Extending the Effectiveness of 3D-Stacked DRAM Caches with an Adaptive Multi-Queue Policy

Gabriel H. Loh
Georgia Institute of Technology
College of Computing
Atlanta, GA, USA
loh@cc.gatech.edu

ABSTRACT

3D-integration is a promising technology to help combat the "Memory Wall" in future multi-core processors. Past work has considered using 3D-stacked DRAM as a large last-level cache (LLC). While significant performance benefits can be gained with such an approach, there remain additional opportunities beyond the simple integration of commodity DRAM chips. In this work, we leverage the hardware organization typical of DRAM architectures to propose new cache management policies that would otherwise not be practical for standard SRAM-based caches. We propose a cache where each set is organized as multiple logical FIFO or queue structures that simultaneously provide performance isolation between threads as well as reduce the number of entries occupied by dead lines. Our results show that beyond the simplistic approach of stacking DRAM as cache, such tightly-integrated 3D architectures enable new opportunities for optimizing and improving system performance.

Categories and Subject Descriptors

B.3.1 [**Memory Structures**]: Semiconductor Memories

General Terms

Design, Performance

1. INTRODUCTION

Three-dimensional die-stacking is a promising technology for increasing the integration capacity of future microprocessors and computer systems. Many researchers have proposed a variety of 3D architectures that stack multiple cores [16, 18, 30, 43], or even split the pipeline over multiple layers [7, 31, 36, 42]. Before industry decides to implement such aggressively partitioned 3D architectures, however, the first step will likely be the stacking of additional memory on top of a commodity processor to increase its last-level cache (LLC) capacity [3].

Additional LLC capacity can store larger working sets that can greatly improve performance by increasing cache hit rates and reducing contention for the off-chip bus and memory controller queue entries. The reduction in off-chip traffic also translates into power savings by reducing activity in the main memory DRAM chips as well as related I/O drivers. In this paper, we observe that using 3D integration to augment cache capacity does of course increase performance for memory-intensive workloads. Stopping at this obvious design point, however, leaves significant performance opportunities uncovered. In particular, we study the use of 3D-stacked DRAM to implement a large last-level cache (LLC), and we show how to exploit the physical organization of DRAMs to further enhance the performance of multi-core systems.

We propose a novel cache replacement policy that has the properties of providing performance isolation between cores and reduces the lifetimes of dead cache lines, while maintaining a low-complexity implementation that is compatible with the clock-based pseudo-LRU approximation algorithm used in real processors [40]. We organize each cache set into multiple queues or FIFO structures (one per core) that act as "filters" for different types of cache access patterns with limited reuse. We then use a generalization of the Set-Dueling approach to dynamically adapt the sizes of the queues (or whether to use them at all) as well as how lines may be advanced between different queues [14, 34]. The physical implementation of this cache organization relies on the DRAM's row-buffer architecture which makes our approach unique to 3D-stacked DRAM caches. The resulting cache management scheme is simple, yet flexible, and it provides 29.1% additional performance on top of the benefits of simplistic 3D-stacking of DRAM as a large but conventional LLC.

MICRO'09, December 12–16, 2009, New York, NY, USA.

2. 3D-STACKED CACHES

2.1 3D Integration and Cache Organizations

Three-dimensional integration technology has received an increasing amount of attention in the computer architecture community in recent years. 3D technology vertically stacks multiple layers of silicon connected by high-density, low latency vias to provide high interconnect bandwidth between the layers. Initial 3D microprocessor products will likely only employ simple 3D structures, with 3D-stacked caches being one of the most obvious choices. In this section, we review the most related work on using 3D caches to increase a system's last-level cache (LLC) capacity. A more comprehensive overview of 3D integration technology is not included in this paper as it has already been well described in many earlier 3D computer architecture papers [3, 24].

Black et al. examined several different 3D cache organizations to increase LLC capacity [3]. Figure 1 shows their 3D configurations, along with a baseline 2D organization (a). The first 3D structure (b) dedicates the entire second layer to an SRAM cache; assuming that the LLC occupies 50% of the area in the baseline 2D version, this 3D-stacked SRAM configuration provides 3× the capacity. The second 3D structure (c) uses DRAM on the second layer; Black et al. cite an 8× advantage in DRAM density over SRAM, and so this configuration provides 16× the capacity (since the DRAM layer is twice as large as the footprint of the original 2D cache). At a first-level of approximation, chip cost is directly proportional to total

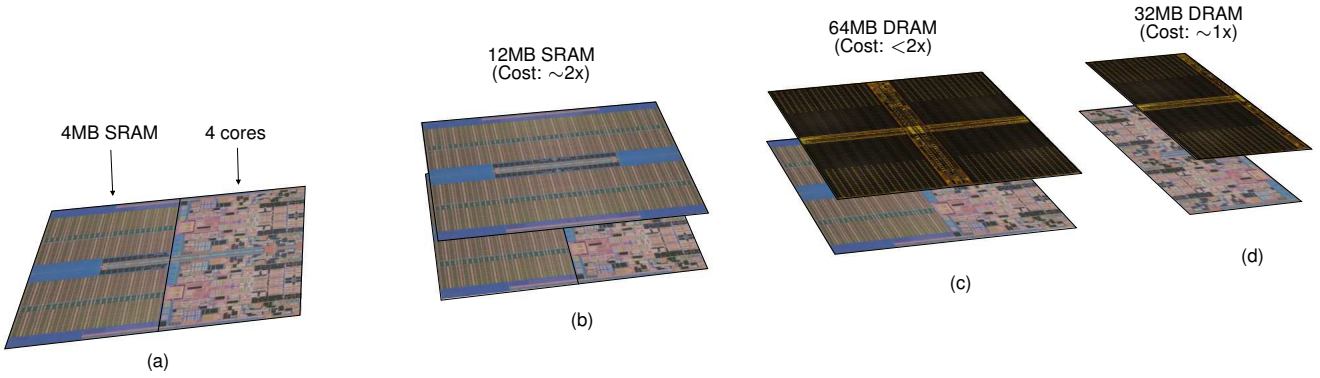


Figure 1: Different 3D-stacked cache configurations. (a) Baseline quad-core processor with 4MB SRAM LLC, (b) with +8MB additional SRAM cache for 12MB total at twice the silicon area, (c) with +64MB DRAM cache at twice the silicon area, (d) with +32MB DRAM cache with similar area as the baseline.

silicon area, and so a major problem with the approaches shown in Figure 1(b) and (c) is that they both approximately double the area, and hence the cost, of the chip. The last configuration (d) is an approximately area-neutral configuration where the footprint of the bottom layer is reduced by removing the SRAM cache and replacing it with a similarly sized 3D-stacked DRAM cache that provides 8× the capacity compared to the baseline. We believe that comparing this 3D-cache organization to the 2D case is fairer as the cost differential between the two should be relatively smaller.

Over time as the fabrication technology, design tools, test methodologies and the rest of the 3D ecosystem develop, computer architectures may move to more sophisticated 3D microarchitectures. Before we get there, however, being able to demonstrate that even the first generation of high-performance processors employing simple 3D-stacked memories provides a good return on investment will be critical to start the adoption process. This will in turn motivate the continued investment necessary to fully develop a complete 3D manufacturing environment that must be in place for the more interesting 3D computer architectures to reach eventual commercial success.

Other Related Work

There has already been a significant amount of work on using 3D to improve various aspects of a processor's cache hierarchy. These target more advanced generations of 3D integration involving finer levels of 3D partitioning [31, 36, 42], massively parallel NUCA-based designs for 16, 32 or more cores [16, 18, 28, 30, 43], or the integration of many stacked layers [15, 20, 22, 25, 27]. Ghosh and Lee examined some of the thermal challenges associated with stacking DRAM on top of a processor (in particular, the higher temperature requires a faster refresh rate), but this work was focused on the energy/thermal issues rather than the performance potential of 3D-stacked DRAM [9]. These are all certainly interesting proposals, but the focus of this paper is on exploring the immediate opportunities of nearer-term (i.e., less aggressive) 3D organizations and demonstrating that even such relatively simple 3D topologies offer researchers new avenues of exploration.

2.2 Physical Organization of 3D DRAM Caches

A processor's cache is most frequently implemented as an SRAM array (or possibly multiple banks of arrays and/or sub-arrays). The interface to the SRAM array is straightforward, consisting of an address bus and a data bus (read/write signal not shown) as illustrated in Figure 2(a). Accessing any set of the array incurs the same latency as accessing any other set.

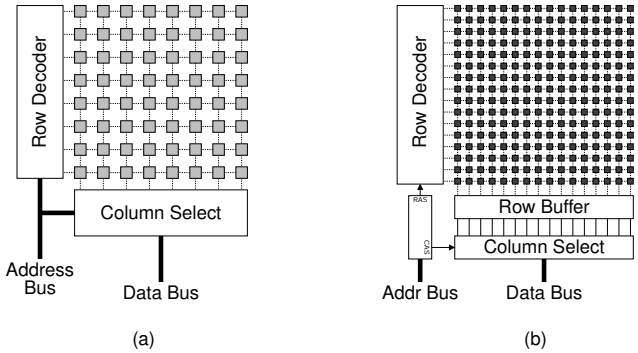


Figure 2: Simplified example organizations of (a) an SRAM array and (b) a DRAM array.

