Thread Row Buffers: Improving Memory Performance Isolation and Throughput in Multiprogrammed Environments

Enric Herrero, *Member*, *IEEE*, José González, *Member*, *IEEE*, Ramon Canal, *Member*, *IEEE*, and Dean Tullsen, *Fellow*, *IEEE*

Abstract—The widespread adoption of chip multiprocessors in recent years has increased the number of applications simultaneously accessing DRAM memories. Therefore, memory access patterns have also changed and this has reduced row buffer locality significantly, degrading performance and energy efficiency. Furthermore, concurrent execution of applications also has shown the need of performance isolation among threads in the memory controller to enforce a quality of service in virtualized environments. Existing DRAM memories, however, enforce a tradeoff between throughput and isolation. To solve these problems, this paper proposes the addition of Thread Row Buffers (TRBs) to DRAM memories. TRBs keep an active row per thread, thereby increasing DRAM efficiency by avoiding alternate accesses to a limited number of rows and allowing the implementation of a memory scheduler not bound to the throughput-isolation tradeoff. Thread Row Buffers with Service Partitioning (TRB-SP) increase the row hit-rate by 38 percent with respect to FR-FCFS and by 11 percent with respect to Cache DRAM. This, in turn, increases overall performance by 17 and 7 percent, respectively. TRB-SP is also able to reduce the standard deviation of the memory access time of an application by 40 percent over FR-FCFS, 31 percent over PAR-BS, and 42 percent over Cache DRAM.

Index Terms—Memory controllers, DRAM, thread row buffers

1 Introduction

During the last decade memories have greatly evolved in terms of capacity and integration but still remain one of the main limiting factors of current processor performance. This problem has been exacerbated with the introduction of chip multiprocessors, which require much larger amounts of data and have different access patterns. Such changes suggest that the memory hierarchy must be adapted to deal with the new requirements.

Due to the large size of memory arrays, memories use row buffers (typically of 8 kB) that store a whole page to allow faster reads and writes. This buffer needs to be updated every time a different row is read or written, consuming time and energy. If the row locality is high, the row buffer increases performance and reduces energy consumption. However, if the row hit rate is low, significant energy and time are lost replacing the data in the row buffer. Therefore, it is critical that memory systems make as much use as possible of row locality to both increase performance and reduce energy consumption.

Traditionally for uniprocessors, a First-Ready First-Come-First-Serve (FR-FCFS) policy [29], [43] implemented in the memory controller reached reasonable hit rates, using a simple reordering mechanism. The execution of several simultaneous applications, however, leads to different memory access patterns, which often alternate between a limited number of rows. This behavior significantly reduces the row hit rate for traditional configurations and, therefore, reduces overall performance.

Another concern introduced by multicore execution is that the usage of a shared resource like DRAM memory by different threads makes it necessary for the system to provide some kind of fairness or performance isolation control. Existing solutions [14], [25], [26], [27], however, rely on traditional DRAM memories that heavily penalize row misses and limit the adoption of more aggressive scheduling policies to avoid hurting throughput.

In this paper, we present Thread Row Buffers (TRBs), a simple mechanism to provide performance isolation in multiprogrammed environments while maintaining throughput. Thread Row Buffers maintain an active row for each thread simultaneously, avoiding the throughput-isolation tradeoff.

TRBs have three main advantages. The first one is that they improve the access latency in multiprogrammed environments. TRBs avoid the row alternation of traditional memories produced by several applications accessing different memory regions. This behavior can penalize

- *E. Herrero is with the Department of Arquitectura Computadors, Universitat Politècnica de Catalunya, and Intel Labs Barcelona, Edifici Nexus II, C/ Jordi Girona, 29 - 3A, 08034 Barcelona, Spain. E-mail: eherrero@ac.upc.edu.*
- *J. González is with the Visual Computing Group, Intel Labs Barcelona, Edifici Nexus II, C/ Jordi Girona, 29 - 3A, 08034 Barcelona, Spain. E-mail: pepe.gonzalez@intel.com.*
- *R. Canal is with the Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, c/Jordi Girona 1-3, Mòdul C6-107, 08034 Barcelona, Spain. E-mail: rcanal@ac.upc.edu.*
- *D. Tullsen is with the Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Drive, La Jolla, CA 92093-0114. E-mail: tullsen@cs.ucsd.edu.*

Manuscript received 29 Dec. 2011; revised 13 Apr. 2012; accepted 22 June 2012; published online 11 July 2012.
Recommended for acceptance by L. John.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2011-12-0991.
Digital Object Identifier no. 10.1109/TC.2012.173.

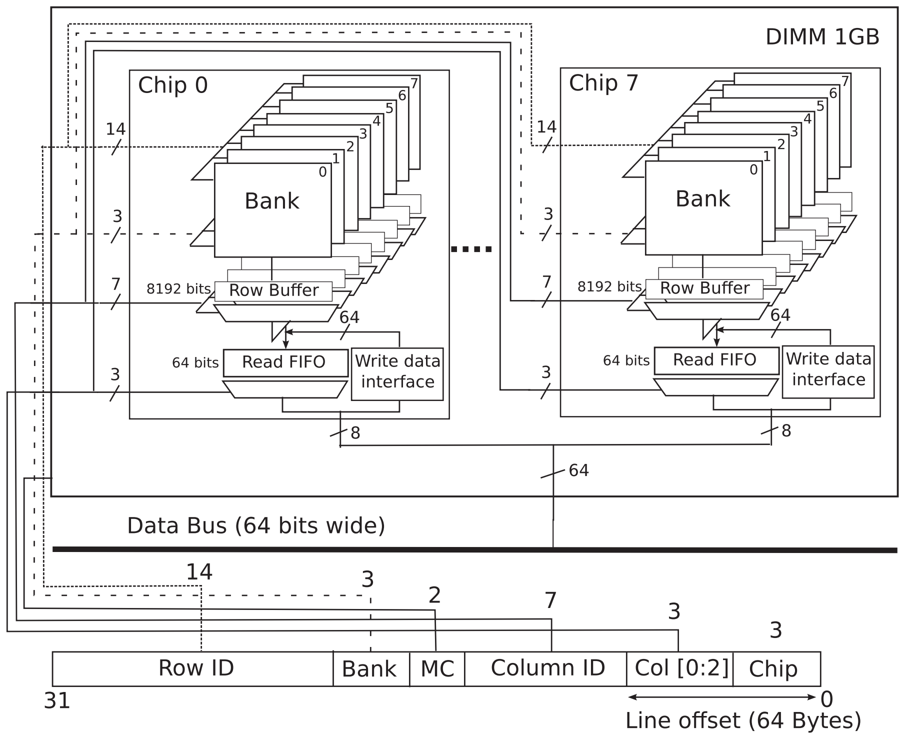


Fig. 1. Memory organization and mapping.

significantly performance in multiprogrammed environments and is avoided with TRBs.

The second is that they improve the performance isolation of the applications being executed. Quality of service and responsiveness are increasingly important in computing systems. TRBs reduce the interference of memory intensive applications over applications with more bursty or limited use of the memory with dedicated row buffers.

And finally, they allow the implementation of new policies in memory schedulers with lower impact in performance. Application prioritization has been traditionally limited by the performance penalization of lower row hit rates brought by different scheduling policies. TRBs avoid this problem, by having multiple active rows, and allow the implementation of scheduler prioritization mechanisms less focused on row hit rate and more on performance isolation or fairness.

As an example of memory scheduler, we present the addition of a service partitioning mechanism which is able to reduce the interference between applications and increase the row hit-rate by 38 percent. This results in an increase of the overall performance of 17 percent with respect to FR-FCFS and of 7 percent with respect to Cache DRAM. This performance improvement is explained by the avoidance of row alternation in TRBs that penalizes significantly row-hit rates for applications with low memory usage in configurations like Cache DRAM. In terms of performance isolation, our technique is able to reduce the standard deviation of an application latency by 40.4 percent over FR-FCFS, 31.4 percent over PAR-BS [26], and 42.1 percent over Cache DRAM [6] and unfairness by 18.8 percent over FR-FCFS, 5.6 percent over PAR-BS, and 28.5 percent over Cache DRAM.

This paper makes the following contributions:

- We study the memory access patterns of multicore architectures and show that they differ in important ways from traditional uniprocessor patterns. We show that multiprogrammed workloads reduce memory access locality significantly and that prefetchers play an important role in maintaining it but still leave room for improvement.
- We show that the inclusion of prefetchers changes the behavior of previous proposals due to a higher memory locality.
- We present Thread Row Buffers, an efficient technique to increase the row hit rate and provide performance isolation without hurting throughput. This technique is able to increase overall system performance and energy efficiency with low hardware overhead.
- We evaluate Thread Row Buffers with service partitioning and compare them to alternative techniques like Cache DRAM or Parallelism-Aware Batch Scheduling (PAR-BS), showing better throughput and performance isolation.

2 Background

2.1 Memory Organization

Fig. 1 shows the organization of a typical DRAM memory. These memories have several chips, each responsible for providing a part of the block simultaneously. Inside the chip, memory is organized in banks, each holding a part of the address space. Since memory parts are very big, and to reduce access latency, data are accessed inside the memory in rows (also called pages). Therefore, every bank has a row buffer and every time that an address is accessed the corresponding row is loaded to the row buffer. Subsequent accesses to addresses in the same row require a much smaller access time because they hit in the row buffer. Therefore, different situations can arise when accessing memory:

Row hit. In this case, the row is already in the row buffer so we can read it directly. Access latency will be the Column Access Strobe latency (T_{CL}), the time between column access and data return by the DRAM.

