

High Miss Rate			HMIPC
H1	S.all, libquantum, wupwise, mcf		0.153
H2	tigr, soplex, equake, mummer		0.105
H3	qsort, milc, lbm, swim		0.406
Very High Miss Rate			HMIPC
VH1	S.all × 4		0.065
VH2	S.copy, S.scale, S.add, S.triad		0.058
VH3	tigr, libquantum, qsort, soplex		0.098
High-Moderate Mixes			HMIPC
HM1	tigr, equake, applu, astar		0.138
HM2	libquantum, mcf, apsi, bzip2		0.386
HM3	milc, swim, mesa, namd		0.907
Moderate Miss Rate			HMIPC
M1	omnetpp, apsi, gzip, bzip2		1.323
M2	applu, h264-decode, astar, vortex		1.319
M3	mgrid, mesa, zeusmp, namd		1.523

**Table 2. (a) Simulated benchmarks, originating suites (I=SpecInt, F=SpecFP, '00=cpu2000, '06=cpu2006, Media=MediaBench), and stand-alone (single-threaded) DL2 cache miss rates, (b) four-threaded workloads used in this study. HMIPC is the harmonic mean IPC across all four benchmarks of a workload when executing on the baseline processor configuration using off chip (non-3D) memory.**

the H and VH groups. We did evaluate many more application mixes from each of these groups, but the overall results and conclusions were the same and so we do not include them for the sake of brevity.

For each mix, we warm the caches for 500 million x86 macro instructions per application (which is usually more than 500 million  $\mu$ ops) prior to detailed cycle-level simulation for 100 million instructions. When an application finishes executing 100 million instructions, we freeze its statistics (e.g., IPC) but then continue simulating the program so that it continues to compete for the shared L2 cache and bus resources (this is effectively the same as some other recent multi-core simulation methodologies [33]). Since we collect statistics across an equal number of instructions per benchmark, the “total” reported IPC per workload uses the harmonic mean (HMIPC). Table 2(b) includes the HMIPC per workload on the baseline processor configuration with off-chip DRAM.

We assume a total of 8GB of physical memory for our quad-core system. We perform a virtual-to-physical memory translation/allocation based on a first-come-first-serve basis, and all accesses to the L2 and main memory deal with physical addresses only. Current DRAM memory densities are as high as 10.9 megabits (Mb) per  $\text{mm}^2$  (or 1.36 megabytes/MB) in a 80nm process. Scaling this to 50nm yields a density of 27.9Mb/ $\text{mm}^2$  (3.5MB). For our study, we assume 1GB per layer, which implies an overall per-layer footprint requirement of 294 $\text{mm}^2$ . This die size is within the range of current quad-core systems; for example, the AMD Barcelona 45nm quad-core processor is reported to have a die size of 285 $\text{mm}^2$  [2]. This means we need eight stacked layers (nine if the logic is implemented on a separate layer as described in Section 2.3) to realize 8GB of physical memory.

We also conduct thermal analysis of 3D-stacked DRAMs using the University of Virginia HotSpot toolset [36]. Our

findings, however, are consistent with previously reported results [26], and so we omit them due to space limitations. The important result is that the worst-case temperature across the entire 3D stack is still within the maximum thermal limit of the SDRAMs as specified by the Samsung datasheets.

### 3. DRAM-on-Processor Organizations

In this section, we present our performance analysis of 3D-stacked memory-on-processor configurations based on traditional memory architectures. For the previously proposed configurations [20, 24, 26], we assume an organization with one rank per die. Since we evaluate a system with 8GB and 1GB per die, this results in eight total ranks. Figure 4 shows the performance speedups of simple 3D-DRAM stacking. The 2D configuration is the baseline where main memory is located off chip. The memory controller is on-chip, but we assume that it is clocked at the FSB speed. The 3D configuration assumes the memory is now stacked on the cores and the FSB and memory controller run at the same speed as the processor core. The timing of the memory arrays themselves are still the same (i.e.,  $t_{CAS}$ ,  $t_{RAS}$ , etc. are unchanged; this is consistent with the methodology used by Loi et al. for their 3D memory parameters [26]). 3D-wide is the same as 3D, except that the bus has been increased to 64 bytes which matches the size of an entire L2 cache line. Finally, 3D-fast extends 3D-wide to make use of “true” 3D-split DRAMs in a 9-layer implementation ( $8 \times \text{memory} + 1 \times \text{logic}$ ). The numbers reported in Tezzaron’s datasheets indicate that the memory access latency for a five-layer implementation improves by 32.5% over a traditional 2D memory. While we would expect an even greater benefit for a nine-layer version due to further reductions in intra-array interconnects, we conservatively assume a latency benefit equal to that of the five-layer version. The exact timings are listed in Table 1(b), although everything is rounded up to be integral



multiples of the CPU cycle time (e.g., the CAS latency (CL) is an integer number of cycles).

The results in Figure 4 show that for even these simple approaches to 3D-stacking memory, the performance benefits are quite substantial. Note that all results are reported as speedups over the 2D (off-chip DRAM) configuration. Simply placing the DRAM on the same stack as the processor (3D) provides a 34.7% (26.3%) increase in performance on our memory-intensive workloads (all workloads). Note that this is simply due to reducing wire delay between the memory controller and main memory, and running the memory controller at a faster speed.