DRAM arrays have more complicated interfaces than SRAMs. Figure 2(b) shows the organization of one array. Reading a row from the DRAM array destroys the contents of that row. As a result, the row must be buffered in a *row buffer*. Row sizes (and consequently row-buffer sizes) are typically fairly large, from hundreds of bytes to a few kilobytes. Before reading from a different row in the array, the contents of the row buffer must be written back. Due to this organization, "reading" from the DRAM actually requires a sequence of multiple commands. In this case, a request to load a particular row into the row buffer (i.e., the row access), a second request to read particular bits/bytes from the row buffer (i.e., the column access), and finally a command to write the row back into the DRAM array (called precharge). A write to the DRAM follows a similar sequence, except during the column access, the new data are written into the row buffer rather than directly into the array as is typically done for SRAMs. For a DRAM employing an open-page policy, a sequence of accesses to the same row only requires one row access in the beginning, and then can be followed by many reads and writes that all operate directly on the row buffer, concluded with a single precharge command. Some chipsets also support an auto-precharge operation that automatically performs the row buffer writeback following a read/write command.

When using off-chip main memory, each DRAM chip typically has a data bus with a restricted width due to pin limitations. With 3D integration, however, the die-to-die interconnects are dense (on the order of a few microns [3]) and so the same pin-limitations of off-chip DRAMs do not apply. Without the pin limitations, we can implement a wide bus so that, for example, an entire cache line can

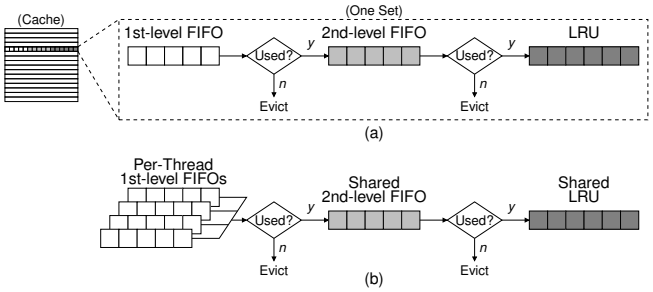


Figure 3: Logical organization of the basic multi-queue cache management scheme for (a) a single core and (b) multiple cores.

be transferred in a cycle. When using DRAM arrays as on-chip caches, it makes sense to map a complete cache set into a physical DRAM row. In this fashion, reading (or writing) a cache line only involves accessing a single row of a single array. Multiple arrays can be used to provide concurrent accesses similar to traditionally banked caches.

3. MULTI-QUEUE CACHE MANAGEMENT

3.1 Basic Multi-Queue Algorithm

Our proposed cache management scheme organizes the ways of a cache set into multiple FIFOs or queues as shown in Figure 3. Each queue entry corresponds to a single cache line. The union of all of the entries cover all of the lines in a single cache set. Each set in the cache is organized in this fashion. In particular, Figure 3(a) shows the organization for a single core. All cache lines are initially inserted into the first-level queue/FIFO (we use the terms queue and FIFO interchangeably in this context). Each queue entry has an associated u-bit (u=used) that is set to zero on entry. Any subsequent hit on this line sets the u-bit. For a queue of size Q, after Q more insertions, the original line leaves the queue. If its u-bit is still zero, then the cache immediately evicts the line. If the u-bit is one, then the line inserts itself into a second-level queue/FIFO.

The shared second-level queue behaves in a similar fashion as the first-level queue. Each entry also has a u-bit that is cleared on insertion, marked on a hit, and then used to either evict a dequeued line or to allow it to advance into the final region of the cache set.

The final region of the cache set uses a conventional clock-based pseudo-LRU replacement policy [40]. On insertion, a line clears its u-bit and the clock pointer advances to the next line. On an eviction, the line pointed at by the clock is checked. If its u-bit is zero, then the line is evicted; if the u-bit is one, then the cache clears the u-bit, advances the clock pointer, and then repeats the same check on the next victim candidate. This repeats until the cache finds an entry with a zero u-bit.

Figure 3(b) illustrates one set of the multi-queue organization for a multi-core scenario. Each core has its own dedicated (private) first-level queue into which new lines are initially inserted. As lines leave the first-level queues, they may be inserted into a shared second-level queue. Finally, lines that leave the shared queue may be inserted into a separate pseudo-LRU managed area.

Intuition

Our proposed scheme targets three different types of cache behaviors. The first is for some programs, there are a significant number of cache lines that are inserted into the cache, and then never referenced again [14, 34]. This may occur when the DL1 provides all of the hits and effectively "filters" out the cache traffic from the LLC. The left portion of Figure 4(a) shows an example where such a line (■) is inserted into a cache. Assuming conventional LRU

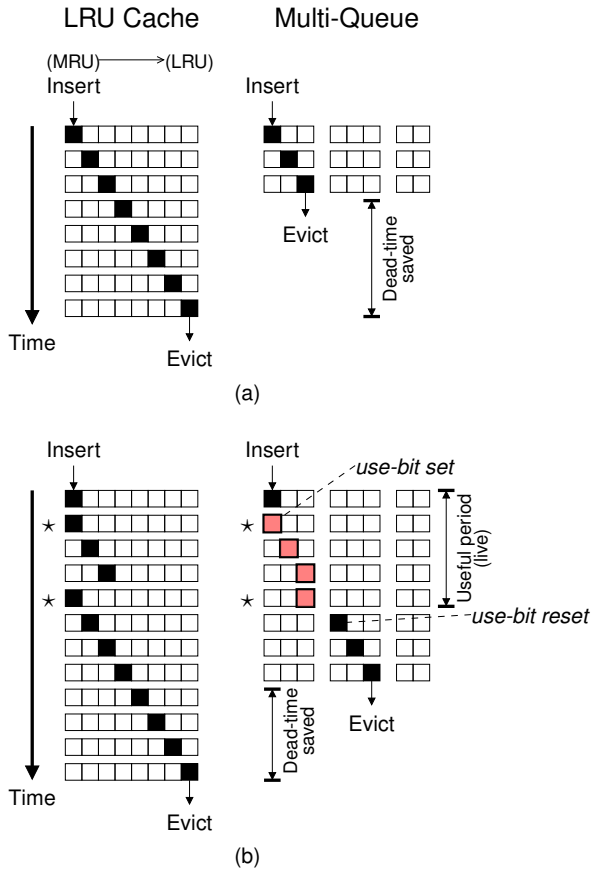


Figure 4: Examples illustrating (a) the insertion and subsequent eviction of a line with no reuse, and (b) a line with an initial burst of usage. A ⋆ indicates an access resulting in a cache hit.

replacement and a w-way cache, after w more insertions (insertions shown in the figure by the original line ■ moving toward the LRU position), the original line finally gets evicted. This line consumes valuable cache capacity while not providing any additional hits. The right portion of Figure 4(a) shows that in our multi-queue cache organization, such a no-reuse line is initially inserted in the first-level queue. After only $Q < w$ insertions (for a Q-entry first-level queue), the line leaves the first-level queue. Since the line was not reused, its u-bit is still zero, and therefore the cache directly evicts this line. The no-reuse line's residency time can be greatly reduced, thereby increasing the overall *efficiency* of the cache [4].

The second cache behavior targeted by our multi-queue technique is related to the phenomenon of *cache bursts* [21]. Figure 4(b) shows a cache line with temporal locality as illustrated by several hits (⋆) early in the timeline, but then after this initial *burst* of references the processor never reuses the line again prior to eviction. From the line's last touch until eviction, w more insertions must occur. In our multi-queue approach, the first-level queue should be sized such that the flurry of accesses finishes before Q insertions have been observed. In this scenario, the burst of reuses would have set the line's u-bit to one, and so it may advance into the second-level queue where its new u-bit is reset to zero again. Since the burst of accesses has ended, the line is not re-referenced before the cache dequeues the line from the second-level queue, at which point the line is evicted.

The last cache behavior, which works in conjunction with the previous two, is performance isolation/protection between the dif-

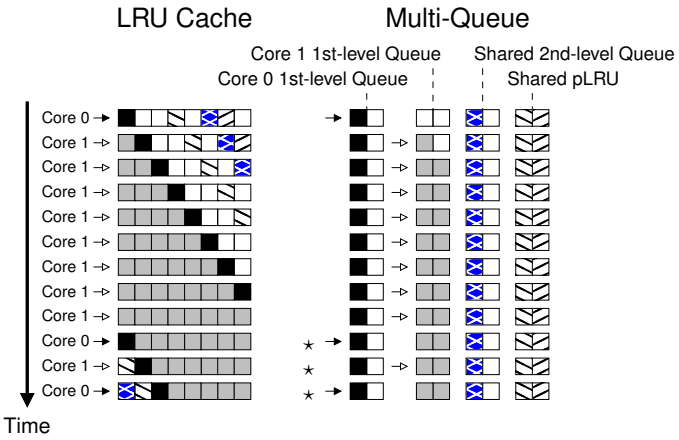


Figure 5: (*left*) Two cores where Core 0 reuses its lines and Core 1 does not, but Core 1 accesses the cache at a much faster rate. (*right*) The same scenario when Core 0 and 1 have their traffic isolated using multiple queues.

ferent cores. A core with a high access rate can quickly evict the cache lines of another core from the shared cache. The left portion of Figure 5 illustrates a simple two-core example with an LRU-managed cache. The cache initially contains some useful items (the cache lines with striped patterns in Figure 5), and Core 0 inserts a line ■. Core 1 then inserts a large number of cache lines with no reuse in rapid succession, thereby causing Core 0's line, along with any other useful lines, to get evicted from the cache. Subsequent accesses to these more useful lines all result in misses (last three rows of the example).