Row closed. There is no data in the row buffer. Access latency is the time required to load (*activate command*) the row and then read. Access latency will be the Row to Column command Delay (T_{RCD}), the delay between the row access command and the data ready at the row buffer, plus the read latency; $T_{RCD} + T_{CL}$.

Row conflict. In this case, we have a different row in the row buffer, and therefore, we need to write this row back (*precharge command*) and load the one we want to access before reading. Access latency will be the Row Precharge time (T_{RP}) plus the activate and read latency; $T_{RP} + T_{RCD} + T_{CL}$.

Since no data from one bank can be transferred during its row activation or row precharge, multiple banks are used. Therefore, when a row is activated in one bank a block can be read in another one and there is always data available to be transferred.

Address mapping to physical memory also has significant influence on the overall performance. In a general configuration suited for all kind of applications, it would be desirable to distribute memory accesses among all memory banks and also maximize the hit rate. Fig. 1 shows the address mapping used in this paper. In this mapping, Column ID is mapped to the least significant bits to keep consecutive addresses in the same row and maximize the hit rate. The next bits after the row are mapped to DIMMs and banks to spread requests among memory controllers. Finally, more significant bits are devoted to Row ID to reduce the row miss rate. This mapping is intended for open-page configurations (row is not precharged after being accessed). For closed-page configurations (row is always precharged after being accessed) adjacent lines are usually mapped to different banks to take advantage of the available bandwidth and because spatial locality is expected to be small between consecutive accesses [39].

2.2 Memory Controller

Memory controllers are tightly related to memories. They arbitrate between requests to different banks, arbitrate the data bus usage, and enforce the memory timing constraints.

Fig. 2 shows the structure of a memory controller. Requests are separated in different queues depending on the bank where the address is mapped. In these queues, accesses are reordered following the desired policy, usually a First-Ready First-Come-First-Serve policy [29], [43] that prioritizes accesses to active rows.

The second part of the memory controller is the bank arbiter. This part usually uses a simple round-robin arbitration policy. However, only idle banks can issue requests; therefore, it is necessary to keep track of bank states to know if they are performing a row activation or precharge or if a previous request is waiting to use the data bus.

Once the desired row is activated, requests are enqueued in the data bus queue. The data bus arbiter is a critical part of the memory controller because the limited bandwidth is often the bottleneck in current processors. Due to the delay between servicing a read request and having the data available a look-ahead queue is required to reserve the bus at the time the data is going to be in the Read FIFO. This arbiter also must take into account the 1 cycle delay incurred when requests change from read to write or vice versa.

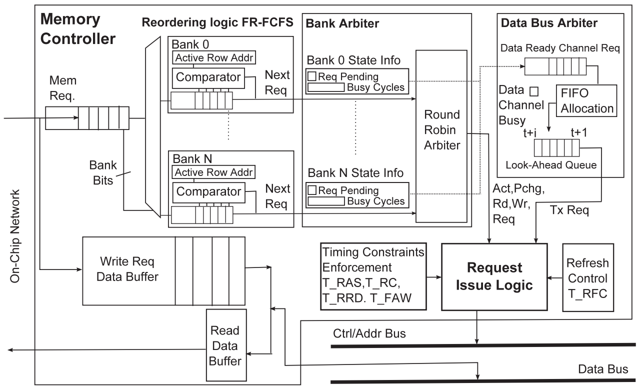


Fig. 2. Memory controller structure.

Another important function of the memory controller is the enforcement of timing constraints. Due to noise effects and energy and thermal limitations, memories are constrained by some parameters like T_{RAS}, T_{RC}, T_{RRD}, and T_{FAW} (see Table 2). These parameters ensure that the maximum currents and temperatures are not exceeded by imposing a maximum amount of activity over a period of time. It is also necessary to refresh the data to ensure no information is lost and this must be done every T_{RFC} cycles for a given region of memory. Therefore, one important part of the memory controller must include cycle and event counters that enforce these constraints regardless of the number of requests.

Finally, the request issue logic must select the request that is going to use the ctrl/addr bus, priority is given first to memory constraints, then the data bus arbiter, and finally the bank arbiter.

2.3 Influence of Prefetching

As we have seen from the memory organization, it is very important to maximize the hit rate to limit the number of row activations. This results in reduced access latency and energy consumption. In configurations running a single application, the usage of a FR-FCFS policy achieves a reasonably high row hit rate [29]. The chip multiprocessor consolidation, however, has brought new execution environments, where this solution is insufficient.

Fig. 3 shows a snapshot of the requests received in Bank 0 for a multiprogrammed execution when executing Gafort (SPEC OMP) with eight threads, four copies of 456.hmmer and four copies of 459.GemsFDTD (SPEC CPU). The first plot shows the row numbers of requests in arrival order and the second one the order in which these requests are finally issued. It can be seen that although FR-FCFS reorders some of the requests (e.g., Req 7 is issued before Req 6), we can see an alternation in the rows being accessed; leaving room for optimization. The request reordering mechanism is not able to reorder more requests because, as can be seen in the last plot, the number of requests waiting in the buffer queue is small. This small amount of requests is explained by the limited memory-level parallelism (MLP) of applications.

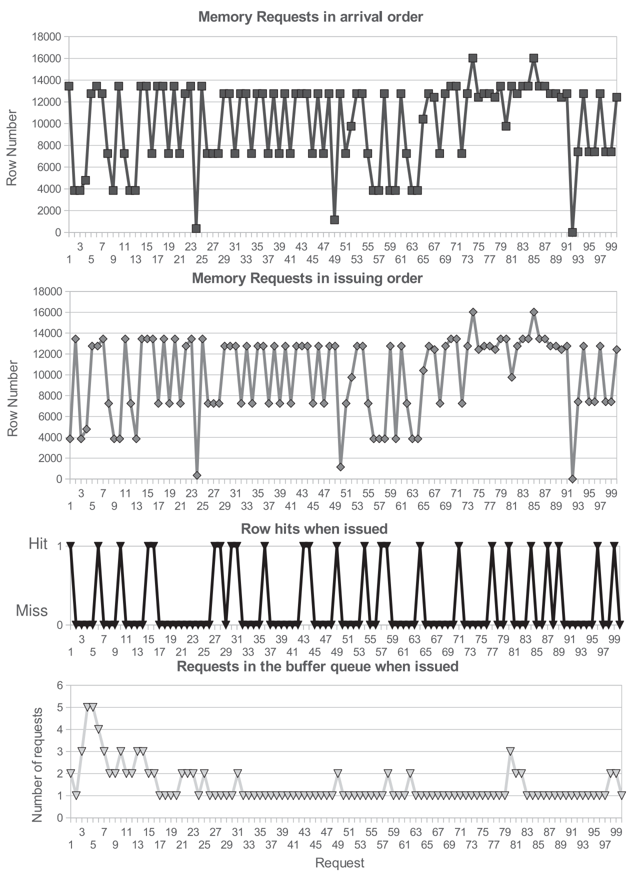


Fig. 3. Request behavior in Bank 0 in a multiprogrammed environment.

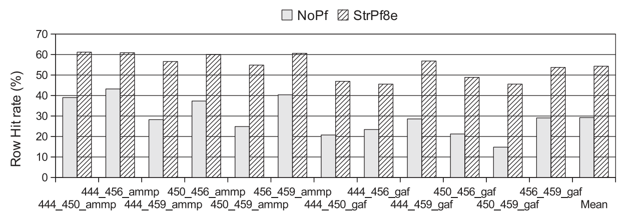


Fig. 4. Row hit rate with and without prefetch.

One way to improve row hits and reduce the row alternation is to use on-chip stream prefetchers [11] to group requests. Stream prefetchers increase the MLP and do not interfere with caches. Therefore, we have evaluated the influence of stream prefetchers in the memory controller.

Fig. 4[1] shows how the addition of 8-entry stream prefetchers in the memory controller can significantly improve the row hit rate, by generating extra memory parallelism and grouping requests. On average, row hit rate increases from 29.2 to 54.3 percent.

3 Related Work

There has been extensive work in optimizing memory controllers for chip multiprocessors. This work has been heavily focused on the optimization of bank and channel arbitration. The previously described First-Ready First-Come-First-Serve policy [29], [43] is an optimal solution from the memory throughput point of view and it is widely used. This technique prioritizes accesses to active rows, minimizing the amount of row activations. Some mechanism to avoid starvation must be included to ensure that under heavy load conditions all requests are serviced. Since only memory throughput is considered, this technique can present fairness and isolation problems.

Many techniques have appeared lately that take into consideration the interaction between threads. One of them is the Adaptive History-Based memory scheduler [9] that reorders requests making use of the command history and the set of available commands to minimize the request delay. This technique extends the FR-FCFS policy to also consider change of rank and port delays. In addition proposes a mechanism to enforce that the number of reads and writes matches the application behavior to avoid filling the reorder queues. The Self-Optimizing memory controller [10], on the other hand, uses reinforcement learning techniques to estimate the long term performance impact of each action and reorders requests to maximize long-term performance. And Yuan et al. [40] focus on reducing the complexity of DRAM schedulers.