Implementing a wider bus to memory enables data transactions to complete in fewer bus cycles. Expanding the bus to 64 bytes increases the performance benefit to 71.8% over 2D. It has been previously reported that increasing bus width does not provide a significant performance boost since the multiple cycles of data transfer occur with critical-word-first ordering which effectively does not impact a load’s execution latency [24]. The difference is that in our simulations, the contention for the memory bus is much greater due to differences in the application selection (e.g., SPEC2006 has much larger working sets than SPEC2000, especially for FP [12]) and our multi-core configuration directly multiplies the amount of memory traffic as well. Increasing the bus width allows each L2 miss to occupy the bus for many fewer cycles which allows memory transactions from other cores to proceed that much sooner. These results demonstrate that different baseline assumptions can lead to very different conclusions and recommendations for the design of 3D-stacked DRAMs.

Finally, implementing the DRAM arrays as true 3D structures reduces the actual latency per access, resulting in a total of 116.8% performance improvement ( $2.17\times$  speedup) over 2D. The benefits do vary by workload; for example, the moderate-miss applications do not observe as large of a benefit, but this makes sense since these programs have better L2 cache hit rates and therefore spend less time waiting on memory in the first place. The general trends are fairly consistent, with each optimization from 2D to 3D to 3D-wide to 3D-fast providing an approximately equal additional performance boost. This indicates that in a heavily loaded memory system, the latency to memory (MC→DRAM), the latency of memory (array access time), and the latency to even get to access memory (bus contention) are all very significant factors. 3D-fast represents the *combination* of all of these previously proposed techniques, and we use this high-performance configuration as our baseline for comparison in the next section. That is, our reported performance gains are *in addition* to those already provided by this 3D-fast configuration.

#### 4. More Aggressive 3D Memory Organizations

We have verified that even simple approaches to 3D stacking can provide a significant benefit on our workloads. Note

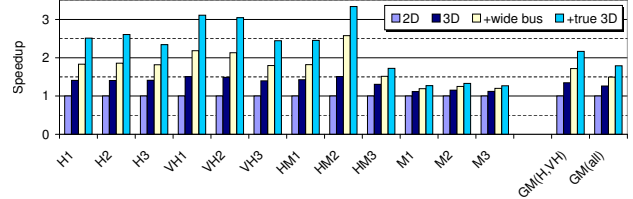


Figure 4. Performance speedup of simple 3D-stacked memories compared to off-chip (2D) memory.

that the topology of these organizations are largely the same as traditional off-chip memories: single shared bus, single memory controller interface, relatively few ranks. Note that for all remaining 3D-stacked DRAM designs discussed in this paper, we will assume “true” 3D-split organizations.

##### 4.1. Ranks and Memory Interfaces

We first consider varying two parameters: the number of ranks and the number of memory controller interfaces. Increasing the number of ranks is straightforward as it simply amounts to reorganizing the memory into a larger number of smaller arrays. This is traditionally not practical due to pin and chip-count limitations, but the plentiful die-to-die bandwidth in 3D makes this a viable option. There are a few ways to increase the number of memory controller interfaces. One approach is to use a single memory controller, but increase the number of channels where each channel accesses a disjoint set of ranks. Another approach is to implement multiple, or banked, memory controllers. Traditionally, multiple memory controllers would not be practical because the CPU package would require more pins to independently communicate with the different MCs, and then the additional MCs increase the chip count and overall motherboard routing and total area. In a 3D stack, however, the routing is far more flexible, and, so long as silicon area is available, implementing additional MCs is not a great hurdle. Note that the peripheral logic layer in Figure 3(b) contains a significant amount of whitespace that can be used for the additional logic that we require. Figure 5 shows the different design options that we consider with varying numbers of ranks and MCs. The bold lines illustrate which ranks are accessed by which MCs.

We maintain a constant total aggregate capacity of 32 requests across all MCs for all experiments. For example, a four-MC configuration would have an eight-entry request queue per MC. To the first order, the amount of MC state, area, and power will be proportional to the total aggregate capacity, and therefore the area and power are about the same across configurations, independent of the total number of MCs used. The MC arbitration/scheduling logic actually becomes less costly since four 8-entry schedulers are generally easier to implement than a single, monolithic 32-entry scheduler.

It is necessary to briefly discuss the L2 cache and miss handling architecture organizations. The example in Fig-