Our multi-queue approach isolates traffic from the different cores into their separate first-level queues. The right portion of Figure 5 shows how Core 0's line ■ is inserted into Core 0's own first-level queue, and therefore is subsequently protected from the onslaught of Core 1's stream of requests. Furthermore, since Core 1's lines show no reuse, then these lines are evicted as soon as they dequeue from Core 1's respective first-level queue. Other cache lines with proven reuse (those with the striped patterns in the figure) are maintained in the cache in the second-level queue or the pLRU region; this also shields these lines from Core 1's streaming behavior. By protecting these lines, additional cache hits ($\star$) are possible.

Our multi-queue organization also fits naturally with a cache hierarchy that enforces the inclusion property. This is particularly relevant as the recent Intel Core i7 implements such a policy [13]. In particular, a cache line in the LLC may appear to have few uses, but this may simply be a result of high hit rates in the L1/L2 caches. While the line may appear "dead" in the LLC, attempting to evict this line before the line is dead in the L1/L2 would cause the line to be invalidated from L1/L2, which would then immediately be followed by re-retrieving the line from main memory (since it was just evicted from the LLC). If the total capacity of a core's first-level queues (across all sets) is greater than the size of its L2 cache, then the L2 working set should mostly stay resident in the core's first-level queue. Any lines that advance past the first-level queue will also tend to experience far less contention because the most troublesome streaming/no-reuse access patterns from other cores have already been filtered out.

Related Work

Hallnor and Reinhardt's *generational replacement* (GR) scheme for large last-level caches has some similarities with our proposed technique [10]. In a fashion somewhat similar to our multiple queues, the GR approach divides the cache into multiple pools or generations. Based on usage, cache lines can be promoted or demoted between generations. A few key differences are that the multi-queue (MQ) approach is designed for a set-associative cache while GR targets a fully-associative cache (although GR can potentially be retargeted for highly-associative caches such as those considered in this paper), MQ handles multiple cores, and MQ more aggressively eliminates dead lines from the cache (e.g., a no-reuse line can be directly evicted from MQ's first-level queue, while GR demotes the line through two generations beyond the initial "fresh pool" before eviction). Furthermore, MQ's implementation complexity is significantly lower; GR implements its generations/pools using hardware doubly-linked lists along with timestamps both of which require non-trivial control logic to perform updates, comparisons, and movement between generations.

3.2 Implementation Issues

For all of the experiments in this paper, we assume that the baseline processor implements the clock-based pseudo-LRU replacement policy, referred to as just "clock" for brevity. The overhead for the clock policy is one u-bit per entry and a single counter that tracks the current clock position. For a w-way set associative cache, this adds up to only $w+\lceil \log_2 w \rceil$ bits per set. For our proposed multi-queue scheme, we do not need any additional u-bits because, whether part of a queue or part of the clock-managed region of the cache, each entry still only needs a single u-bit. The only additional overhead is one extra counter per queue to track the location of the queue head. Figure 6(a) shows the logical organization of the multiple queues for one cache set (in a four-core setting), and Figure 6(b) illustrates how these components are mapped into the data and tag arrays of the cache.

The multi-queue organization introduces an additional potential complication that is not present in conventional cache replacement policies for SRAM caches. The division of the cache lines into separate queues may occasionally require moving a line from one queue to the other. In a conventional SRAM array, physically shuffling lines around is inefficient in terms of both latency and power. Figure 7(a) shows an example where line X is moved from column 3 to column 7 and line Y is moved from column 5 to column 3. Line X must first be read out of the cache and buffered, which requires the latency and power of a regular SRAM access. Line X can then be written back into the SRAM array, requiring another full SRAM access. This process is repeated again to move line Y. Overall, four total SRAM accesses are required in this example. For a large last-level cache, each operation may take many cycles.

With a DRAM array, accessing any row requires loading the row into the row buffer and eventually writing the row back. Outside of any activity that occurs in the row buffer, the loading of the row buffer and the write-back/precharge operation are effectively a fixed cost. The row buffer itself comes equipped with a multiplexer for read operations and a demultiplexer for write operations, as shown in Figure 7(b). By simply providing the column addresses of the source and destination to the mux and demux, respectively, we propose to "loop back" the data bus to provide single-cycle move/shuffle operations. The same shuffling of lines X and Y would now only require one cycle each. Furthermore, the power required to manipulate data in the row buffer is much less than would be required for the SRAM-based cache which needs to precharge bitlines, power the sense amplifiers, etc. Note that this type of shuffle operation is not supported in current DRAMs; this is a new operation that would need to specifically be implemented for such a 3D-stacked DRAM cache.

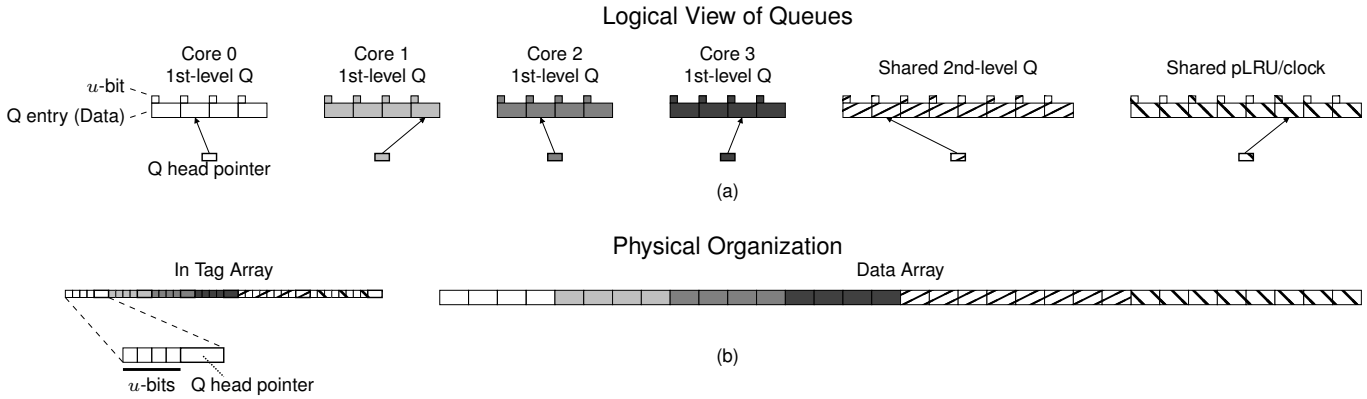


Figure 6: (a) The logical organization of the multiple queues, and (b) how the components are divided into the Tag Array and Data Array contents; the figure illustrates the cache lines and meta-data for only a single cache set.

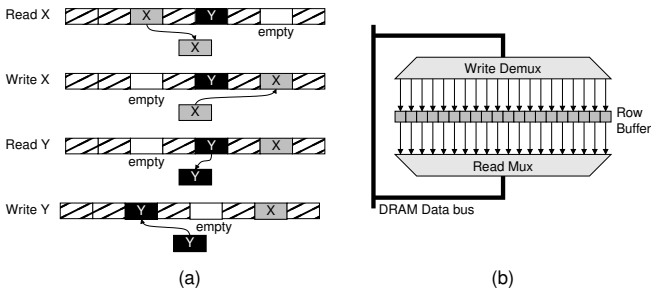


Figure 7: (a) An example of two shuffle operations on one set of a conventional SRAM, and (b) a simplified view of the read and write paths for a DRAM row buffer.

Pipeline Microarchitecture	
ISA	x86
Fetch	16 bytes/cycle
Decode/Issue/Commit	6/6/4 μops/cycle
ROB/RS sizes	128/36 μops
LDQ/STQ sizes	36 loads/24 stores
Clock	3.2 GHz
Cache/Memory Hierarchy	
DL1, IL1	32KB, 8-way, 2-cycle, 64B, private
DL2	256KB, 8-way, 5-cycle, 64B, private
LLC	4MB, 16-way, 9-cycle, 64B, shared, inclusive
next-line prefetcher	IL1, DL1, DL2
stride prefetcher	DL1, DL2, LLC
correlating prefetcher	LLC
ITLB/DTLB	512/1024-entry, 4-way
FSB	800 MHz (1.6GHz DDR for data transfers)
Main Memory	t_{RAS}=45ns, t_{WR}=15ns, t_{RCD}, t_{CAS}, t_{RP}=12.5ns

Table 1: Simulation parameters for the baseline processor and cache hierarchy.