Other work studies the interaction of prefetchers and memory controllers in multiprogrammed environments, and proposes coordinated control between prefetchers or memory aware prefetching. McKee and Wulf [22] show that prefetching blocks in streams can reduce the row alternation and increase the row hit rate. Ebrahimi et al. [3] show that prefetchers can cause a significant interference to other cores and, therefore, need to be dynamically adjusted in a coordinated way. Lin et al. [4] also study the influence of prefetchers and propose a memory controller with two priorities, high for demand requests and low for prefetch requests. This technique schedules prefetches only during idle cycles and do not harm the overall latency of demand requests. Always prioritizing demand requests over prefetch requests, however, can degrade performance in some cases because if these prefetches are useful they can reduce the number of demand misses. Srinath et al. [31] realized of this fact and proposed a prefetcher to dynamically adapt the aggressiveness of prefetchers but without considering the memory controller. Later, this idea was exported to the memory controller with the Prefetch-Aware DRAM Controller [17] Lee et al. [18], on the other hand, propose a Bank-Level Parallelism aware prefetcher and scheduler. Since banks can operate concurrently, the best way of granting a continuous amount of data to the bus is to maximize the bank parallelism of requests. The proposed prefetcher prioritizes requests to different banks over requests to the same bank. While prioritizing requests to different banks, it is possible to increase the memory-level parallelism, this prioritization also to reduce the row locality that determines the overall row hit rate. In uniprogrammed configurations, it may not affect significantly because no other requests arrive to the banks and the used rows remain active. In multiprogrammed environments, however, this configuration encourages row alternation and, therefore, a reduction of the row hit rate that greatly impacts in the overall performance and energy consumption of DRAMs.

Other work increases performance by a memory-aware management of the last-level on-chip cache [33] or by using

1. Combination of two SPECCPU (four copies of each) and one SPECOMP (with eight threads) benchmarks. Further details of the simulation environment can be found in Section 5.

a cache replacement-aware mapping [41]. Power consumption is an important factor and, therefore, several techniques have sought to reduce it [16], [34], [38], [42].

With the advent of chip multiprocessors, however, new problems arise. Performance isolation and fairness have become as important as throughput and several publications have appeared centered on solving these problems. Fair Queuing [27] uses Network Fair Queuing algorithms to grant fairness by prioritizing requests with earliest virtual finish-times. Rafique et al. [28] propose an improvement to this technique using start time fair queuing, treating each memory request as a unit of scheduling. Stall-Time Fair scheduling [25], on the other hand, provides quality of service by reordering requests to equalize the memory-related slowdown between threads. ATLAS [14] is more focused on throughput and reorders thread priorities based on the service they have attained previously, prioritizing the ones that have requested the least service. This technique is based on queuing theory which shows that when the job size distribution is exponential and the arrival process is Poisson, then the shortest-queue task assignment policy is optimal. Thread Cluster Memory (TCM) [15] scheduling also is focused on system throughput and divides threads into two separate clusters, latency-sensitive and bandwidth-sensitive. This scheduler is also based on the assumption that the system throughput benefits of prioritizing memory-nonintensive threads over memory-intensive ones. Therefore, this technique prioritizes always latency-sensitive threads. Within the latency-sensitive cluster, lower MPKI threads are prioritized following the same principle. Minimalist Open-Page [12], on the other hand, provides a tradeoff between throughput and fairness through the usage of a different mapping that allows a high enough number of page hits but does not penalize too much applications with low memory usage or bad locality. In addition, they also present a memory scheduler directed with processor-generated prefetch metadata.

And finally, Parallelism-Aware Batch Scheduling [26] is a technique that enforces fairness and performance by processing requests from a thread in parallel in the DRAM banks to reduce the memory-related stall-time experienced by the thread. One of the main problems when enforcing fairness in the memory scheduler is starvation avoidance. PAR-BS creates request batches to ensure that all requests within a batch are serviced before creating a new one. This allows more aggressive reordering techniques without starvation issues and provides some performance isolation. However, batching requests can also hurt performance because it is prioritized over achieving a higher row hit rate. The reordering policy implemented in PAR-BS is based on the creation of a thread rank. The thread ranking is computed giving higher priority to the thread with the lowest number of requests in the batch. This policy tries to increase the intrathread bank parallelism within a batch.

The main problem in all the existing reordering techniques is that the important influence of row buffer locality in the memory system throughput generates a difficult tradeoff between throughput and fairness or performance isolation.

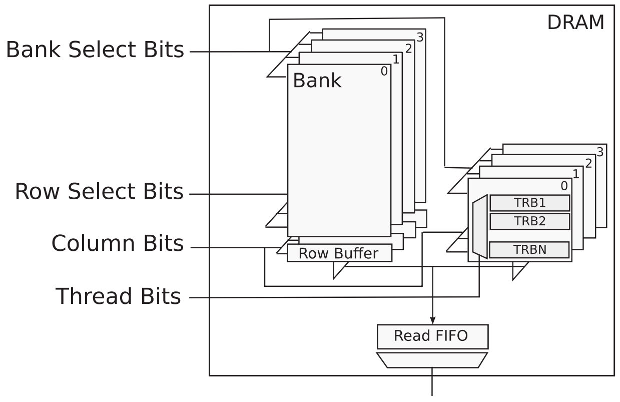


Fig. 5. DRAM with TRBs structure.

4 Thread Row Buffers

To solve the limitations of existing configurations, we propose to add a certain number of independent row buffers. These row buffers are extra storage added into the DRAM memory to keep several rows active at the same time. Unlike Cache DRAMs [6], [8], [19] which were thought as a normal cache, where more capacity means more performance, row buffers are based on the insight that with only one buffer per thread or application performance and isolation can be benefited significantly without having the overheads of Cache DRAM.

DRAM Cache proposed a unique cache for all the DRAM memory to store all the rows being activated in the different banks or in a specific bank. This approach, however, requires a significant amount of storage capacity and does not ensure that all applications make an efficient use of memory in a multiprogrammed environment. Furthermore, managing the extra storage as a cache implies to add a replacement mechanism in the memory controller. The replacement mechanism would require to store all the cache tags on-chip and the Least Recently Used (LRU) replacement mechanism, which is usually implemented with binary trees due to its implementation complexity. Since memory controllers are nowadays integrated on-die, this technique would involve a significant overhead in terms of area and energy consumption.

In this paper, we propose to assign a row to each thread, although row buffers could be also assigned to cores or applications depending on the hardware underneath. Thus, we talk about Thread Row Buffers. Every time that a row is activated in the DRAM, the data are copied to the entry of the requesting thread without requiring expensive replacement mechanisms. As it is going to be seen, this allocation is much more efficient than one based in least recently used data and requires a lower hardware overhead.

Our proposed mechanism only requires the reordering logic to store the row addresses of the TRBs. Since row buffers are assigned to threads, the replacement mechanism is trivial and only requires to add the thread ID in the request command. While TRBs change the traditional CPU-DRAM interface, we show in the remainder of the paper the benefits of moving in this new direction. Plus, in the context of 3D stacking [19] this separation is already nonexistent. Fig. 5 shows the structure of the DRAM memory with TRBs. The implementation of Thread Row Buffers requires extra

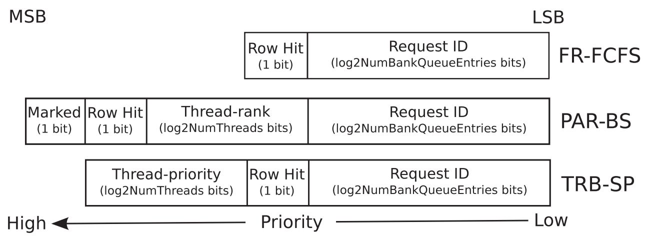

MSB
LSB
Row Hit (1 bit)
Request ID (log2NumBankQueueEntries bits)
FR-FCFS
Marked (1 bit)
Row Hit (1 bit)
Thread-rank (log2NumThreads bits)
Request ID (log2NumBankQueueEntries bits)
PAR-BS
Thread-priority (log2NumThreads bits)
Row Hit (1 bit)
Request ID (log2NumBankQueueEntries bits)
TRB-SP
High
Priority
Low

Fig. 6. Priority calculation for FR-FCFS, PAR-BS, and TRB-SP.

storage in each bank and the corresponding multiplexers to access them. Memory arrays would not need to be modified, thus maintaining the same high level of integration that characterizes current DRAM memories. Memory controllers store the tag of the active row to know if a precharge and activation is required. In the same way, the memory controller responsible of handling TRBs would have the addresses of the active rows, and therefore, if a row was already loaded in one TRB, it would not be loaded into another. This solution allows a more effective use of TRBs and avoids coherency problems.

The implementation of Thread Row Buffers in DRAM chips can be done in several ways, using sense amplifiers like existing row buffers [13] or using SRAM, like on-chip caches, mixed with the DRAM cells [1], [7]. Assuming the usage of 6T SRAM cells and using the Cacti [37] tool, we have estimated the area overhead of the Thread Row Buffers, which would be 2.04 percent of the total memory die (assuming a $56.7\ \mathrm{mm}^2$ die size [35]).