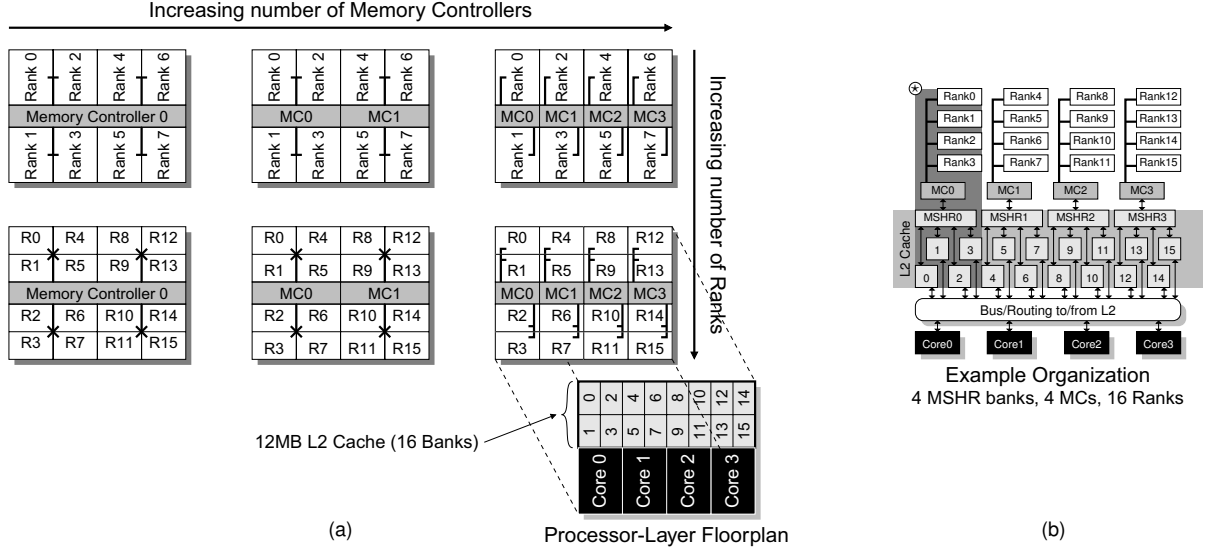


Figure 5. (a) Floorplan examples for varying the number of memory controllers and ranks. (b) Example configuration.

Figure 5(b) illustrates the overall organization of a 16-bank L2 cache with four-way banked memory controllers and 16 memory ranks. At the L2 level, bank interleaving typically occurs at the granularity of cache lines. That is, an access to address  $x$  would get routed to L2 bank number  $x \gg 6 \pmod{16}$ , assuming 64-byte cache lines and 16 banks. Main memory, however, is interleaved based on the physical page size (4KB in our study). The mismatch in banking granularities means that any L2 bank may need to read-from/write-to any memory controller, requiring either a bus or a crossbar connecting every L2 bank to every MC. Therefore, we change the banking of the L2 such that the interleaving occurs on 4096-byte boundaries rather than 64 bytes. This results in a far more streamlined communication layout, where each set of L2 banks routes to one and only one MC. We also bank our MSHRs [37] in a similar fashion such that when a miss occurs in one of the L2 cache banks, the miss attempts to allocate an MSHR only in the bank aligned with its memory controller. Note that the only significant “horizontal” routing occurs in getting to an L2 bank; after that, all remaining communications are effectively “vertical.” For example in Figure 5(b), if Core 3 accesses an address in bank 2, after the request reaches cache bank 2, all subsequent communications remain within the shaded region marked with a star (★) since a miss in bank 2 can only allocate in MSHR0, which can only communicate with MC0, which can only access ranks 0 through 3.

Figure 6(a) shows the results of increasing the number of ranks and memory controllers. We report the geometric mean speedups over the 3D-fast configuration for the H and VH workloads (the numbers in parentheses are the geometric mean speedups across all four workload groups). The per-workload trends are similar to those reported in Figure 4. As we increase the number of memory channels, per-

formance improves because a larger number of memory requests can be processed in parallel. In addition to the increased parallelism, performance also benefits for other reasons. First, since each MSHR bank directly communicates with its banked MC, there is no need for a globally shared bus between all MCs which greatly reduces delays due to bus contention. Second, the MC schedulers process requests from only a subset of the overall physical memory space, which increases the opportunities to find and reschedule accesses to open pages.

As we increase the number of ranks, performance also improves, but not nearly by as much as increasing the number of memory interfaces. Increasing the number of ranks directly increases the total number of banks in the memory system. Since each bank has its own row buffer, this results in a memory system being able to keep a much larger number of rows open at the same time. Even though the performance benefits are rather small in comparison to the previous changes, increasing the number of ranks is likely desirable because it reduces the size of each of the individual banks. This can decrease the lengths of wordlines and bitlines, resulting in further reductions in access latency and simultaneous reductions in the dynamic power consumed per access. Overall, increasing the number of ranks to 16 and the number of MCs to 4 provides an average performance benefit of 33.8% on top of the 3D-fast configuration, on our memory-intensive workloads.

Increasing the number of ranks also increases the total number of row buffers which requires additional logic; additional logic that could possibly be put to other uses. An eight-rank configuration has eight banks per rank, with one 4KB row buffer per bank. This totals to 256KB of storage to implement all of the row buffers. Increasing this to 16 banks requires an additional 256KB for the row buffers. There-

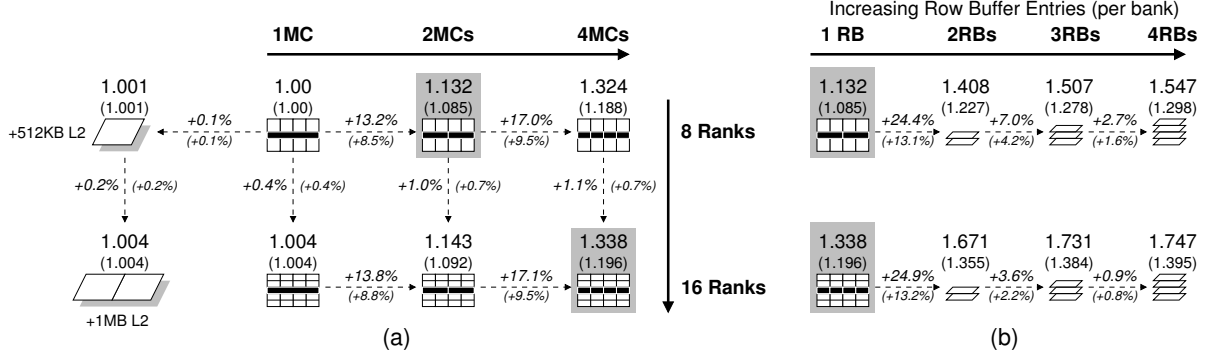


Figure 6. Performance benefit over the 3D-fast configuration for (a) increasing memory controllers and memory ranks, and then (b) further increasing the number of row buffer entries per bank.

fore we also include results for the original 8-rank 3D-fast where we instead use the additional transistor budget to increase the L2 cache size by 512KB and 1MB. Despite the fact that the additional L2 storage would be faster to access than main memory, adding less state in the form of more row buffers is actually better than adding more state in the form of additional L2 cache. The reason for this is that, for our memory-intensive workloads, the additional L2 cache only slightly decreases the number of misses while the increase in the row buffer hit rate greatly reduces the average latency for the many remaining misses.