Overall, our proposed multi-queue cache organization is unique to 3D-stacked DRAM-based cache structures.[1] The row-buffer architecture makes it practical to physically shuffle cache lines around (and this is only performed when a LLC eviction causes a line to move between queues, which does not always happen, for example when the u-bit is not set). One could argue that a row buffer could be added to a conventional SRAM cache. Such an approach would slow down the SRAM's common-case access pattern which is simply reading or writing a single cache line from a single set. With the DRAM, we have no choice but to read the data into a row buffer due to the fundamental physical operation of DRAM cells, but we turn this otherwise inconvenient organization to our advantage. As mentioned earlier, the size of typical DRAM row buffers is often a few kilobytes. This provides the opportunity for a large cache set with high set associativity.

4. EXPERIMENTAL EVALUATION

4.1 Simulation Methodology

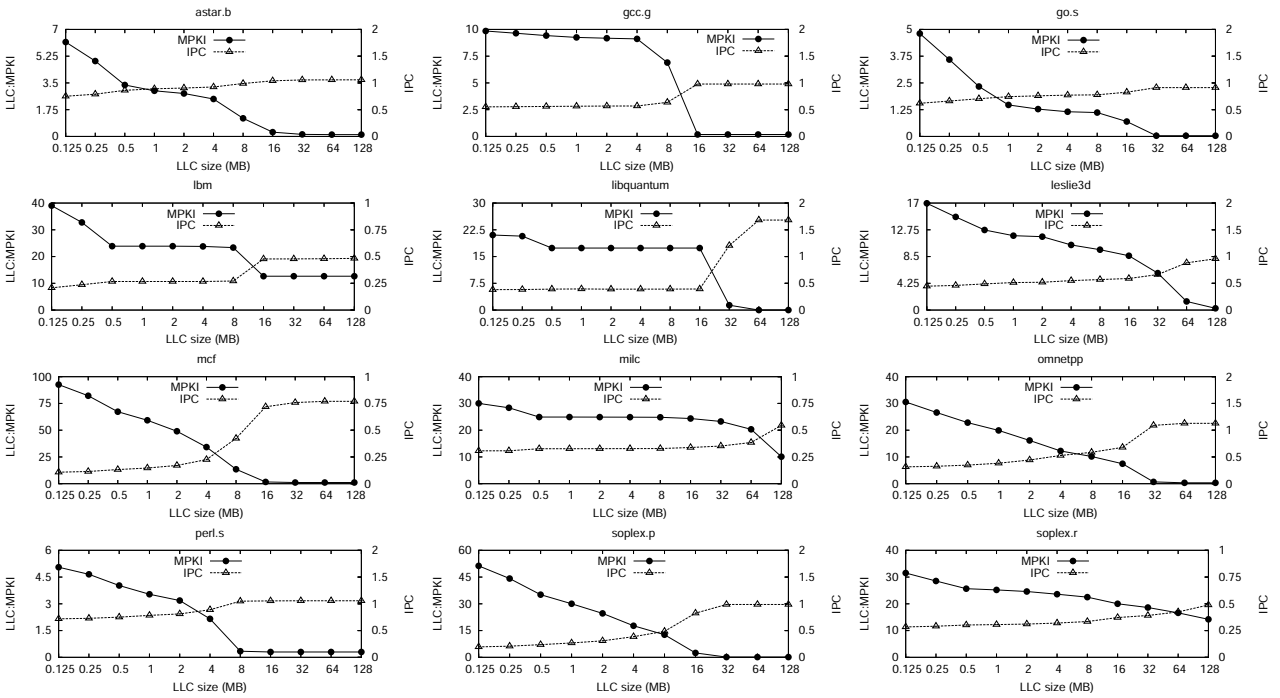
For our simulations, we use the x86 version of SimpleScalar (Zesto) [1, 23]. Table 1 lists the processor and cache hierarchy configurations used in this study. The baseline system is a quad-core processor with a shared, inclusive 4MB, 16-way cache using the clock replacement policy. The DL1, DL2 and LLC all use multiple prefetchers, with prefetch requests enqueued in a prefetch FIFO[2] that issues the prefetches when bus conditions are favorable [8]. We use the same estimates as Black et al. where the stacked DRAM provides eight times as much storage capacity as the equivalent area of SRAM [3]. To compute the latencies of the caches, we used CACTI v5.3 with a 45nm technology for the SRAM structures and 70nm for DRAM [41].

We focus on the area-neutral organization illustrated earlier in Figure 1(d) where the DRAM has the same footprint as the 4MB SRAM cache, and therefore provides 32MB worth of capacity. We evaluated a wide space of DRAM bank organizations (up to eight banks), set associativities (up to 128 way), and line sizes (up to 512 bytes) while accounting for the latency difference induced by the different physical organizations. The best configuration for a 32MB DRAM cache uses four banks, 64-way set associativity and 128-byte cache lines. Using CACTI's DRAM model, we use the following DRAM cache timing parameters: c_{RAS}, c_{RCD}=19, c_{CAS}, c_{WR}=17, and c_{PR}=14, where all of the timings are specified in *processor* clock cycles. Each DRAM bank allows a separate row to be open at the same time. We model an open-page policy which makes sense for facilitating multiple updates to the same row (e.g., when shuffling between queues is required). Our multi-queue configuration uses per-core first-level queues with Q=8 entries each, a second-level shared queue with S=12 entries, and the remaining 20 entries are managed with clock replacement.

For our quad-core system, our performance evaluations make use of multi-programmed workloads consisting of various memory-intensive programs from SPEC2006. The cache sensitivity results for each of the programs used are shown in Figure 8, providing both LLC MPKI (misses per thousand instructions) and IPC curves. The individual workloads are also listed in the figure along with thumb-

[1]The 3D-stacking is perhaps not strictly necessary. One could potentially apply this approach to a non-stacked eDRAM-based LLC, but we do not evaluate eDRAM in this paper and the results and conclusions would not likely be significantly different.

[2]This prefetch FIFO has nothing to do with the FIFOs in our multi-queue scheme.



Workload	Applications	Cache Sensitivity Curves
MIX01	*soplex.p*, astar.b, mcf, omnetpp	
MIX02	astar.b, go.s, libquantum, omnetpp	
MIX03	*lbm*, astar.b, omnetpp, perl.s	
MIX04	*leslie3d*, *lbm*, astar.b, omnetpp	
MIX05	*lbm*, astar.b, gcc.g, perl.s	
MIX06	*leslie3d*, *milc*, *soplex.r*, libquantum	

Workload	Applications	Cache Sensitivity Curves
MIX07	*soplex.p*, astar.b, libquantum, omnetpp	
MIX08	*lbm*, *milc*, gcc.g, omnetpp	
MIX09	*lbm*, *leslie3d*, *milc*, gcc.g	
MIX10	*milc*, *soplex.r*, astar.b, gcc.g	
MIX11	*milc*, astar.b, gcc.g, omnetpp	
MIX12	*leslie3d*, *soplex.r*, gcc.g, omnetpp	

Figure 8: SPEC2006 integer and floating point benchmarks used in our workloads. Plots show LLC MPKI and IPC rates for each program running on a single core for different LLC cache sizes. The individual workloads list the constituent applications (FP programs in italics) with thumbnails of the LLC MPKI cache sensitivity curves reproduced for convenience.

nails of the MPKI curves to make it easier to see the composition of memory patterns in each workload. For the multi-core performance simulations, we fast-forward each program by 500 million instructions while warming the caches, and then perform detailed simulation until *every* individual program within the workload has committed 250 million instructions. Simulation points were chosen with the SimPoint 3.2 toolset [11]. For each core, we only collect statistics up to this simulation limit, but the core continues executing so that it contends with the other cores for shared resources; this approach is consistent with other previous works on evaluating resource contention in multi-core processors [14, 34].

For most of our results, we simply report IPC throughput which is equal to $\sum_{i=1}^{n} IPC(i)$, where $IPC(i)$ is the IPC rate observed for Core$_i$ and there are n=4 four cores in total. After we have presented the final version of our multi-queue cache management technique, we also report the weighted speedup (also known as SMT speedup) $\sum_{i=1}^{n} \frac{IPC(i)}{IPC_{solo}(i)}$ [37], where $IPC_{solo}(i)$ is the IPC of Core$_i$ when it runs by itself without any contention from the other three cores, and the "fair speedup" metric $\frac{n}{\sum_{i=1}^{n} IPC_{solo}(i)/IPC(i)}$ which is often used as a fairness metric because the harmonic mean tends to emphasize/amplify its smallest inputs (i.e., those corresponding to slowdowns) [26].

4.2 Performance Results

Figure 9 shows baseline IPC throughput results for various 3D cache organizations (those shown in Figure 1), with the speedups normalized to the non-3D baseline 4MB SRAM cache. All configurations use clock replacement. All variants of the 3D-stacked caches demonstrate substantial performance gains over the baseline 4MB non-stacked cache. This in of itself is not too surprising since our workloads are memory intensive and stacking additional cache enables the cores to keep a larger fraction of their working sets on chip. This simply shows that when memory is tight, 3D caches increase capacity thereby leading to more performance, as has been shown in previous studies [3]. For the remainder of this paper, we will only focus on 32MB stacked-DRAM configurations.