Our implementation assumes the same number of threads as TRB entries to allow performance isolation between threads. Nevertheless, TRBs can be used in other contexts with a different number of threads and TRB entries. The implementation of TRBs enables the memory controller to actively decide the replacement policy and management of the TRBs, unlike previous proposals (i.e., DRAM Cache), and to manage memory accesses in ways that have not traditionally been possible due to the limit of a single active row. These alternatives are left for future study. For instance, TRBs which could be switched off when not used to reduce the static power consumption or TRBs could be assigned on an application basis (not on a thread basis as we show in this paper).

4.1 Service Partitioning Scheduler

Traditional memory schedulers have been limited by the tradeoff of providing maximum throughput versus some kind of performance isolation. These schedulers reorder requests in the bank queues by selecting the request with highest priority. Priorities are calculated by concatenating different parameters depending on the type of scheduler. Fig. 6 shows an example of how the request priority is calculated for a traditional FR-FCFS scheduler and for the PAR-BS [26] scheduler. Since the memory row hit rate has a very high influence on the overall memory throughput, these techniques give it more importance, potentially reducing the final isolation. The usage of TRBs, on the other hand, has the advantage that the last row accessed by each thread is always going to be loaded in the TRBs, allowing the memory controller to apply any scheduling policy without hurting throughput.

As an example to show how easy it is to provide performance isolation with TRBs, we have prioritized requests with the method depicted in Fig. 6 as TRB-SP. To isolate the performance of the different threads, we have used a system that maintains a thread prioritization and uses these priorities to reorder requests, giving more importance to this ordering than the row hit information.

To calculate the thread priority for service partitioning, we have used a very simple mechanism that uses the most recent history and requires very little hardware. The working principle of our priority calculator is that every time that a request is issued, the owner thread changes its priority to the minimum (zero). The remaining threads that used to have lower priority than that thread increase their priority by one. Therefore, the thread that has been longest without issuing a request is going to have the maximum priority. The extra hardware required for priority calculation is very low and can be seen in Fig. 7. If needed, more complex scheduling mechanisms could be implemented in conjunction with TRBs without having to deal with a performance degradation due to a reduction of the row hit rate. As we will see, the proposed service partitioning mechanism is able to enforce a strong performance isolation without requiring a large hardware overhead.

Fig. 8 shows an example of how the different scheduling priorities work. We assume two threads (A and B), each of them always accessing the same row. For each configuration, we can see on the top the requests stored in the bank queue and on the bottom the requests issued to the DRAM. It is possible to see how FR-FCFS is able to finish very fast. This is because it prioritizes requests if there is a row hit, saving time in precharges and activations. This technique, however, is not fair and in some cases can stall a request from a different thread for a long time if multiple requests to the active row arrive. On the other hand, PAR-BS prioritizes fairness. The example shows how requests are grouped in a batch at the beginning and how requests from threads with fewer requests are given priority. Batching, however, does not allow new requests to bypass those within a batch, even if they are to the active row. Therefore, this reduces the row hit rate and the overall throughput. TRB-SP is able to combine the benefits of both techniques by providing the lowest completion time and, in addition, servicing requests in a fair way. We can see in Fig. 8 how requests are alternated and this does not penalize the hit rate due to the ability to store several active rows.

5 Experimental Setup

We have evaluated our proposed framework with Simics [20], a full-system execution-driven simulator extended with the GEMS [21] toolset that provides a detailed memory hierarchy model. A detailed memory controller and DRAM memory model has been added, including a power model based on activity counters and the Micron DDR3 energy requirements [24], to evaluate our proposal. We have assumed the usage of SRAM TRBs and, therefore, their static power and area has been evaluated with Cacti [37].

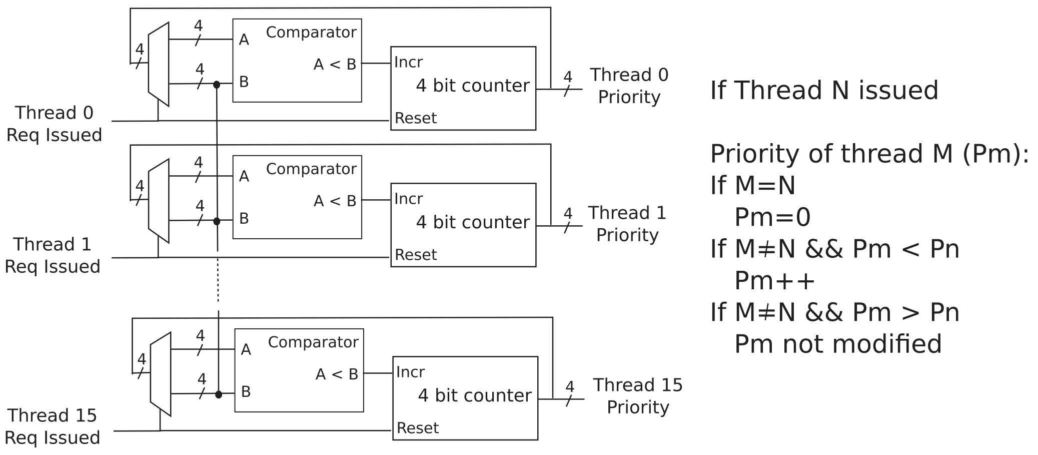


Fig. 7. Thread priority calculation hardware.

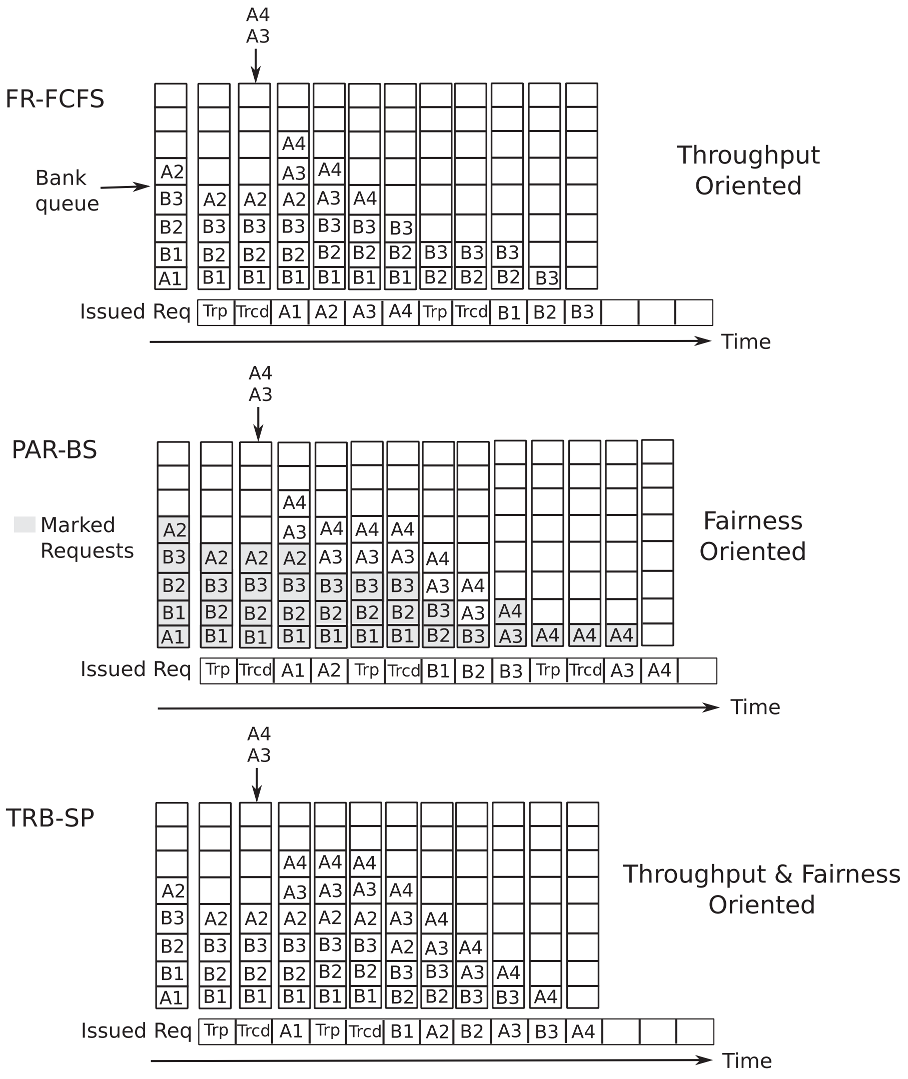

FR-FCFS
Bank queue
A4
A3
Throughput Oriented
Issued Req Trp Trcd A1 A2 A3 A4 Trp Trcd B1 B2 B3
Time
PAR-BS
Marked Requests
Fairness Oriented
Issued Req Trp Trcd A1 A2 Trp Trcd B1 B2 B3 Trp Trcd A3 A4
Time
TRB-SP
Throughput & Fairness Oriented
Issued Req Trp Trcd A1 Trp Trcd B1 A2 B2 A3 B3 A4
Time

Fig. 8. Scheduling example.

Table 1 shows the values for the most important configuration parameters.