#### 4.2. Making Use of the Extra Space

When implementing “true” 3D-stacked memory arrays, the amount of silicon required on the bottom logic layer does not actually consume all of the available silicon real estate. Increasing the number of ranks would consume some of this area as mentioned earlier. We also consider one further optimization, which is to increase the storage capacity from a single row buffer entry to a multiple-entry row buffer cache (also called *cached DRAM*) [17]. Any access to a memory bank performs an associative search on the set of row buffers, and a hit avoids accessing the main memory array. We manage the row buffer entries in an LRU fashion.

Figure 6(b) shows the performance impact of increasing the number of row-buffer entries per bank from one (baseline) to four for the two highlighted configurations from Figure 6(a). We chose these two configurations as they represent different levels of aggressiveness in terms of modifying the memory system organization. The majority of the benefit comes from adding the first additional row buffer entry. Even with diminishing performance benefits, adding more row buffer entries may still be desirable because each row buffer cache hit avoids the power needed to perform a full array access. Increasing row buffer cache size and increasing the number of ranks both result in an overall increase in the number of open pages; adding row buffer cache capacity, however, is clearly a bigger win. The reason for this is very similar to how increasing a cache size by increasing the

number of sets is often not as effective as increasing the set associativity.

Section 3 demonstrated that previously explored “straight-forward” 3D-DRAMs provide considerable performance benefits. An important conclusion of our results is that to fully exploit 3D technology, the interface between the stacked modules needs to be re-optimized. We emphasize that the  $1.75\times$  speedup shown in Figure 6(b) is *in addition* to the  $2.17\times$  speedup provided by the full combination of the simpler 3D-memory stacking techniques (for a total of  $3.80\times$  speedup over 2D).

### 5. A Scalable L2 Miss Handling Architecture

We have thus far kept the capacity of the L2 miss handling architecture (MHA) constant across all configurations so that performance improvements could be entirely attributed to the changes in the 3D-stacked memory. The organization and capacity of the MHA does, however, play a significant role on overall performance.

#### 5.1. Increasing MSHR Capacity

Our baseline configurations all assumed MSHR capacities of eight entries for each of the DL1 caches and eight for the shared L2. At first, this may seem like the L2 MSHR may be an unfairly constrained bottleneck since four cores times eight L1 MSHR entries would suggest that the L2 should have 32 MSHR entries. Note that, however, most DL1 misses should (hopefully) still result in L2 hits, in which case fewer L2 MSHRs would be required. Zhao et al.’s multi-core study assumed 16 L2 MSHRs for a 32-core system [42], and Kim et al. used 8 MSHRs for a quad-core system [21]; our baseline configuration is similarly sized and thus represents a reasonably fair baseline. Furthermore, under-provisioning the MSHR capacity is conservative for our previous studies because it prevents full utilization of the additional bandwidth and memory parallelism provided by our 3D-stacked memory organization.

We now explore the performance impact of increasing the number of outstanding misses from the L2. We start with



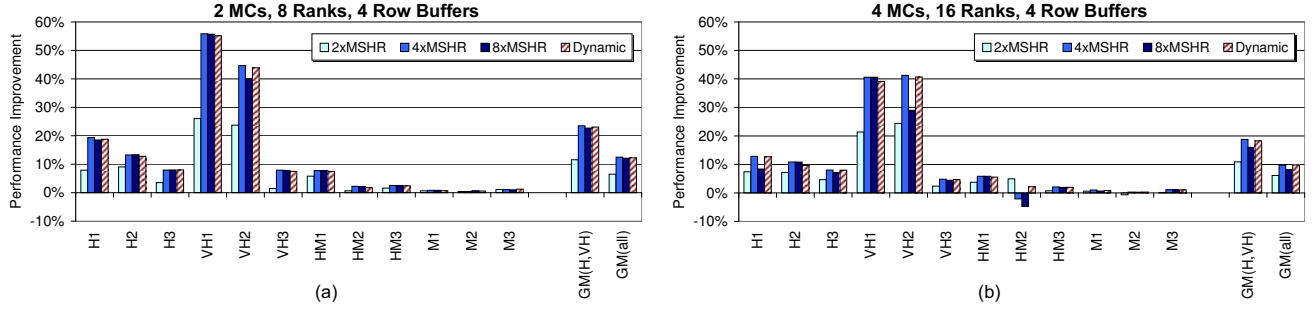


Figure 7. Performance impact of increasing the L2 MSHR capacity for (a) dual-MC/8-rank and (b) quad-MC/16-rank 3D-DRAM.