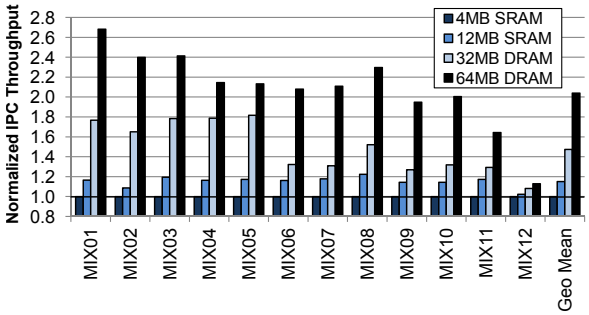


Figure 9: Baseline IPC throughput results for the configurations corresponding to Figure 1. Results are normalized to the non-3D 4MB SRAM case.

For the 32MB 3D-stacked DRAM configuration, we evaluate four different policies: the baseline clock replacement, the thread-aware dynamic insertion policy (TADIP) [14], utility-based cache partitioning (UCP) [34], and our multi-queue cache management. UCP uses 32 sets for dynamic set sampling and repartitions the cache every one million cycles. TADIP uses 32 leader sets per core with dynamic feedback (i.e., TADIP-F) and 9-bit PSEL counters. While there has been a considerable amount of past work on managing shared caches in multi-core processors [5, 6, 12, 17, 19, 33, 34, 35, 38, 39], we focused on UCP and TADIP as they are both directly compatible with clock-based replacement policy implementations.[3]

Figure 10 shows the IPC throughput results for different schemes to manage a 32MB 3D-stacked DRAM cache. All results are normalized to the clock configuration. The results for a somewhat idealized 64MB 3D-stacked DRAM cache configuration are also shown for comparison; the area and latency costs of the larger 64MB cache are optimistically ignored. For the different quad-core workloads, each core has different (and time-varying) cache capacity requirements. The UCP technique attempts to dynamically partition the cache by ways among the cores to minimize overall misses. For our workloads, UCP closes about half of the performance gap (+18.9% over clock) between the conventional clock-managed 32MB DRAM cache and the 64MB configuration (+38.4%). Our multi-queue (MQ) approach is able to provide slightly more performance than UCP (+23.6%). There are several workloads where UCP is still able to perform better than our MQ technique because UCP is able to more effectively *adapt* to dynamic changes in per-core memory requirements. We will introduce some adaptivity into our MQ scheme in Section 5.

In these results, TADIP does not perform as well as has been previously reported [14]. A significant contributor to this is the fact that our LLC enforces the inclusion property. As discussed earlier, the LLC only observes traffic that has been filtered through the L1/L2. So while a line may be heavily referenced in the L1/L2, it may still appear as a no-reuse line in the LLC. DIP-based policies tend to aggressively remove these lines leading to premature invalidation of the L1/L2 copies.[4] Also discussed earlier, so long as the aggregate capacity of a core's first-level queues is greater than the size of its DL2, then the problem of inclusion-induced invalidations is largely not a problem. For the 32MB DRAM LLC and a 256KB DL2 used in our simulated processor, dedicating $\frac{256KB}{32MB} = \frac{1}{128}$ of

[3]UCP simply needs one clock pointer per partition, and TADIP requires a simple change to the clock logic so that an "LRU" inserted line does *not* update the clock pointer so that it is immediately the next candidate for eviction.

[4]It might be possible to modify TADIP to account for inclusion effects, but such an extension to TADIP is outside the scope of this paper. We also verified that when inclusion is not enforced, TADIP does indeed perform better than UCP.

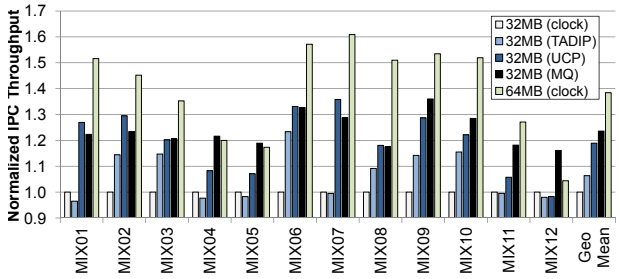


Figure 10: Normalized IPC throughput for various cache management schemes on a 32MB 3D-stacked DRAM cache, and a 64MB 3D-stacked DRAM cache for comparison. MQ stands for "multi-queue."

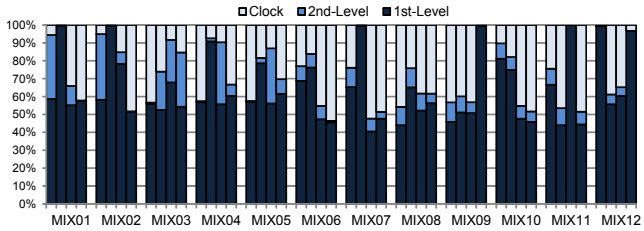


Figure 11: Distributions for where hits occur in the multi-queue managed LLC. Each core's distribution is normalized to its total number of LLC hits. The cores for each workload are not listed in any particular order.

the LLC to a core's first-level queue should be sufficient to avoid the inclusion problem. Since our set-associativity is 64-way and Q=8, we more than exceed this requirement. The reason that the queues are not made smaller is that lines cannot be evicted so quickly that the cache does not even have an opportunity to observe any reuses (which are required to set the line's u-bit to keep it resident).

The multi-level structure does play an important role in the behavior of our MQ organization. We simulated a reduced configuration where the first-level queues still have Q=8 entries each, but the second-level queue has been removed (S=0) so that lines with reuse are moved directly into the clock-managed region of the cache set. In this configuration, the speedup over the clock baseline is reduced to 15.4% which is below that of UCP.

For each workload, the constituent applications make use of the different levels of queues in different ways. Figure 11 shows the distribution of hits for each program in each quad-core workload. For example in MIX01, Core 0 (the leftmost of the four bars in the group) derives 58.6% of its LLC hits from lines in its first-level queue, 35.8% of its hits from the shared second-level queue, and the remaining 5.6% of its hits from the clock-managed region. Core 1 on the other hand, derives almost all of its hits from the first-level queue. Note that these hits advance to the second-level queue where they are not used again (otherwise the chart would report hits in this queue) and are evicted early instead of advancing to the main clock-managed region of the cache. In turn, this decreases the pressure on the clock-region allowing Cores 2 and 3 to make more efficient use of the remaining cache resources. The multi-queue organization naturally sorts the cache lines into different regions of the cache based on *proven* utility thereby improving the overall efficiency of the cache utilization.

5. ADAPTIVE MULTI-QUEUE (AMQ)

The results from the previous section showed that there are some workloads where UCP is able to achieve higher IPC throughput than our multi-queue approach. This is not entirely unexpected because

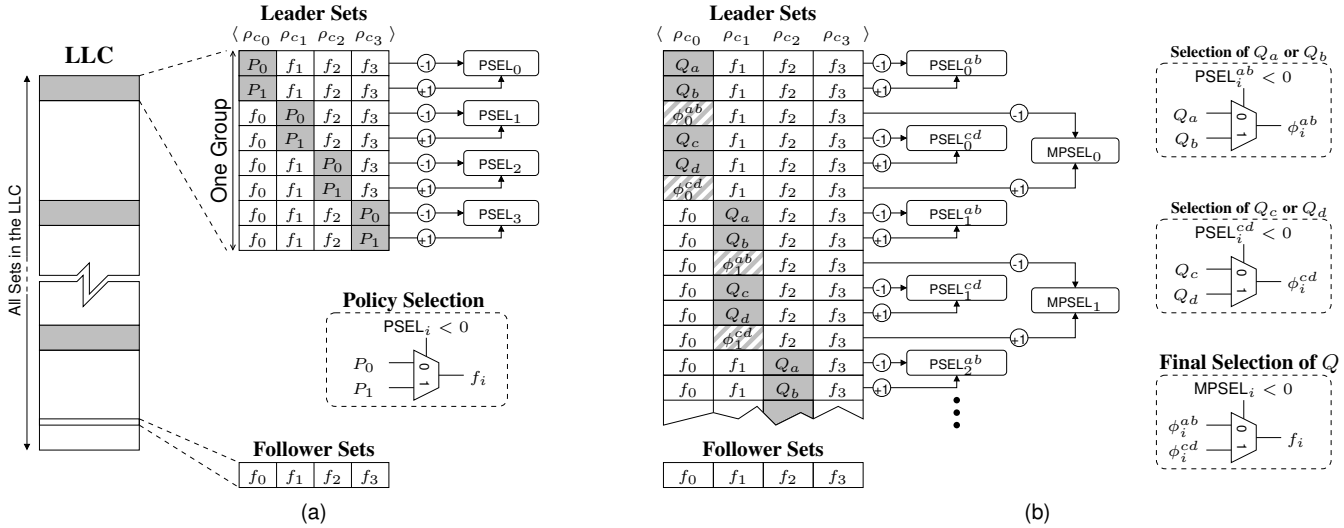


Figure 12: Different cache policy mechanisms: (a) multi-core, two-policy-per-core selection, and (b) multi-set dueling for multi-core, multiple-policy selection. Shading indicates fixed policies, and the partial shading represents partial followers.

we have thus far simply made use of statically-sized queues that in some cases may be over-provisioned for some cores causing dead lines to stay resident longer than they should, and in other cases under-provisioned, leading to the early eviction of lines when they will be re-referenced in the near future.