Fig. 9 shows the evaluated CMP, using 16 processors and four memory controllers with a dedicated bus. The overall chip bandwidth is similar to recent processors like IBM Power7 [32] that has a sustained memory bandwidth of more than 100 GB/s with eight channels operating at 6.4 GHz. In all the tested configurations, two levels of cache are used; as well as a MOESI protocol to grant coherence between nodes. On-chip coherence is granted through a distributed directory [5]. Local and private L1 and L2 caches are used in every processor allowing sharing through the directory.

TABLE 1
Configuration Parameters

Parameter	Value
Number Processors	16
Instr Window/ROB	16/48 entries
Branch Predictor	YAGS
Technology	70 nm
Frequency	4 GHz
Voltage	1.1 V
Block size	64 bytes
L1 I/D Cache	16 KB, 4-way
L2 Cache	256 KB, 8-way
DCE Size	8192 entries
Prefetcher Streams	32 per MC
Stream entries	8
Network Type	Mesh with 2 VNC
Hop Latency	3 cycles
Link BW	16 bytes/cycle
Number Memory Controllers	4
Number of channels	4
Data bus width	64 bits
Off-chip BW	170.4 GB/s
Memory Capacity	4 x 1 Gb
Memory Clock Speed	667 MHz
Memory Speed Grade	-15
Number of memory banks	8 per rank
Number of memory ranks	1 per DIMM

As we have seen in Section 2.3, hit rate in CMPs is highly impacted by hardware prefetching; therefore, we have used stream prefetchers in all the evaluated configurations. A stream prefetcher like the one in IBM Power4 [11], [36] has been added to the simulator. As we will see, the addition of prefetching penalizes the PAR-BS configuration and our technique due to the increase in the memory-level parallelism. One of the main contributions of TRBs is the ability to deal with alternating row accesses that make the traditional FR-FCFS useless in multiprogrammed environments. We added the prefetchers to provide a more fair comparison against FR-FCFS, increasing the number of simultaneous requests to the same row.

To emulate the multiprogrammed execution environment of current chip multiprocessors, we have used a mix of single and multithreaded applications. Two multithreaded applications from the SPECOMP2001 workload set have been combined with single-threaded applications from the SPECCPU2006 workload set. All applications are simulated with reference input sets. In each combination, one SPECOMP2001 application is executed with eight threads in combination with two SPECCPU2006 applications, each of them running four copies with independent input data.

TABLE 2
Memory Timing Parameters [23]

Symbol	Parameter	Value
T_{RCD}	Row to Column command Delay	10 cycles (15ns)
T_{RP}	Row Precharge time	10 cycles (15ns)
T_{CL}	Column access strobe Latency	10 cycles (15ns)
T_{WR}	Write Recovery time	10 cycles (15ns)
T_{RAS}	Row Access Strobe	24 cycles (36ns)
T_{RC}	Row Cycle time	34 cycles (51ns)
T_{RRD}	Row activation to Row activation Delay	4 cycles (6ns)
T_{FAW}	Four bank Activation Window	20 cycles (30ns)
T_{RFC}	Refresh Cycle time	74 cycles (111ns)

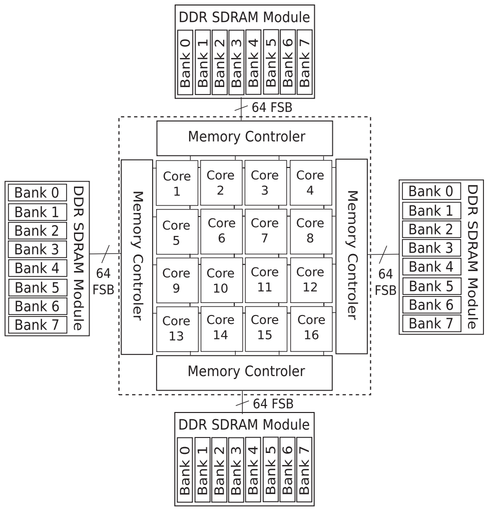


Fig. 9. Simulated CMP structure.

Fig. 10 shows with a shaded area the region with more representative applications. From that area, applications have been divided in four categories depending on the number of off-chip misses (MPKI) and the memory-level parallelism. Fig. 10 shows the division, which considers applications with more than 1 miss per 1,000 instructions to have a high number misses and to have a high MLP when the average number of simultaneous requests is higher than 8. Then, we have selected representative applications from each category to combine them and have all types of applications represented. The evaluated applications are 459.GemsFDTD, Gafort, 450.Soplex, 456.hmmer, 444.namd, and Ammp, which have been used in the evaluation of the proposed techniques.

These applications have been synchronized to the most significant execution regions and after warming up the caches during 1,000M instructions they have been executed for 50 million cycles per thread (which represents 400 million instructions on average). System throughput has been measured using Weighted speedup [30] and normalized performance with respect to FR-FCFS:

$$W.Speedup = \sum_{i=0}^{N} \left(\frac{IPC_i}{IPC_i^{FR-FCFS}} \right).$$

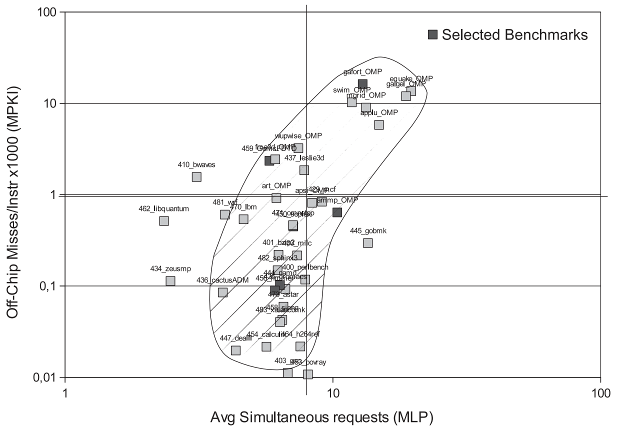


Fig. 10. Spec OMP2001 and CPU2006 classification.

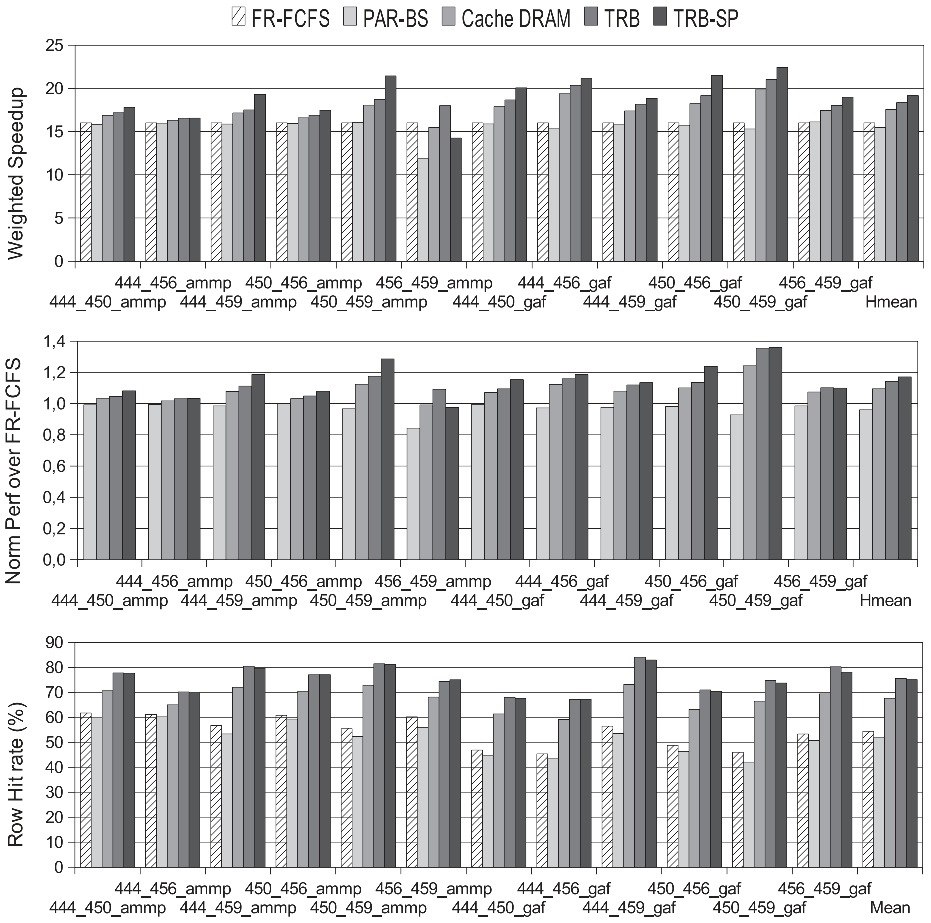


Fig. 11. Weighted Speedup, normalized throughput, and row hit rate.

To compare the fairness of the evaluated techniques, we have used the unfairness index [26]. This is the ratio between the maximum and the minimum memory-related slowdown. Being the memory related slowdown, the memory stall time per instruction when running with other threads divided by the memory stall time per instruction when running alone on the same system:

$$MemSlowdown_i = \frac{MCPI_i^{shared}}{MCPI_i^{alone}},$$

$$Unfairness = \frac{maxMemSlowdown}{minMemSlowdown}.$$

We have evaluated the following configurations to compare the proposed techniques:

FR-FCFS. Our baseline configuration uses eight bank memories, a FR-FCFS scheduler and an on-chip prefetcher with 32 stream buffers of eight entries each.