two baseline configurations: one with two MCs and eight ranks, the other with four MCs and sixteen ranks. Both configurations have four row-buffer entries per bank and correspond to the two rightmost configurations in Figure 6(b). From here, we increased the MSHR capacity of each configuration by factors of 2, 4 and 8. Figure 7 shows the performance impact of varying the total MSHR capacity. Overall, performance improves when doubling and quadrupling the capacity of the L2 MSHR. For applications like the Stream benchmarks that generate a very large number of misses, increasing the L2 miss handling capacity will obviously be of benefit. Further increasing the MSHR capacity either provides no significant additional benefit, or in some cases hurts performance compared to the  $4\times$  capacity configuration. For two of the lower memory traffic workloads (HM2 and M2), increasing MSHR capacity can result in a performance degradation compared to the baseline MSHR size. While the high-memory traffic workloads are the focus of this study, optimizations that negatively impact other application mixes are not desirable. After carefully analyzing simulator statistics, we found that increasing the number of outstanding misses can have a negative effect on L2 cache hit rates. The many additional requests can cause the eviction of lines that are still useful. Effectively, by having too many requests coming and going, this causes significant increase in L2 “churn” resulting in performance losses for these workloads.

To provide greater MSHR capacity while at the same time preventing any performance degradations, we make use of a simple dynamic MSHR capacity tuning scheme. Each MSHR has a limited number of possible sizes;  $1\times$ ,  $\frac{1}{2}\times$  and  $\frac{1}{4}\times$  the maximum MSHR size in our simulations. The resizing algorithm uses a brief training phase where the processor runs with different limits on the MSHR size and records the number of committed  $\mu$ ops from each core for each setting. The processor then fixes the MSHR size at the setting that performed the best during sampling until the next sampling period. Similar training techniques have been used for other processor optimizations [4, 31]. Figure 7 also includes the performance of dynamic MSHR capacity tuning. While this technique does not provide any significant overall per-

formance benefit, it yields a more *robust* design where the larger MSHRs do not negatively impact performance on any of our workloads.

## 5.2. Scalable L2 MSHRs

Having demonstrated that larger MSHRs can be consistently put to use to increase performance, we now discuss the problem of building larger MSHRs. MSHRs are relatively complex, fully associative structures. Traditional CAM-based implementations do not scale well, which has lead researchers to invent other organizations. Tuck et al. proposed a hierarchical MSHR organization that uses several small, banked MSHRs at the first level, and then a larger shared MSHR at the second level [40]. This organization provides high bandwidth because all of the banks can be accessed independently. The shared MSHR provides additional “spare” capacity when an individual bank receives too many requests. We do not use hierarchical MSHRs for the L2 cache because all MSHR banks would need to route to the shared MSHR, and then the actual miss request must route back to the banked memory controllers. This would break the streamlined floorplan illustrated in the example of Figure 5. Note that the hierarchical MSHRs may still be a reasonable match for a single-MC organization.

At the L1 level, high bandwidth is an important requirement of the MHA because the relatively higher miss rates of the DL1 caches cause the MSHRs to be accessed far more frequently. At the L2 level, however, MSHR bandwidth is less important and the main goal is simply capacity. We propose a new MSHR organization that uses a very scalable direct-mapped structure rather than a fully-associate CAM-based organization. We start with a simple direct-mapped hash table. The address of the access can be used to hash into a particular entry. If there is a miss, then one can simply proceed to check the next sequential entries until a hit is found, or all entries have been checked which would indicate a miss. For even moderately-sized MSHRs, however, this would lead to a large number of probings, especially in the case of a miss where all entries must be checked.

We propose a new data structure called the Vector Bloom Filter (VBF) that reduces the number of probings for both hit and miss situations. We will explain the operation of

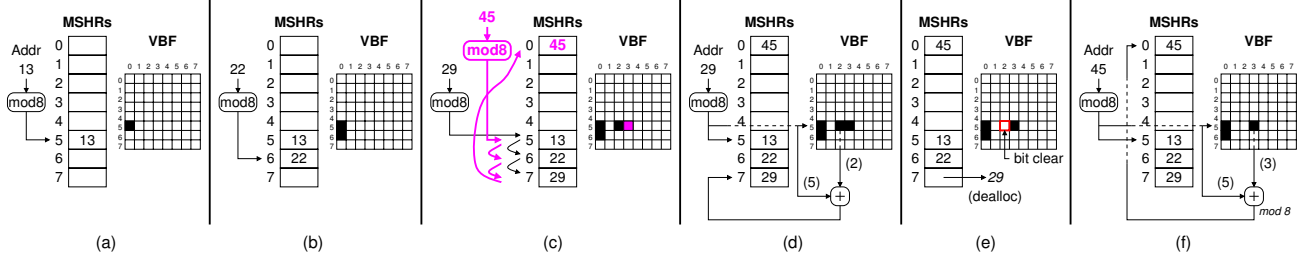


Figure 8. Example operation of a direct-mapped MSHR with the Vector Bloom Filter.