We propose an extension of our multi-queue scheme that dynamically adjusts queue sizes based on the needs of each core. While UCP uses sophisticated cache partitioning, implementing such optimization algorithms directly in hardware may be somewhat challenging for increasing numbers of cores and set associativities. Exploiting the large DRAM row buffer structure to implement highly-associative caches amplifies the difficulty of the partitioning problem. Instead of allowing arbitrary queue sizes, we take a simpler approach and restrict the queues to only a few choices, but we still need some mechanism to choose among the remaining options.

5.1 Multi-Set Dueling

Given $|\mathbb{Q}|$ possible selections for the size of *each* of the first-level queues and $|\mathbb{S}|$ selections for the second-level queue, there are $|\mathbb{Q}|^n \times |\mathbb{S}|$ possible unique configurations for an n-core system. Finding the best parameters among such a potentially large configuration space may be daunting. To tackle this problem, we propose a simple generalization of the *set-dueling* principal. Set dueling was proposed for the Dynamic Insertion Policy (DIP) to adaptively choose between the better of two different policies [34]. The idea is to dedicate a small, but statistically significant, number of cache sets where the sets follow *fixed* policies. A few such "leader sets" *always* manage their lines using a fixed policy P_0, and a few other leader sets always use policy P_1. Misses in leader sets following P_0 cause a *policy selection* (PSEL) counter to be decremented, and misses in leader sets following P_1 increment the PSEL counter. The PSEL counter effectively estimates which policy causes more misses based on the observed behaviors of these sampled leader sets. The remaining "follower" sets simply use the policy that should result in fewer misses as indicated by the PSEL counter.

In a multi-core context, each individual core may wish to follow a different policy. The TADIP multi-core extension of DIP introduced the use of per-core leader sets with per-core PSEL counters, as shown in Figure 12(a). In this example, each set is annotated with a policy vector $\langle \rho_{c_0}, \rho_{c_1}, \rho_{c_2}, \rho_{c_3} \rangle$, where c_i represents Core i, and ρ_{c_i} indicates the policy followed by Core i *for this set*. In each group of leader sets (each gray portion of the cache indicates one group), there is one leader set per policy, per core. For example, the first leader set in Figure 12(a) always applies policy P_0 to Core 0, while the second leader set always uses P_1 for Core 0. Note that the remaining cores (Core 1 through Core 3) do not use a fixed policy and simply follow the policy specified by their respective PSEL counters; i.e., the policy specified by f_i (f stands for follower). If a miss occurs in a set where Core 0 is forced to always follow P_0, then its counter PSEL_0 is decremented. Similarly, misses in sets where it is forced to always follow P_1 increment PSEL_0. For all remaining sets, including leader sets for other cores, cache decisions involving Core 0 will use the policy f_0 chosen by PSEL_0. The leader set structure is symmetric for all remaining cores. Each core can choose the policy that works the best for it, but the determination of what is "best" accounts for the policy selections of the other cores.

The set-dueling approaches for both DIP and TADIP assume that each core has only one of two policies to choose from. The selection of a queue size for our MQ approach is effectively a "policy" decision. For $|\mathbb{Q}| > 2$ we use a *multi-set* dueling approach that is similar to tournament branch predictors [29]. Consider the case for $\mathbb{Q} = \{Q_a, Q_b, Q_c, Q_d\}$ shown in Figure 12(b). For the first leader set, Core 0 always uses a first-level queue of size Q_a. For the next set, Core 0 always uses size Q_b. Misses in the former cause the counter PSEL_0^{ab} to be decremented, and misses in latter increment the counter. The third set follows the policy ϕ_0^{ab}, which sets Core 0's queue size (in this set) to Q_a or Q_b based on PSEL_0^{ab}. The symbol ϕ is used to indicate a partial follower; it is partial because the sizes Q_c and Q_d are not considered. Any miss in the set following ϕ_0^{ab} causes a "meta-policy" counter MPSEL_0 to be decremented. The next three sets are similar to the first three, except that one always sets Core 0's first-level queue size to Q_c, the next to Q_d, and the third to the best of these two (ϕ_0^{cd}). A miss in the set following policy ϕ_0^{cd} causes MPSEL_0 to be incremented. This policy selection process can be viewed as a single-elimination tournament, where the PSEL counters correspond to the "semi-final" rounds and the MPSEL counter represents the championship. Finally, all other follower sets set Core 0's first-level queue size according to policy f_0, which is determined by the tournament results of PSEL_0^{ab}, PSEL_0^{cd} and MPSEL_0. Figure 12(b) shows how the next six sets

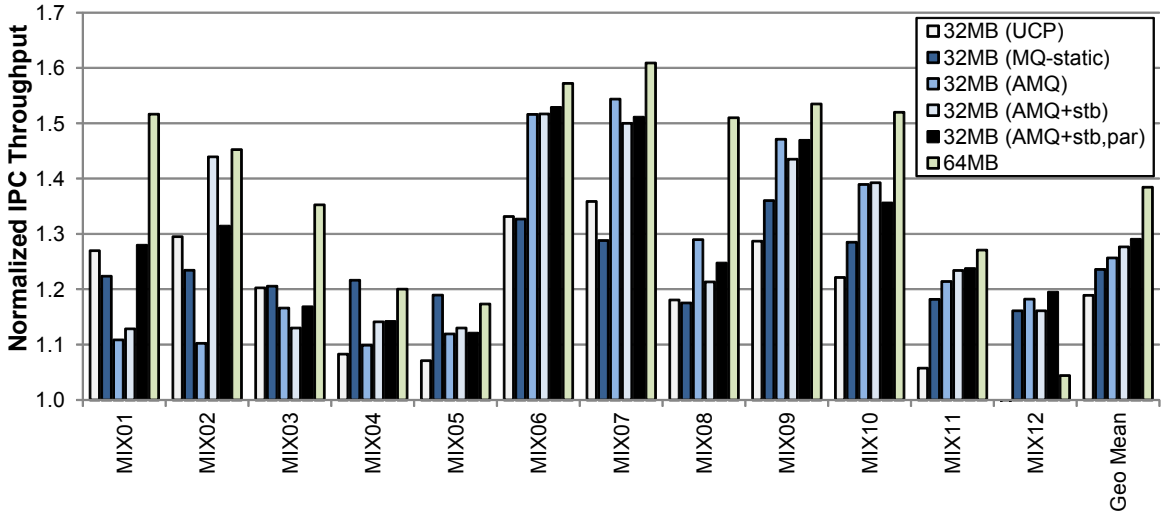


Figure 13: IPC throughput normalized to the quad-core configuration with 32MB 3D-stacked clock-managed DRAM cache. AMQ stands for "adaptive multi-queue", "+stb" is AMQ with the additional stability mechanisms, and "+par" also includes the dynamic pardon probability selection.

effectively repeat the process to determine the size of Core 1's first-level queues. This repeats again for Core 2 (not fully shown) and Core 3 (not shown at all). Likewise, another six leader sets (also not shown) determine the size of the shared second-level queue.

For our adaptive multi-queue (AMQ) approach, the first-level queues use one of four policies $\mathbb{Q} = \{0_s, 0_m, 4, 8\}$, and the second-level queue selects from one of four sizes $\mathbb{S} = \{0, 4, 8, 12\}$. For the first-level queues, there are actually two different choices for zero-sized queues. The policy 0_s means that the queue has no entries, and incoming cache lines should be inserted into the **s**econd-level queue. The policy 0_m is similar except that lines are inserted directly into the **m**ain clock-based region of the set.

Stability Issues

With so many policy and meta-policy decisions, there is some danger that the overall system can become unstable and rapidly switch through many different configurations and not actually converge on a good one. To combat this problem, we introduce two simple throttling mechanisms to slow down the rate of policy change. The first is a simple time delay where independent of the actual PSEL values, once a policy change has been made, no other changes may occur until at least δ cycles have elapsed, although the PSEL counters are still updated. The second mechanism adds hysteresis to the PSEL counters. When a PSEL counter goes negative, it must actually be decremented below $-h$ before the change in policy is invoked. Similarly, one must increment the counter above $+h$ to switch the policy back.

Occasional Lines with Long Reuse Distances

Due to the filtering mechanism used in the first and second-level queues, the queue size must match up reasonably well with the actual reuse distances for each core. The multi-set dueling approach will tend to select the queue size that most closely covers the *majority* of a core's cache line reuse patterns, but it is possible that there are still a significant number of lines that simply have reuse distances longer than the queue size, resulting in a situation where they will *always* be evicted early. We include a *pardon probability* which is similar in spirit to the Bimodal Insertion Policy from DIP [34], as well as statistical trace cache filtering [2]. If a line's u-bit is set, then it is always advanced to the next region of the cache. If the u-bit is zero, then with some probability p_{pardon}, the line is advanced anyway. We consider four possible pardon probabilities $\mathbb{P} = \{0, \frac{1}{32}, \frac{1}{8}, 1\}$ and use multi-set dueling to select p_{pardon} on a per-core basis.