PAR-BS. We have compared to a state-of-the-art scheduling technique, the Parallelism-Aware Batch Scheduling [26]. We have used a Marking-Cap of 5, which is said to be the best compromise between system throughput and fairness. We also evaluated a version with a Marking-Cap of 16 to ensure that prefetch streams were kept within a batch and results were very similar.

Cache DRAM. This configuration evaluates a Cache DRAM [19] with 16 row entries per bank, an LRU replacement policy and a FR-FCFS bank scheduler.

TRB. This organization shows the results obtained for the Thread Row Buffers, using a row buffer per thread in each bank. This configuration also uses a FR-FCFS bank scheduler that only reorders if the first request produces a row miss.

TRB-SP. In this configuration, TRBs are evaluated with the service partitioning mechanism in the memory controller to grant performance isolation.

6 Results

In this section, we present the evaluation of the TRB compared to existing configurations described previously. Fig. 11 shows the weighted Speedup, normalized throughput and row hit rate of the evaluated organizations. It can be seen in the third plot that Thread Row Buffers are able to increase the row hit rate significantly, from the 54.4 percent of FR-FCFS to 75 percent. This increase is explained by the ability of TRBs to keep several active rows at the same time. Cache DRAM is able to increase the hit rate to 67.6 percent. The row hit rate improvement results in a lower latency and, therefore, an increase in the overall throughput. Weighted speedup is increased on average by 19.7 percent over FR-FCFS, 23.9 percent over PAR-BS, and 9.1 percent over Cache DRAM. In terms of aggregated throughput TRBs also show the best results, improving it on average by 17.1 percent over FR-FCFS, 21.9 percent over PAR-BS, and 6.9 percent over Cache DRAM.

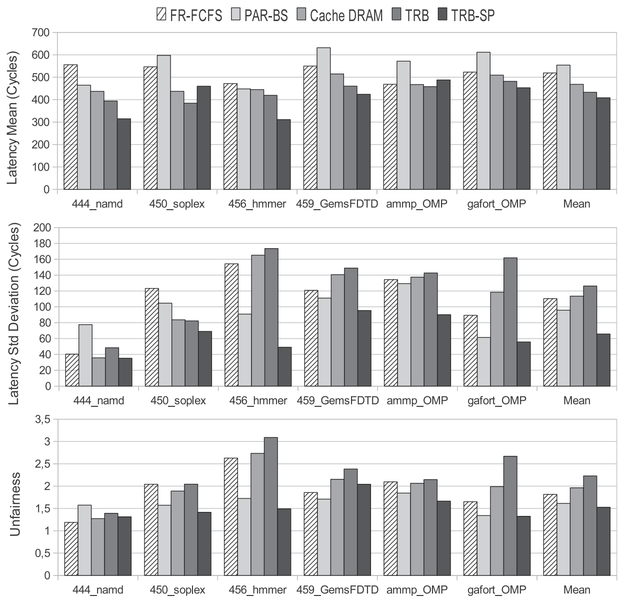


Fig. 12. Average BM latency, standard deviation, and unfairness.

The performance of PAR-BS is similar to FR-FCFS. This is explained by the low hit rate that this configuration achieves. Previous studies evaluated this technique without prefetching, and therefore lower memory-level parallelism. In that environment, a FR-FCFS scheduler is not able to get a reasonable hit rate and, therefore, the penalization of grouping requests into batches is small. The addition of stream prefetchers, however, increases the MLP and shows that under the presence of prefetching batch grouping may reduce the overall throughput.

On the other hand, the TRB configuration has a slightly higher row hit rate than TRB-SP and its performance is lower. This is explained by the performance isolation capability of TRB-SP, which is able to reduce the memory latency of threads with lower number of requests. This results, in most cases, in a performance improvement for the majority of threads. In the case of 456_459_ammp, however, configurations that enforce performance isolation show worse performance. This application combination has the highest average number of simultaneous requests and the impact of the scheduling policy is more clear. In this case, the service partitioning scheduler stalls some requests from ammp to grant a fair access to the other threads, especially those from 456_hmmer. Since ammp represents half of the threads, the overall performance is reduced. This reduction, however, is in exchange for a more fair resource allocation and does not imply a reduction in memory throughput.

To measure the performance isolation, we have used the average memory latency of each application, its standard deviation, and the unfairness metric, shown in Fig. 12. The second plot shows how TRB-SP enforces performance isolation. The performance reduction of ammp in favor of 456_hmmer seen in the 456_459_ammp combination reduces the latency deviation of this application by 68 percent over FR-FCFS.

PAR-BS also shows a good standard deviation due to the usage of a fair scheduler, and is able to reduce it on average by 13.2 percent over FR-FCFS. The average latency, however, is high due to the low hit rate. The configuration with service partitioning, however, shows the best average results. It is able to reduce the average latency by 21.3 percent over FR-FCFS, 26.3 percent over PAR-BS, and 12.8 percent over Cache DRAM. Standard deviation is also reduced by 40.4 percent over FR-FCFS, 31.4 percent over PAR-BS, and 42.1 percent over Cache DRAM. TRBs, on the other hand, grant that the row hit rate is not affected by thread alternation but treat all requests with the same priority. Therefore, applications with a higher number of requests can penalize those with less memory accesses. This explains the variability of TRBs, which is increased by 14.4 percent compared to FR-FCFS. If performance isolation is desired, a specific memory scheduler (like SP) must be implemented in addition to them.

If we compare the techniques through the unfairness metric in the third plot from Fig. 12, we can see that also in this case TRB-SP provides the best fairness. In this case, the unfairness is reduced by 18.8 percent over FR-FCFS, 5.6 percent over PAR-BS, and 28.5 percent over Cache DRAM.

Furthermore, DRAM power in modern server systems can account for 30 percent of total system power [2]. Of this power, around one-third is spent in the precharge and activation of rows. Hence, the aforementioned reduction in row misses also is useful from an energy consumption point of view. Fig. 13 shows that the power consumption per request of the TRB is similar to the power consumed in the other configurations except for the 444_456_ammp configuration. The extra power in this case is caused by a very small number of requests that do not take advantage of the hit rate improvement. The second plot of Fig. 13 shows a decomposition of the memory power consumed in the TRB-SP configuration. It can be seen that in all cases, the DRAM power consumption is reduced due to a higher hit rate that avoids row activations which consume a high amount of energy. However, the extra storage adds a significant amount of static power. On average, TRBs increase the required power per request by 1.8 percent compared to FR-FCFS but reduce it by 8.3 percent compared to Cache DRAM.

If we look at the energy efficiency, however, results are much better for TRBs due to the reduction in the average memory latency (first plot of Fig. 14). Results show that TRB-SP reduces latency by 18.1 percent compared to FR-FCFS and by 3.5 percent compared to Cache DRAM. The second plot shows the energy-delay squared product (ED^2), which is a good measure of energy efficiency (the lower the better). As expected, a reduced latency with similar power requirements leads to a much higher energy-efficiency for TRB configurations. TRB-SP is able to increase the energy efficiency, reducing the ED^2 by 29.1 percent over FR-FCFS, 41.2 percent over PAR-BS, and 14.7 percent over Cache DRAM.

6.1 Addition of Extra Banks

A different alternative to increase the memory access parallelism and increase the overall hit rate would be to use a higher number of banks, each of them with its

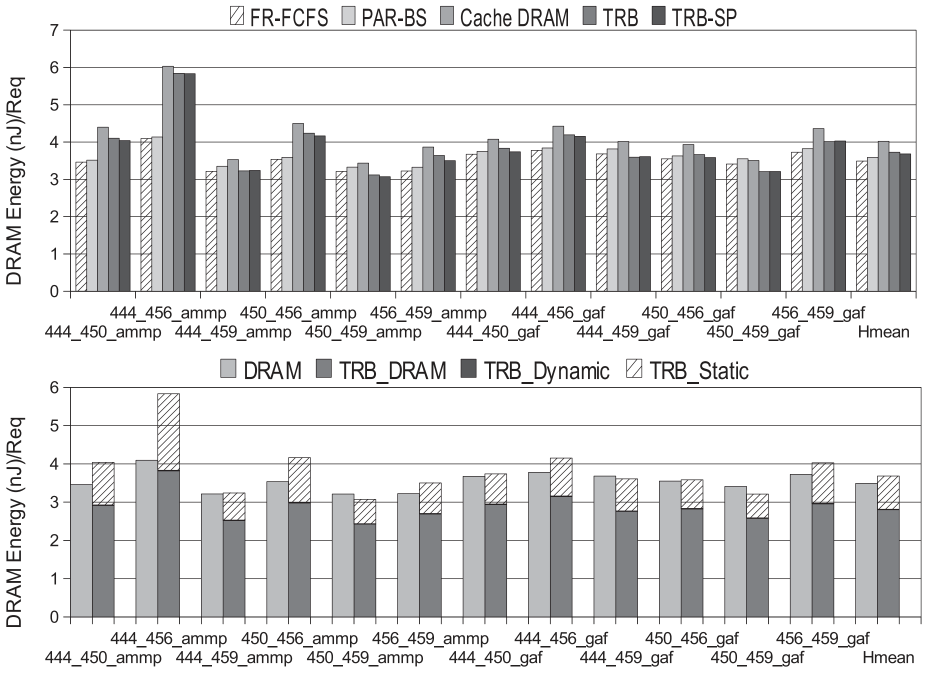


Fig. 13. TRB-SP memory power decomposition.

corresponding row buffer. This solution is orthogonal to the proposed technique and could be combined with it.

