Reducing NoC and Memory Contention for Manycores

Vishwanathan Chandru and Frank Mueller

North Carolina State University, Raleigh, NC, mueller@cs.ncsu.edu

Abstract. Platforms consisting of many computing cores have become the mainstream in high performance computing, general purpose-computing and, lately, embedded systems. Such systems provide increased processing power and system availability, but often impose latencies and contention for memory accesses as multiple cores try to reference data at the same time. This may result in sub-optimal performance unless special allocation policies are employed. On a multi-processor board with 50 or more processing cores, the NoC (Network On Chip) adds to this challenge. This work evaluates the impact of bank-aware and controller-aware allocation on NoC contention. Experiments show that targeted memory allocation results in reduced execution times and NoC contention, the latter of which has not been studied before at this scale.

1 Introduction

On many-core platform(s), memory (DRAM) is a resource critical to performance. As applications share cores and become more and more memory intensive, DRAM tends to become a performance bottleneck that critically affects system performance [1].

Performance problems usually arise due to serialization of memory accesses. This can be avoided using bank-aware and controller-aware allocation. The DRAM in manycore platform(s) is divided among multiple memory controllers and, within a controller, ranks and banks. Controllers and banks can be accessed in parallel. Therefore, performance of an application varies significantly depending on how data is placed and how many cores/processors access a given bank at same time. In the best case, each core/processor accesses a different controller/bank. This ensures that contention of accesses does not occur and that accesses are resolved in parallel. One strategy to improve bank-level parallelism is to use bank interleaving. In case of single threaded applications it improves performance. However, when multiple programs or multiple threads are running simultaneously, it can cause cross-interference. The higher the degree of parallelism, the higher is the probability of bank sharing resulting in more cross-interference.

Linux on the Tilera platform can be configured to allocate memory on the closest NUMA node (physical memory controller) [2,3]. Since Linux on the Tilera platform handles DRAM as a single resource following a NUMA allocation policy

(unless and until disabled), banks are not considered during allocation and it is not possible to predict the exact location of allocated memory over the banks.

The Tilera architecture features a mesh NoC (Network on Chip) instead of a bus. All data accesses and data exchanges go through the NoC. Given the large number of cores, traffic over the network can lead to high latencies. Therefore, NoC contention also becomes important for bandwidth and performance. Due to the large number of cores, controller contention also becomes a critical factor.

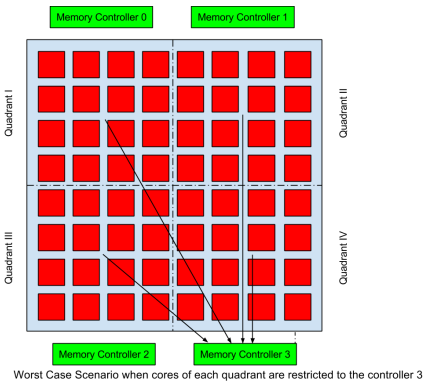

Fig. 1. Worst Case Scenario

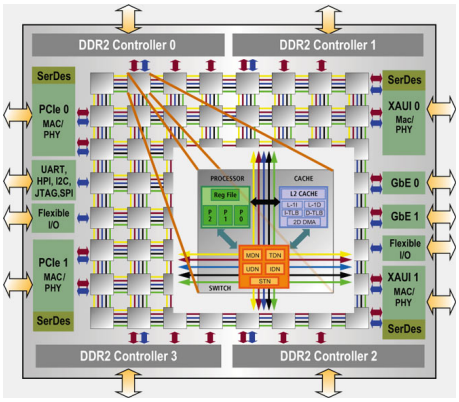

Fig. 2. Tilera Architecture

If all accesses are directed via a particular memory controller as depicted in Fig. 1, then the latencies increase since requests get queued at controller(s) and core(s) stall. Furthermore, cores in each quadrant may access all controllers, which leads to a high volume of traffic. Even though all controllers are being utilized, the latency is high.

Contributions: This work contributes a user space bank-aware and controller-aware allocator that keeps track of bank(s) and controller(s) of allocated memory, i.e., it returns memory addresses requested on a particular bank/controller. This allows users to bind a core/processor to a specific bank and controller reducing the contention and serialization, thus improving performance. Using the allocator, an extensive experimental study was performed to evaluate the impact of bank-aware and controller-aware allocation. It was observed that performance (in terms of execution time) improved with our allocator while its variance was reduced due to less NoC contention.

2 Background And Problems

The Tilera [4] architecture differs from most modern systems due to presence of a mesh NoC instead of a bus (see Fig. 2). Other platforms with large core count now use a mesh or ring NoC as well (Intel Xeon Phi [5], Intel SCC [6], Adapteva [7] etc). Like modern DRAM systems, the memory system is composed of a controller that handles the arbitration, scheduling and conversion of packets to external memory commands. Memory is organized into ranks (4 in our setup) and each rank has multiple banks (8 in our case). The systems may or may not

have multiple independent memory controllers. The Tilera architecture supports four independent controllers that operate in parallel [8].