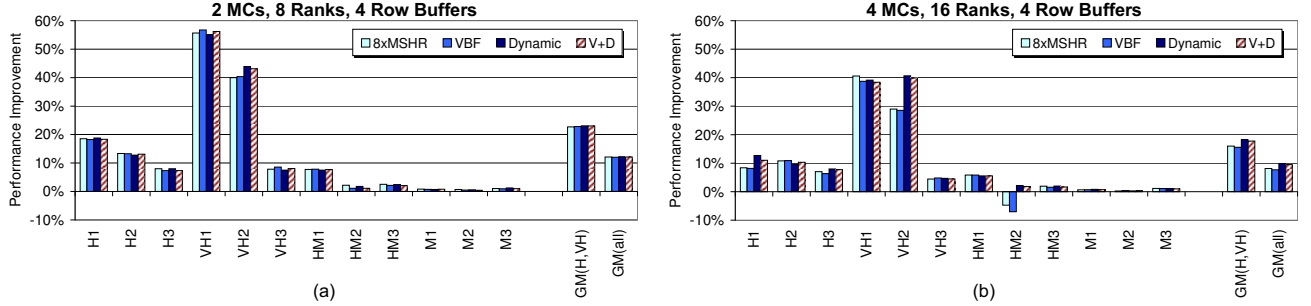


Figure 9. Additional performance benefit of the proposed L2 MHA over our 3D-stacked memory organizations.

the VBF and the direct-mapped MSHR with the example shown in Figure 8. The MSHR is initially empty. (a) A miss on address 13 occurs, and we compute a hash based on the number of MSHR entries; in this case  $13 \bmod 8$  yields an index of 5, and we allocate this entry. On the side, we have our VBF data structure which consists of one row (vector) per MSHR entry, with one column per MSHR entry as well. Using the same index, we choose the fifth row of the VBF, and we place a 1 (■) in column zero. (b) We have a different miss to address 22, which maps to index 6. We allocate entry 6 and set the bit in column zero of row 6 of the VBF. (c) We have an access to address 29 which also maps to index 5. In this case, MSHR 5 is already used, so we allocate the next sequentially available entry, which turns out to be entry 7. This time in the VBF, we mark the bit in *column two* of row 5. The reason for this is that we have now allocated an entry that is two positions from the default position of 5. A subsequent miss for address 45 also maps into the same set and gets allocated entry 0 (the updates for this are shown in red/grey). (d) We search for address 29, and we access MSHR entry 5 and the VBF in parallel. If there is a match with MSHR 5, then we have a hit and are done. In this case, there is a miss and so we consult VBF row 5 to determine the next MSHR entry to search. The bit vector indicates that the next possible candidate is two entries away, and so on the next cycle we check MSHR entry 7 which turns out to be a hit. (e) The miss for address 29 has been serviced and we deallocate the corresponding MSHR entry. At this point, we simply invalidate the MSHR entry and clear the corresponding bit in the VBF (row 5, column 2). (f) A search for

address 45 occurs. The first step is the same as before where we check MSHR entry 5 and read row 5 from the VBF in parallel. This is a miss, so we consult the VBF information to determine the next MSHR entry to check. Since the previous deallocation of address 29 causes the bit in column 2 to get cleared, the next set bit is in column 3. Adding this offset to our default MSHR position tells us to check MSHR entry  $5 + 3 \pmod{8} = 0$ , where we have a hit. Note that with only linear probing (no VBF), this search for address 45 would have required a total of four probings (check MSHR entries 5, 6, 7 and 0), whereas the VBF reduces this to only two (5 and 0).

As described, our VBF-based MSHR is effectively a hashtable with linear probing,<sup>2</sup> augmented with the VBF which speeds up the search by remembering the entries that definitely do not contain the address that we are searching for. A ‘1’ in an VBF row does not guarantee that the requested address is at that (relative) location, but a ‘0’ means that it is definitely not. The possibility for a false hit is why we consider this algorithm to be a form of a Bloom Filter [6]. Note also that on a miss, we can quickly check to see if all columns in a row are zero; if that is the case, then there is definitely a miss and no further probings are required. The VBF size increases quadratically with the number of MSHR entries per bank, but banking results in relatively few en-

<sup>2</sup>We also experimented with other secondary hashing schemes, such as quadratic probing, to deal with potential problems of miss clustering. The VBF, however, does a sufficiently good job and reducing probings that there was no measurable difference between the different secondary hashing techniques that we studied.

tries per bank, and so even for the largest per-bank MSHR size that we consider (32 entries), the VBF bit-table only requires 128 bytes of state. The direct-mapped MSHR coupled with the VBF can provide scalable MSHR capacity for the L2 MHA. This approach is not likely to be as effective at the DL1 MSHRs due to the L1's need for much greater access bandwidth, and so previously described techniques such as the hierarchical MSHRs would be more appropriate there [40].

The VBF can potentially reduce performance by increasing the search latency of the MSHR structures. Figure 9 shows the performance impact of using our new MSHR scheme. Using the direct-mapped VBF-based L2 MHA, we achieve performance that is about the same as the ideal (and impractical) single-cycle, fully-associative traditional MSHR. The reason for this is that the VBF mechanism is very effective at filtering out unnecessary MSHR probings. On average, the dual-MC configuration required 2.31 probings per MSHR access and the quad-MC configuration needed 2.21 probings per access, which includes the first mandatory access. In the pathological case of HM2 on the four-MC configuration, the additional latency of the VBF checks results in an overall performance penalty of 7% compared to the baseline configuration with default MSHR sizing. The results in Figure 9 shows that by combining our VBF scheme with dynamic MSHR resizing (V+D), we can provide a robust, scalable, high-performance L2 MHA for our 3D-stacked memory architecture. Overall, we observe 23.0% (17.8%) improvements on our memory-intensive workloads over the baseline L2 MSHR architecture for the dual-MC (quad-MC) configuration. Note that these gains are in addition to those from increasing the ranks and memory interfaces of the 3D-stacked DRAM, which provides overall speedups of  $1.90\times$  ( $2.06\times$ ) over the previously proposed 3D DRAM organizations, and  $4.13\times$  ( $4.46\times$ ) over a conventional 2D system with off-chip DRAM. Even when considering all workloads including the less-memory-intensive applications, the overall speedups benefits are still  $1.46\times$  ( $1.53\times$ ) over the simple 3D DRAM organization, and  $2.60\times$  ( $2.73\times$ ) over off-chip memory.