It is important to note that the addition of extra banks has a high cost in terms of hardware, especially in the memory controller. It requires adding buffer queues and arbiters for each new bank, therefore doubling the required area when doubling the number of banks. In addition, because the extra banks are not allocated to threads, this solution still has to deal with the throughput-isolation tradeoff. This configuration, however, does have the benefit of being able to activate or precharge more rows simultaneously. Since it is interesting to see the tradeoffs between these alternatives, we have evaluated a configuration with extra banks (16 Bks) and a configuration that combines both techniques (TRB-SP_16 Bks).

Fig. 15 shows the performance improvements and row hit rate of both configurations. It can be seen that Thread Row Buffers are able to improve the overall throughput as much as the configuration that doubles the number of banks with a smaller complexity. If we look at the third plot, we can see that in terms of row hit rate TRBs are much more efficient than adding extra banks. TRBs achieve an average row hit rate of 75 percent compared to the 58 percent of the 16 Bks configuration. Finally, we can see

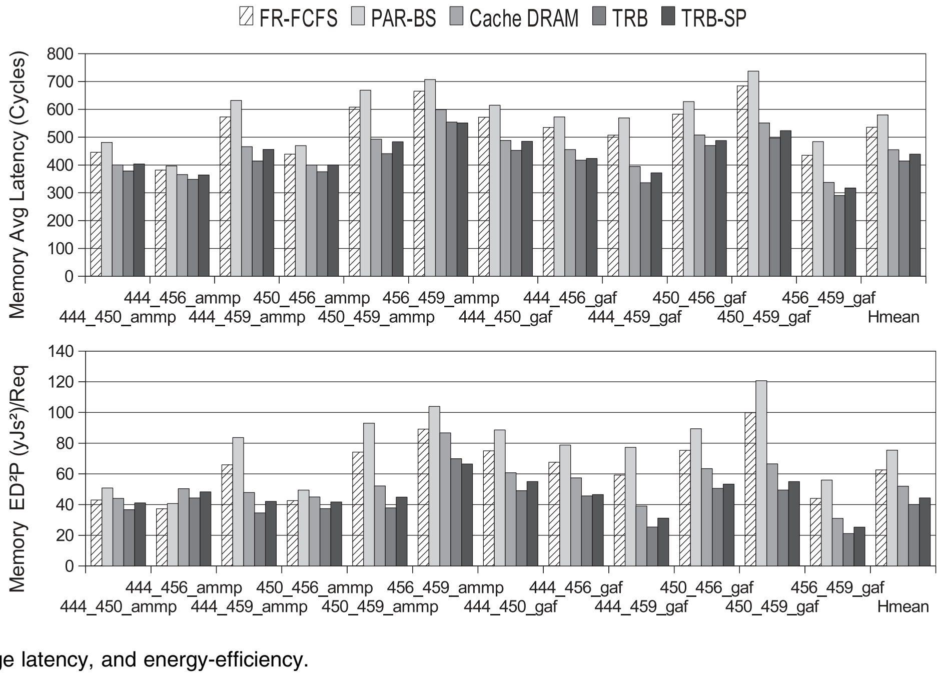


Fig. 14. Power, average latency, and energy-efficiency.

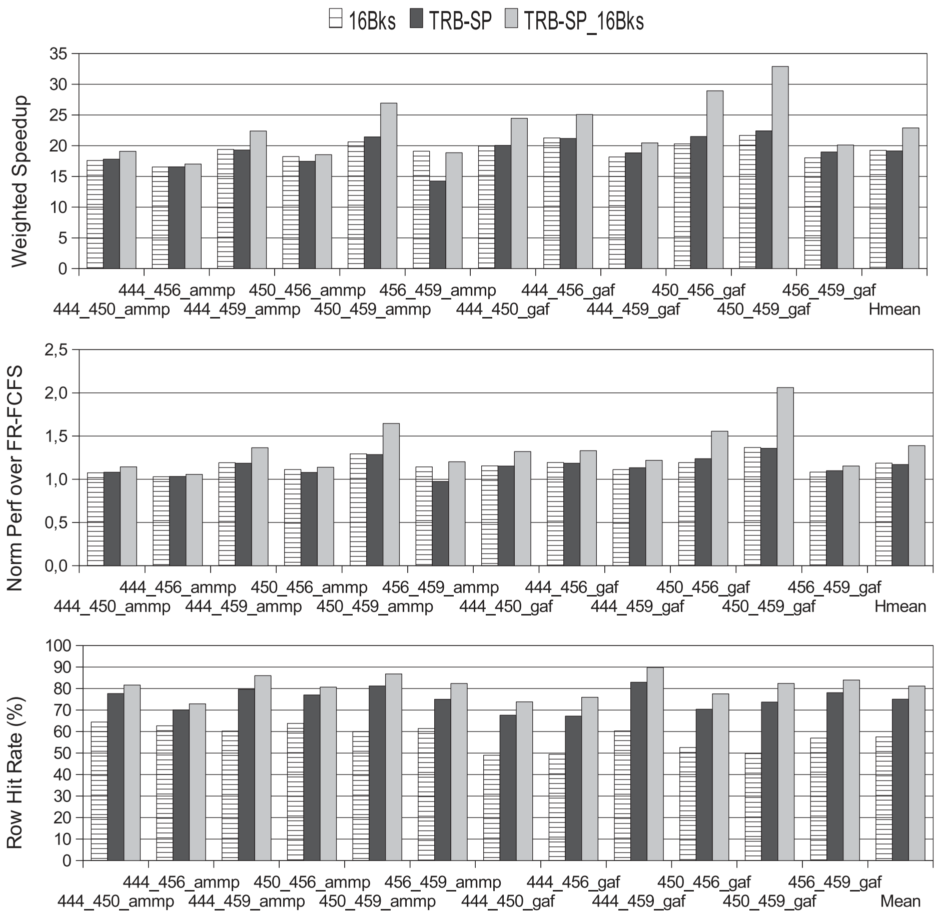


Fig. 15. Weighted speedup, normalized throughput, and row hit rate with extra banks.

that a combination of both techniques, although having a high cost in terms of hardware, can double the overall throughput in some cases, and increase it by 39 percent on average over FR-FCFS.

7 Conclusions

In this paper, we have seen that current chip multiprocessors require that we deal with an increasingly heterogeneous set of applications, which change the memory access patterns and make organizations optimized for single threaded workloads obsolete. Thread Row Buffers are able to deal with alternating row access with the usage of multiple active rows. They are able to increase the row hit-rate by 38 percent with respect to FR-FCFS and by 11 percent with respect to Cache DRAM. This, in turn, increases the overall performance by 17 and 7 percent, respectively.

The increasing number of virtualized environments shows a demand for performance isolation between threads. Isolation in existing memory schedulers has been granted by sacrificing throughput while using traditional DRAM memories. The addition of TRBs eliminates this tradeoff, allowing the implementation of more aggressive schedulers. The configuration with TRBs and service partitioning is able to reduce the standard deviation of an application latency by 40 percent over FR-FCFS, 31 percent over PAR-BS, and 42 percent over Cache DRAM and unfairness by 18.8 percent over FR-FCFS, 5.6 percent over PAR-BS, and 28.5 percent over Cache DRAM.

Overall, we have shown that Thread Row Buffers are an energy-efficient mechanism that allows us to avoid the throughput-isolation tradeoff. It enables the implementation of simple fairness or quality-of-service oriented schedulers while maintaining memory throughput.

Acknowledgments

This work has been supported by the Generalitat de Catalunya under grant 2009SGR1250, and the Spanish Ministry of Education and Science under grant TIN2010-18368.

References

[1] F. Assaderaghi, L.-C. Hsu, and J. Mandelman, "Mixed Memory Integration with NVRAM, Dram and Sram Cell Structures on Same Substrate," US Patent 6,424,011, July 2002.

[2] L. Barroso and H. Holzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Morgan & Claypool, 2009.

[3] E. Ebrahimi, O. Mutlu, C.J. Lee, and Y.N. Patt, "Coordinated Control of Multiple Prefetchers in Multi-Core Systems," *Proc. IEEE/ACM 42nd Ann. Int'l Symp. Microarchitecture (MICRO '42),* pp. 316-326, 2009.

[4] W. fen Lin, S.K. Reinhardt, and D. Burger, "Reducing DRAM Latencies with an Integrated Memory Hierarchy Design," *Proc. Seventh Int'l Symp. High-Performance Computer Architecture (HPCA '01),* pp. 301-312, 2001.

[5] E. Herrero, J. Gonzalez, and R. Canal, "Distributed Cooperative Caching," *Proc. 17th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '08)*, pp. 25-29, Oct. 2008.

[6] H. Hidaka, Y. Matsuda, M. Asakura, and K. Fujishima, "The Cache DRAM Architecture: A Dram with an On-Chip Cache Memory," *IEEE Micro*, vol. 10, no. 2, pp. 14-25, Apr. 1990.