5.2 Results

Figure 13 shows the IPC throughput results for our adaptive multi-queue (AMQ) cache management scheme. All of these results are normalized to the conventional 32MB 3D-stacked LLC with clock replacement. The figure shows the additional performance gains *beyond* the simple stacking of DRAM as a cache. Overall, UCP provides a 18.9% benefit over clock, and MQ-static provides 23.6%. The basic AMQ scheme delivers a 25.7% improvement, but the gains are not consistent across the workloads. For example, MIX01, MIX02 and MIX04 all exhibit significant performance reductions when adaptivity is introduced (although still always better than the baseline clock), and other workloads like MIX06 and MIX07 experience substantial improvements.

For most of the workloads where adaptivity hurts, the stability mechanisms (δ=32,768, h=32) provide some relief. In the cases where adaptivity is useful, the stability mechanisms sometimes helps, sometimes hurts, but neither by much in most situations. Overall, the stability mechanisms provide a small net benefit, increasing the geometric mean improvement to 27.6% over the baseline 32MB cache. Finally, the inclusion of the dynamic pardon probability selection provides another small boost, bringing the performance gain to 29.1%. It is important to note that the stability modifications and the pardoning mechanism are all simple to implement with low overhead. In return, the adaptivity is far more robust, providing higher performance than the static MQ in all but three of our workloads. Overall, the AMQ technique achieves 75.6% of the performance difference between the 32MB and 64MB clock-managed DRAM caches (recall that the 64MB configuration optimistically assumes the same timing parameters as the 32MB case).

For each workload, we performed additional tracing to illustrate AMQ's adaptations over time; for this experiment only, we collected trace information for only 250 million cycles, not including cache warming, and we collect one sample every one million cycles. Figure 14 shows for each benchmark of each workload, how much time each first-level queue spends configured at differ-

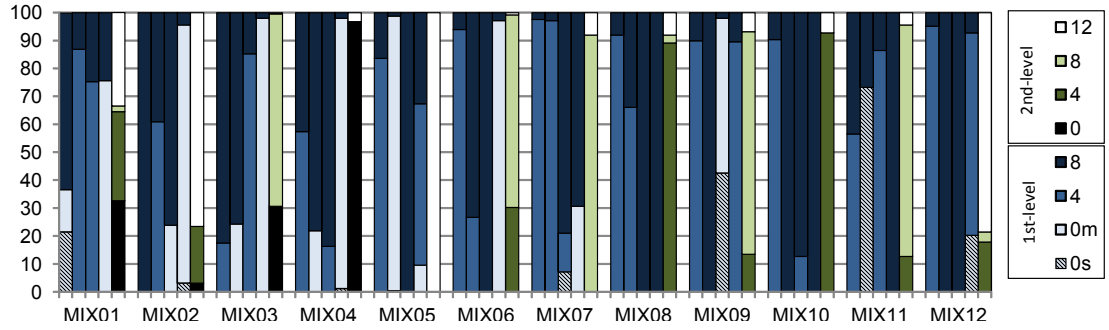


Figure 14: For abbreviated 250 million *cycle* traces, the percentage of time (samples) each queue was observed at a given size.

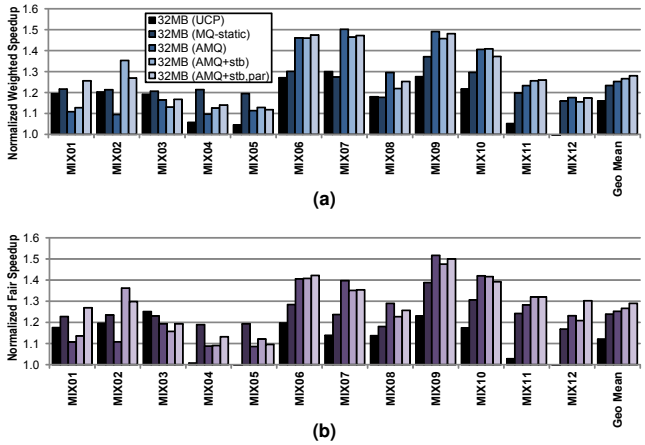


Figure 15: Performance of AMQ as measured by the (a) weighted/SMT speedup and (b) the fair speedup metrics.

ent sizes, as well as the amount of time that the shared second-level cache spends in its different sizes. The first four columns (dark shading on top) of each workload correspond to the per-core queues from Core 0 to Core 3. The 0s and 0m policies correspond to a zero-sized first-level queue with direct insertion into the second-level queue and main region, respectively. The fifth column (light shading on top) is for the shared queue. While a few individual programs find a queue size and then stick with it throughout the traced execution, there are others that clearly vary (i.e., adapt) over time.

In additional to overall IPC throughput, we also evaluated the weighted speedup and fair speedup metrics. These results are shown in Figure 15(a) and (b), respectively. Note that these metrics are only well-defined when IPC_{solo} is derived from the *same* baseline configuration, and so results not involving the 32MB DRAM cache configuration are not included. Overall, AMQ performs well on these metrics. In particular for the fair speedup metric, AMQ (with stability and pardoning) performs better than UCP on all but one workload (MIX03), and always better than clock, indicating that there are no significant concerns over fairness.

5.3 Implementation of AMQ

For the static version of the multi-queue cache, the physical mapping of the logical queues into the ways of the physical cache set is straightforward, as was illustrated in Figure 6. With resizing, however, a naive implementation may incur an excessive amount of data movement; even if all shuffling occurs in the row buffer, this still increases the latency of evicting a line and consumes unnecessary power.

For our implementation, we make use of a simple interleaved layout so that lines *never* need to be shuffled due to resizing (they still move when advancing between queues). Figure 16(a) shows a logical configuration with first-level queue sizes of 4, 8, 0, and 8, and a second level queue size of 12, and the main clock region occupies the remaining 32 ways (assuming a 64-way cache). Figure 16(b) shows the physical mapping of logical queue entries to physical ways. Core 0's first-level queue occupies ways 0, 4, 8, 12. Core 1 occupies ways 1, 5, 9, 13, 17, 21, 25 and 29. This mapping works for any power-of-two number of cores; Core i simply uses ways where the index modulo the number of cores is equal to i. Maintenance of the per-queue head pointers is also trivial. The interleaved view of Figure 16(b) makes it easy to see how increasing any individual first-level queue simply involves taking entries from the clock region, but this does not interfere with any of the other first-level queues. Likewise, shrinking a queue just causes the queue to return lines to the clock-based area of the cache. The bottom figure shows how all of these lines map sequentially into the sets of the cache.

The second-level queue can also change its size, and it is important to avoid having this queue collide with any of the ways used by the first-level queues (otherwise additional shuffling would be needed again). This is easily handled by physically locating the queue at the end of the set. The worst-case/maximum sizes of the queues are selected such that the queues can never grow into each other, and so shuffling due to queue resizing is guaranteed to be avoided. The remaining entries belonging to the clock-based region may end up residing in disjoint ways littered throughout the set. When choosing a victim from the clock region, any way belonging to one of the queues simply always asserts its u-bit and the clock algorithm will never select it for eviction.

There is also a question about what to do with valid cache lines that are effectively ejected from a queue when the queue size is reduced, as well as the reverse case where a line physically located in the main region suddenly becomes assimilated into a queue because the queue's size has increased. Instead of attempting to explicitly deal with such "illegal immigration," we simply ignore it. If a line is logically moved from one queue to another due to queue resizing, then so be it. This just means that on occasion a line here or there may move through the queue organization in a non-sequential order, but to explicitly handle this would require significantly more control logic complexity and likely a series of additional shuffle operations. Overall, our adaptive multi-queue policy maps cleanly into the physical cache set and the logic to implement the cache management policies are all straightforward.

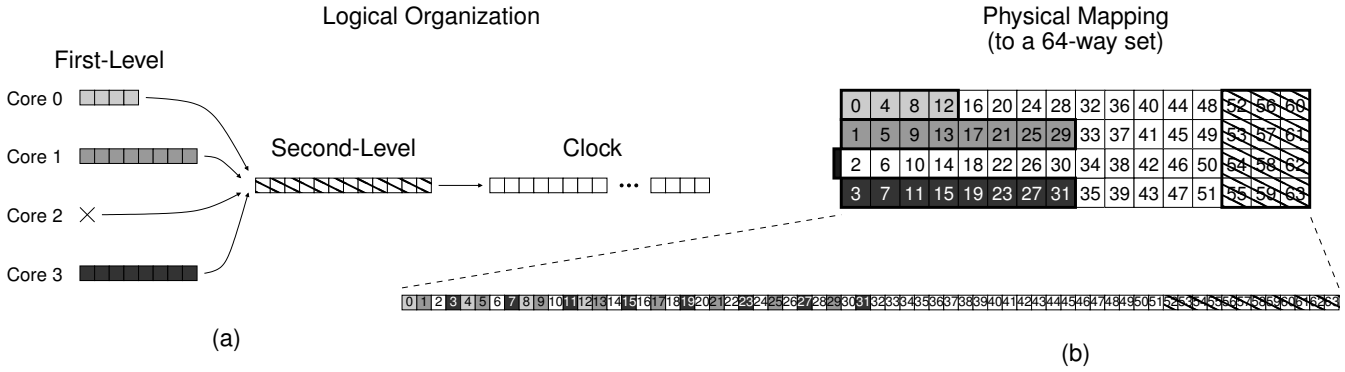


Figure 16: (a) A logical view of a MQ-cache with first-level queue sizes of 4, 8, 0 and 8, and a second-level queue of size 12, and (b) the mapping to physical ways in a cache set where the adjacent physical cache lines are sequentially numbered.