Note that the benefits of our scalable L2 MHA are uniquely required for 3D-stacked memory architectures. We also evaluated the VBF+dynamic MSHR resizing approach for a conventional 2D multi-core processor with off-chip main memory and found that overall performance did not improve, and in a few cases even we observed minor performance degradations. The reason for this is in an off-chip memory organization, there are many other bottlenecks that must first be addressed before the L2 MSHRs constrain performance.

## 6. Conclusions

Previous studies have already demonstrated that 3D stacking of memory on processors can provide substantial relief from the Memory Wall problem. In this study, we have demon-

strated that the prior attempts have left a lot of the potential performance benefit on the table. We have demonstrated how to organize the memory architecture in a 3D environment to increase memory level parallelism through a streamlined arrangement of L2 cache banks, MSHRs, memory controllers and the memory arrays themselves. We have also taken advantage of the additional integration capacity to improve page-level parallelism by increasing row buffer cache capacity to maintain a larger set of open memory pages. Finally, we introduced a novel data structure called the Vector Bloom Filter to enable a scalable L2 miss handling architecture to complement the increased capacity of our 3D-stacked memory system.

The main conclusion of this work is that to fully exploit 3D integration technology, we need to redesign and re-architect those components that we wish to stack. While there exist many “low-hanging fruits” such as stacking conventionally-organized memory on a processor, stacking more cache on a processor, and stacking multiple cores, which will likely be pursued by industry in the shorter term, we believe the most interesting computer architecture research opportunities lie in efforts that target what can be done with 3D beyond these natural first steps. We have demonstrated that revisiting the memory system organization in a 3D context can provide much more performance than simply stacking “2D memory” on top of a processor, and there are likely similar benefits for reconsidering the use of 3D in the design of the processor core, multi-core organizations, and even future many-core platforms.

## Acknowledgments

This project was funded by NSF grant CCF-0643500; support was also provided by the Focus Center for Circuit & System Solutions (C2S2), one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation Program. Equipment was provided by a grant from Intel Corporation.

## References

- [1] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung. BioBench: A Benchmark Suite of Bioinformatics Applications. In *Proceedings of the Intl. Symp. on Performance Analysis of Systems and Software*, pages 2–9, 2005.
- [2] AMD Corporation. Barcelona PR Fact Sheet. <http://www.amd.com>, September 2007.
- [3] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Micro Magazine*, pages 59–67, February 2002.
- [4] R. I. Bahar and S. Manne. Power and Energy Reduction Via Pipeline Balancing. In *Proceedings of the 28th Intl. Symp. on Microarchitecture*, pages 218–229, 2001.
- [5] B. Black, M. M. Annamaram, E. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCauley, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. P. Shen, and C. Webb. Die-Stacking (3D) Microarchitecture. In *Proceedings of the 39th Intl. Symp. on Microarchitecture*, 2006.