[7] L. Hsu, C. Radens, and L.-K. Wang, "Integrated Chip Having SRAM, DRAM and Flash Memory and Method for Fabricating the Same," US Patent 6,556,477, Apr. 2003.

[8] W.-C. Hsu and J. Smith, "Performance of Cached DRAM Organizations in Vector Supercomputers," *Proc. 20th Ann. Int'l Symp. Computer Architecture*, pp. 327-336, May 1993.

[9] I. Hur and C. Lin, "Adaptive History-Based Memory Schedulers," *Proc. IEEE/ACM 37th Ann. Int'l Symp. Microarchitecture (MICRO 37)*, pp. 343-354, 2004.

[10] E. Ipek, O. Mutlu, J.F. Martínez, and R. Caruana, "Self-Optimizing Memory Controllers: A Reinforcement Learning Approach," *Proc. 35th Ann. Int'l Symp. Computer Architecture (ISCA '08)*, pp. 39-50, 2008.

[11] N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proc. 17th Ann. Int'l Symp. Computer Architecture (ISCA '90)*, pp. 364-373, 1990.

[12] D. Kaseridis, J. Stuecheli, and L.K. John, "Minimalist Open-Page: A Dram Page-Mode Scheduling Policy for the Many-Core Era," *Proc. IEEE/ACM 44th Ann. Int'l Symp. Microarchitecture (MICRO-44 '11)*, pp. 24-35, 2011.

[13] O. Kiehl, "Memory Device and Method Using a Sense Amplifier as a Cache," US Patent 7,215,595, May 2007.

[14] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," *Proc. IEEE 16th Int'l Symp. High Performance Computer Architecture (HPCA '10)*, 2010.

[15] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Prefetch-Aware Dram Controllers," *Proc. IEEE/ACM 43th Ann. Int'l Symp. Microarchitecture (MICRO 43)*, 2010.

[16] A.R. Lebeck, X. Fan, H. Zeng, and C. Ellis, "Power Aware Page Allocation," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pp. 105-116, 2000.

[17] C.J. Lee, O. Mutlu, V. Narasiman, and Y.N. Patt, "Prefetch-Aware DRAM Controllers," *Proc. IEEE/ACM 41st Ann. Int'l Symp. Microarchitecture (MICRO 41)*, pp. 200-209, 2008.

[18] C.J. Lee, V. Narasiman, O. Mutlu, and Y.N. Patt, "Improving Memory Bank-Level Parallelism in the Presence of Prefetching," *Proc. IEEE/ACM 42nd Ann. Int'l Symp. Microarchitecture (MICRO 42)*, pp. 327-336, 2009.

[19] G.H. Loh, "3D-Stacked Memory Architectures for Multi-Core Processors," *Proc. 35th Ann. Int'l Symp. Computer Architecture (ISCA '08)*, pp. 453-464, 2008.

[20] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50-58, Feb. 2002.

[21] M. Martin, D.J. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood, "Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92-99, 2005.

[22] S.A. Mckee and W.A. Wulf, "Access Ordering and Memory-Conscious Cache Utilization," *Proc. IEEE First Symp. High-Performance Computer Architecture (HPCA '95)*, pp. 253-262, 1995.

[23] Micron. DDR3 SDRAM MT41J128M8 Datassheet, technical report, Micron Technology, Inc., 2006.

[24] Micron. Tn-41-01, Calculating Memory System Power for ddr3, technical report, Micron, 2007.

[25] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," *Proc. IEEE/ACM 40th Ann. Int'l Symp. Microarchitecture (MICRO 40)*, pp. 146-160, 2007.

[26] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared Dram Systems," *Proc. 35th Ann. Int'l Symp. Computer Architecture (ISCA '08)*, pp. 63-74, 2008.

[27] K.J. Nesbit, N. Aggarwal, J. Laudon, and J.E. Smith, "Fair Queuing Memory Systems," *Proc. IEEE/ACM 39th Ann. Int'l Symp. Microarchitecture (MICRO 39)*, pp. 208-222, 2006.

[28] N. Rafique, W.-T. Lim, and M. Thottethodi, "Effective Management of DRAM Bandwidth in Multicore Processors," *Proc. 16th Int'l Conf. Parallel Architecture and Compilation Techniques (PACT '07)*, pp. 245-258, 2007.

[29] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, and J.D. Owens, "Memory Access Scheduling," *Proc. 27th Ann. Int'l Symp. Computer Architecture (ISCA '00)*, 2000.

[30] A. Snavely and D.M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Mutlithreading Processor," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 234-244, 2000.

[31] S. Srinath, O. Mutlu, H. Kim, and Y.N. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," *Proc. IEEE 13th Int'l Symp. High Performance Computer Architecture (HPCA '07)*, pp. 63-74, 2007.

[32] W.J. Starke, "Power7: IBM's Next Generation Balanced Power Server Chip," *Proc. IEEE Symp. High-Performance Chips (Hot Chips 21)*, 2009.

[33] J. Stuecheli, D. Kaseridis, D. Daly, H.C. Hunter, and L.K. John, "The Virtual Write Queue: Coordinating DRAM and Last-Level Cache Policies," *Proc. 37th Ann. Int'l Symp. Computer Architecture (ISCA '10)*, pp. 72-82, 2010.

[34] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-Pages: Increasing Dram Efficiency with Locality-Aware Data Placement," *Proc. 15th ed. ASPLOS Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*, pp. 219-230, 2010.

[35] "Micron technology mt41j128m8jp-187e:f," *TechInsights*, technical report, http://www.ubmtechinsights.com/, 2009.

[36] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy, "Power4 System Microarchitecture," technical report, IBM Technical White Paper, 2001.

[37] S. Thoziyoor, N. Muralimanohar, and N. Jouppi, "Cacti 5.0," technical report, Advanced Architecture Laboratory HP Labs, Oct. 2007.

[38] A.N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N.P. Jouppi, "Rethinking Dram Design and Organization for Energy-Constrained Multi-Cores," *Proc. 37th Ann. Int'l Symp. Computer Architecture (ISCA '10)*, pp. 175-186, 2010.

[39] D.T. Wang, "Modern DRAM Memory Systems: Performance Analysis and Scheduling Algorithm," PhD thesis, College Park, MD, 2005.

[40] G.L. Yuan, A. Bakhoda, and T.M. Aamodt, "Complexity Effective Memory Access Scheduling for Many-Core Accelerator Architectures," *Proc. IEEE/ACM 42nd Ann. Int'l Symp. Microarchitecture (MICRO 42)*, pp. 34-44, 2009.

[41] Z. Zhang, Z. Zhu, and X. Zhang, "A Permutation-Based Page Interleaving Scheme to Reduce Row-Buffer Conflicts and Exploit Data Locality," *Proc. ACM/IEEE 33rd Ann. Int'l Symp. Microarchitecture (MICRO 33)*, pp. 32-41, 2000.

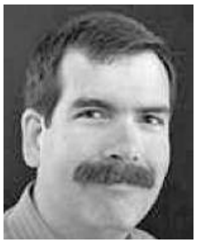
[42] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu, "Mini-Rank: Adaptive DRAM Architecture for Improving Memory Power Efficiency," *Proc. IEEE/ACM 41st Ann. Int'l Symp. Microarchitecture (MICRO 41)*, pp. 210-221, 2008.

[43] W. Zuravleff and T. Robinson, "Controller for a Synchronous Dram that Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued out of Order," US Patent Number 5,630,096, May 1997.

Enric Herrero received the MS and PhD degrees from the Universitat in Politècnica de Catalunya (UPC), in Barcelona, Catalonia, EU, and the MS degree from the Royal Institute of Technology (KTH), in Stockholm, Sweden, EU under a double degree program. He was a visiting researcher at the University of California San Diego (UCSD) in 2009. Currently, he is a research scientist at Intel. His research focuses on reliability and memory hierarchy. He is a member of the IEEE.

José González received the PhD degree in computer science in 2000 from the Universitat Politècnica de Catalunya (UPC), in Spain. Then, he moved to the Universidad de Murcia, where he became an associate professor in 2001. Since 2002, he has been with Intel in the new Microprocessor Technology Lab in Barcelona. His research is focused on processor microarchitecture, especially targeted to power-efficient multicore systems. He has published more than 30 papers in journals and conferences. He is a member of the IEEE.

Ramon Canal received the MS and PhD degrees from the Universitat Politècnica de Catalunya (UPC), in Barcelona, Catalonia, EU. He received the MS degree from the University of Bath, United Kingdom, worked at Sun Microsystems in 2000, and was a Fulbright visiting scholar at Harvard University in 2006/2007. He joined the faculty of the Computer Architecture Department of UPC in 2003. His research focuses on power and thermal aware architectures, as well as reliability and memory hierarchy. He has an extensive list of publications and several invited talks. He has been a program committee member in several editions of HPCA, ISCA, HiPC, IPDPS, ICCD, ICPADS, CF. He has been a cogeneral chair of IOLTS 2012. He is a member of the IEEE.



Dean Tullsen received the PhD degree in computer science from the University of Washington. He is a professor in the Department of Computer Science and Engineering at the University of California, San Diego. His research interests include architecture and compilers for multicore and multithreaded architectures. He is a fellow of the IEEE and the ACM.