5.4 Sensitivity Analysis

Due to space concerns, the full results of our various sensitivity studies are not included here, but we instead summarize the results. One of the most important factors is the configuration for the underlying 3D-stacked DRAM cache. We evaluated many organizations, and UCP and our multi-queue approach perform better as the set associativity increases (at least until the DRAM cache latency becomes too large). In the case of UCP, the mechanism can provide finer provisioning of cache resources to the cores. For MQ, a certain minimum associativity is needed to at least support reasonably sized first-level queues.

We have already discussed the importance of having multiple layers of queues back in Section 4. We also evaluated the importance of adapting the sizes of the first and second-level queues as well as the pardon probability (i.e., those parameters controlled by set dueling). Not adapting the first-level queues results in an average of 1-2% performance reduction, depending on what size the queue is fixed at. Similarly, a fixed second-level queue results in a 1-3% drop, and not adjusting the pardon probability can result in a 2% drop. While the *average* performance impact may not be that large, removing adaptivity tends to increase the *variance* in the performance gains, and so the adaptivity is still desirable. We also tested different numbers of leader sets, but providing 32 leader set groups (where each group provides one leader set per core, per policy) as suggested in earlier work on dynamic set sampling was consistently the best choice [32].

6. CONCLUSIONS

In this paper, we have revisited the simple application of using 3D integration to stack a DRAM layer as a large last-level cache. This seemingly obvious approach delivers significant performance, and at first glance there is nothing particularly obvious for what else one can do with a large, stacked cache. We have shown that the physical architecture of the DRAM and its peripheral logic, which traditionally increases the complexity of the memory interface, actually provides us with an opportunity to derive benefit from these otherwise inconvenient structures. We believe that for many other potential applications of 3D integration, there will be several "obvious" solutions, but based on our experience with the 3D-stacked DRAM cache, we encourage researchers to continue digging to find additional ways that 3D can deliver more value beyond the obvious.

Acknowledgments

This project was funded by NSF grants CCF-0643500 and CCF-0702275; 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.

7. REFERENCES

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Micro Magazine*, pages 59–67, February 2002.

[2] M. Behar, A. Mendelson, and A. Kolodny. Trace Cache Sampling Filter. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 255–266, St. Louis, MO, USA, September 2005.

[3] B. Black, M. M. Annavaram, 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 International Symposium on Microarchitecture*, Orlando, FL, December 2006.

[4] D. Burger, J. R. Goodman, and A. Kägi. The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors. Technical Report 1261, University of Wisconsin, January 1995.

[5] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contenton on a Chip Multi-Processor Architecture. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, pages 340–351, San Francisco, CA, USA, February 2005.

[6] J. Chang and G. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *Proceedings of the 21st International Conference on Supercomputing*, pages 242–252, Seattle, WA, June 2007.

[7] J. Cong, A. Jagannathan, Y. Ma, G. Reinman, J. Wei, and Y. Zhang. An Automated Design Flow for 3D Microarchitecture Evaluation. In *Proceedings of the 11th Asia South Pacific Design Automation Conference*, pages 384–389, Yokohama, Japan, January 2006.

[8] J. Doweck. Inside Intel Core Microarchitecture and Smart Memory Access. White paper, Intel Corporation, 2006. http://download.intel.com/technology/architecture/sma.pdf.

[9] 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 International Symposium on Microarchitecture*, Chicago, IL, December 2007.

[10] E. G. Hallnor and S. K. Reinhardt. A Fully Associative Software-Managed Cache Design. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 107–116, June 2000.

[11] G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: Faster and More Flexible Program Analysis. In *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*, Madison, WI, USA, June 2005.

[12] L. R. Hsu, S. K. Reinhardt, R. R. Iyer, and S. Makineni. Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 13–22, Seattle, WA, USA, September 2006.

[13] Intel Corporation. First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem). *Intel White Paper*, March 2008.

[14] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. S. Jr., and J. Emer. Adaptive Insertion Policies for Managing Shared Caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Brasov, Romania, September 2007.

[15] 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 Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, San Jose, CA, USA, October 2006.

[16] J. Kim, C. Nicopoulos, D. Park, R. Das, Y. Xie, N. Vijaykrishnan, M. S. Yousif, and C. R. Das. A Novel Dimensionally-Decomposed Router for On-Chip Communication in 3D Architectures. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, USA, June 2007.

[17] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, Antibes Juan-les-Pins, France, September 2004.

[18] F. Li, C. Nicopoulos, T. Richardson, Y. Xie, V. Narayanan, and M. Kandemir. Design and Management of 3D Chip Multiprocessors Using Network-in-Memory. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 130–141, Boston, MA, USA, June 2006.

[19] J. Lin, Q. Lu, X. Ding, Z. Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In *Proceedings of the 14th International Symposium on High Performance Computer Architecture*, pages 367–378, Salt Lake City, UT, USA, February 2008.

[20] 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.

[21] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. In *Proceedings of the 41st International Symposium on Microarchitecture*, pages 222–233, Lake Como, Italy, November 2008.

[22] G. H. Loh. 3D-Stacked Memory Architectures for Multi-Core Processors. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 453–464, Beijing, China, June 2008.

[23] G. H. Loh, S. Subramaniam, and Y. Xie. Zesto: A Cycle-Level Simulator for Highly Detailed Microarchitecture Exploration. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, Boston, MA, USA, April 2009.

[24] G. H. Loh, Y. Xie, and B. Black. Processor Design in 3D Die-Stacking Technologies. *IEEE Micro Magazine*, 27(3), May–June 2007.

[25] 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 Conference*, pages 991–996, San Francisco, CA, USA, July 2006.

[26] K. Luo, J. Gummaraju, and M. Franklin. Balancing Throughput and Fairness in SMT Processors. In *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, pages 164–171, Tucson, AZ, USA, November 2001.

[27] N. Madan and R. Balasubramonian. Leveraging 3D Technology for Improved Reliability. In *Proceedings of the 40th International Symposium on Microarchitecture*, pages 223–235, Chicago, IL, December 2007.

[28] N. Madan, L. Zhao, N. Muralimanohar, A. Udipi, R. Balasubramonian, R. Iyer, S. Makineni, and D. Newell. Optimizing Communication and Capacity in a 3D Stacked Reconfigurable Cache Hierarchy. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, pages 262–274, Raleigh, NC, USA, February 2009.

[29] S. McFarling. Combining Branch Predictors. TN 36, Compaq Computer Corporation Western Research Laboratory, June 1993.

[30] D. Park, S. Eachempati, R. Das, A. K. Mishra, Y. Xie, N. Vijaykrishnan, and C. R. Das. MIRA: A Multi-Layered On-Chip Interconnect Router Architecture. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 251–261, Beijing, China, June 2008.

[31] K. Puttaswamy and G. H. Loh. Implementing Caches in a 3D Technology for High Performance Processors. In *Proceedings of the International Conference on Computer Design*, San Jose, CA, USA, October 2005.

[32] M. K. Qureshi, , D. Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-Aware Cache Replacement. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 167–178, Boston, MA, USA, June 2006.

[33] M. K. Qureshi. Dynamic Spill-Accept for Scalable High-Performance Caching in CMPs. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, pages 45–54, Raleigh, NC, USA, February 2009.

[34] 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 International Symposium on Microarchitecture*, pages 423–432, Orlando, FL, December 2006.

[35] N. Rafique, W.-T. Lin, and M. Thottethodi. Architectural Support for Operating System-Driven CMP Cache Management. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 2–12, Seattle, WA, USA, September 2006.

[36] P. Reed, G. Yeung, and B. Black. Design Aspects of a Microprocessor Data Cache using 3D Die Interconnect Technology. In *Proceedings of the International Conference on Integrated Circuit Design and Technology*, pages 15–18, Austin, TX, USA, May 2005.

[37] A. E. Snavely and D. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Machine. In *Proceedings of the 9th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, Cambridge, MA, USA, November 2000.

[38] H. S. Stone, J. Tuerk, and J. L. Wolf. Optimal Paritioning of Cache Memory. *IEEE Transactions on Computers*, 41(9):1054–1068, September 1992.

[39] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *Journal of Supercomputing*, 28(1):7–26, 2004.

[40] Sun Microsystems, Inc. UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007. Order Number: 950-5556-02, September 2007.

[41] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. HPL 2008/20, HP Labs, April 2008.

[42] Y.-F. Tsai, Y. Xie, N. Vijaykrishnan, and M. J. Irwin. Three-Dimensional Cache Design Using 3DCacti. In *Proceedings of the International Conference on Computer Design*, San Jose, CA, USA, October 2005.

[43] Y. Xu, Y. Du, B. Zhao, X. Zhou, Y. Zhang, and J. Yang. A Low-Radix and Low-Diameter 3D Interconnection Network Design. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, pages 30–42, Raleigh, NC, USA, February 2009.