- [6] B. H. Bloom. Space/Time Tradeoffs in Hash Coding with Allowable Errors. *Communications of the Association for Computing Machinery*, 13(7):422–426, July 1970.
- [7] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A Performance Comparison of Contemporary DRAM Architectures. In *Proceedings of the 26th Intl. Symp. on Computer Architecture*, pages 222–233, 1999.
- [8] S. Das, A. Fan, K.-N. Chen, and C. S. Tan. Technology, Performance, and Computer-Aided Design of Three-Dimensional Integrated Circuits. In *Proceedings of the Intl. Symp. on Physical Design*, pages 108–115, 2004.
- [9] J. Doweck. Inside Intel Core Microarchitecture and Smart Memory Access. White paper, Intel Corporation, 2006. <http://download.intel.com/technology/architecture/sma.pdf>.
- [10] J. E. Fritts, F. W. Steiling, and J. A. Tucek. MediaBench II Video: Expediting the Next Generation of Video Systems Research. *Embedded Processors for Multimedia and Communications II, Proceedings of the SPIE*, 5683:79–93, March 2005.
- [11] M. Ghosh and H.-H. S. Lee. Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs. In *Proceedings of the 40th Intl. Symp. on Microarchitecture*, 2007.
- [12] D. Gove. CPU2006 Working Set Size. *Computer Architecture News*, 35(1):90–96, March 2007.
- [13] K. W. Guarini, A. W. Topol, M. Jeong, R. Yu, L. Shi, M. R. Newport, D. J. Frank, D. V. Singh, G. M. Cohen, S. V. Nitta, D. C. Boyd, P. A. O’Neil, S. L. Tempest, H. B. Pogge, S. Purushothaman, and W. E. Haensch. Electrical Integrity of State-of-the-Art 0.13 $\mu$ m SOI CMOS Devices and Circuits Transferred for Three-Dimensional (3D) Integrated Circuit (IC) Fabrication. In *Proceedings of the Intl. Electron Devices Meeting*, pages 943–945, 2002.
- [14] S. Gupta, M. Hilbert, S. Hong, and R. Patti. Techniques for Producing 3D ICs with High-Density Interconnect. In *Proceedings of the 21st Intl. VLSI Multilevel Interconnection Conf.*, 2004.
- [15] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the 4th Work. on Workload Characterization*, pages 83–94, 2001.
- [16] G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: Faster and More Flexible Program Analysis. In *Proceedings of the Work. on Modeling, Benchmarking and Simulation*, 2005.
- [17] H. Hidaka, Y. Matsuda, M. Asakura, and K. Fujishima. The Cache DRAM Architecture. *IEEE Micro Magazine*, 10(2):14–25, April 1990.
- [18] I. Hur and C. Lin. Adaptive History-Based Memory Schedulers. In *Proceedings of the 37th Intl. Symp. on Microarchitecture*, pages 343–354, 2004.
- [19] Intel Corporation. Introducing the 45nm Next Generation Intel Core Microarchitecture. *Technology@Intel Magazine*, 4(10), May 2007.
- [20] T. H. Kgil, S. D’Souza, A. G. Saidi, N. Binkert, R. Dreslinski, S. Reinhardt, K. Flautner, and T. Mudge. PicoServer: Using 3D Stacking Technology to Enable a Compact Energy Efficient Chip Multiprocessor. In *Proceedings of the 12th Symp. on Architectural Support for Programming Languages and Operating Systems*, 2006.
- [21] W. Kim, M. S. Gupta, G.-Y. Wei, and D. M. Brooks. Enabling On-Chip Switching Regulators for Multi-Core Processors using Current Staggering. In *Proceedings of the Work. on Architectural Support for Gigascale Integration*, 2007.
- [22] D. Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th Intl. Symp. on Computer Architecture*, pages 81–87, 1981.
- [23] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *Proceedings of the 30th Intl. Symp. on Microarchitecture*, pages 330–335, 1997.
- [24] C. C. Liu, I. Ganusov, M. Burtscher, and S. Tiwari. Bridging the Processor-Memory Performance Gap with 3D IC Technology. *IEEE Design and Test of Computers*, 22(6):556–564, November–December 2005.
- [25] G. H. Loh, Y. Xie, and B. Black. Processor Design in 3D Die-Stacking Technologies. *IEEE Micro Magazine*, 27(3), May–June 2007.
- [26] G. L. Loi, B. Agarwal, N. Srivastava, S.-C. Lin, and T. Sherwood. A Thermally-Aware Performance Analysis of Vertically Integrated (3-D) Processor-Memory Hierarchy. In *Proceedings of the 43rd Design Automation Conf.*, 2006.
- [27] N. Madan and R. Balasubramonian. Leveraging 3D Technology for Improved Reliability. In *Proceedings of the 40th Intl. Symp. on Microarchitecture*, 2007.
- [28] J. D. McCalpin. Stream: Sustainable Memory Bandwidth in High Performance Computers. Technical report, <http://www.cs.virginia.edu/stream/>.
- [29] S. Mysore, B. Agarwal, S.-C. Lin, N. Srivastava, K. Banerjee, and T. Sherwood. Introspective 3D Chips. In *Proceedings of the 12th Symp. on Architectural Support for Programming Languages and Operating Systems*, 2006.
- [30] D. Nelson, C. Webb, D. McCauley, K. Raol, J. Rupley, J. DeVale, and B. Black. A 3D Interconnect Methodology Applied to iA32-class Architectures for Performance Improvements through RC Mitigation. In *Proceedings of the 21st Intl. VLSI Multilevel Interconnection Conf.*, 2004.
- [31] D. V. Ponomarev, G. Kucuk, and K. Ghose. Dynamic Allocation of Datapath Resources for Low Power. In *Proceedings of the Work. on Complexity-Effective Design*, Göteborg, Sweden, June 2001.
- [32] K. Pottaswamy and G. H. Loh. Thermal Herding: Microarchitecture Techniques for Controlling HotSpots in High-Performance 3D-Integrated Processors. In *Proceedings of the 13th Intl. Symp. on High Performance Computer Architecture*, 2007.
- [33] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Intl. Symp. on Microarchitecture*, pages 423–432, 2006.
- [34] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *Proceedings of the 27th Intl. Symp. on Computer Architecture*, pages 128–138, 2000.
- [35] A. Seznec and P. Michaud. A Case for (Partially) TAGged GEometric History Length Branch Prediction. *Journal of Instruction Level Parallelism*, 8:1–23, 2006.
- [36] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-Aware Microarchitecture. In *Proceedings of the 30th Intl. Symp. on Computer Architecture*, pages 2–13, 2003.
- [37] G. S. Sohi and M. Franklin. High-Bandwidth Data Memory Systems for Superscalar Processors. In *Proceedings of the 18th Intl. Symp. on Computer Architecture*, pages 53–62, 1991.
- [38] Tezzaron Semiconductors. Leo FaStack 1Gb DDR SDRAM Datasheet. [http://www.tezzaron.com/memory/TSC\\_Leo.htm](http://www.tezzaron.com/memory/TSC_Leo.htm), August 2002.
- [39] Tezzaron Semiconductors. Tezzaron Unveils 3D SRAM. Press Release from <http://www.tezzaron.com>, January 24 2005.
- [40] J. M. Tuck, L. Ceze, and J. Torrellas. Scalable Cache Miss Handling for High Memory Level Parallelism. In *Proceedings of the 39th Intl. Symp. on Microarchitecture*, 2006.
- [41] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):20–24, March 1995.
- [42] L. Zhao, R. Iyer, S. Makineni, J. Moses, R. Illikkal, and D. Newell. Performance, Area and Bandwidth Implications on Large-Scale CMP Cache Design. In *Proceedings of the Work. on Chip Multiprocessor Memory Systems and Interconnects*, 2007.