Tiles (similar to cores) do not have direct access to controllers as the architecture implements a DSM (Distributed Shared Memory) protocol to ensure coherence at the L2 level cache. Tilera utilizes a *mesh interconnect* for data exchange [9]. Dimension ordered routing is used in the network [8]. The interface to off-chip memory and I/O devices is done via *I/O shims* that interface the NoC to memory controllers and other I/O devices [9]. The memory controller is connected to the memory dynamic network (MDN, see Fig. 2) and has multiple ports [8] where requests arrive and are fulfilled.

NUMA allocation is the default policy. However, NUMA and STRIPE allocations might not be able to deliver the best possible performance by themselves as the accesses to data structures will be resolved by different memory controllers with different latencies (hops) over the NoC.

None of the policies are bank aware, i.e., they cannot restrict accesses of a task running on a tile to a particular bank or even controller if memory striping is enabled, where a page (64 kB size) is striped across all controllers in an interleaved way at 8 kB granularity to balance load and improve memory parallelism.

In this work, we present a user space allocator utilizing non-striped mode and using controller-interleaved page allocation instead of the default NUMA policy. We use the interleaved policy for allocating our memory pool as it ensures that an equal amount of memory per controller is available for allocation.

3 Bank Aware Allocator

We designed and implemented a user space bank-aware and controller-aware allocator. It exploits the virtual to physical address translation. After determining the bank and controller from bits within the physical address, the address is added to a specific list corresponding to the relevant controller and bank. When a user requests memory from a certain bank and controller, the corresponding list is searched and if memory is available, its address is returned.

Our allocator requires pre-allocation of a large pool of memory, which is then traversed at a granularity of 8 kB and chunks are added to the corresponding free list(s). The default behavior of the allocator is to try to find a memory chunk fitting the requested size, bank and controller. However, if the exact request cannot be full-filled, the allocator supports multiple modes as an automatic fallback from the default behavior, namely:

CONTROLLER RESTRICTED: If a requested bank is not found within the controller, the allocator defaults to the bank that has the most free memory.

CONTROLLER UNRESTRICTED: If a requested bank and the controller cannot be used to satisfy the request, the allocator defaults to the bank and controller that have the most free memory.

SPLIT: If the requested size is greater than 8 kB and this mode is enabled, then standard allocation is performed (i.e., find the first fitting contiguous chunk available) and its address is returned.

```
Input: Request Size, Memory Controller, Bank, modes mask
block ⟵ NULL;
block ⟵ Memory from requested controller and bank;
if block != NULL then
    return block;
end
if CONTROLLER_RESTRICTED in modes_mask then
    Q ⟵ Banks corresponding to requested Memory Controller with banks in
    descending order of memory availability;
    for each bank in Q do
        block ⟵ allocate memory from current bank if chunk of size greater
        than or equal to requested size is available;
        if block != NULL then
            return block;
        end
    end
end
if CONTROLLER_UNRESTRICTED in modes_mask then
    ControllerQ ⟵ Get controllers in descending order of memory availability;
    for each controller in ControllerQ do
        BankQ ⟵ Get banks from current controller (ordered in descending
        order of memory availability);
        for each bank in BankQ do
            block ⟵ allocate memory from current bank if chunk of size
            greater than or equal to requested size is available;
            if block != NULL then
                return block;
            end
        end
    end
end
block ⟵ NULL;
if SPLIT_MODE in modes_mask then
    block ⟵ Allocate contiguous block of memory (may span across banks and
    controllers);
end
return block;
```

Algorithm 1: Bank aware algorithm allocator

Algorithm 1 shows the implementation. Multiple modes can be configured via a bit-mask for finer control of the allocator. Multiple lists are used to improve the performance of the allocator. Multiple doubly linked list(s) are maintained, one for each bank within a controller. The head of the list is indexed using the controller number/bank number. For SPLIT allocation, a separate list is maintained for quicker allocation. Each free memory chunk is part of both lists. Each memory chunk has four pointers, two pointers that are used to traverse the corresponding bank list and two pointers for the list used for SPLIT allocation.

Depending on the allocation modes, we track the memory available per bank/controller via a queue. Five such queues are maintained, one queue per controller to keep track of banks and one shared queue to keep track of memory available per controller.

This multiple-level design accelerates bank-aware allocation for allocations less than 8 kB. SPLIT allocation is slow as multiple data structures and free lists need to be updated. Hence, experiments exclude the allocation time and focus on real-time applications after they pre-allocate data during the initialization.

In contrast to PALLOC [10], a kernel-level allocator, and other software partitioning approaches [11] [12], our allocator works targets manycores with mesh NoCs (not multicores with bus/ring NoCs) and operates in user space as a proof-of-concept implementation. It is more restrictive in terms of usage due to constraints imposed by manycores. Tilera supports a 64 kB page size and utilizes controller-interleaved page placement, i.e., we cannot allocate more than 64 kB of contiguous memory within a controller (see Fig. 3). As the bank varies every 13 address bits (explained in Section 4.1), a contiguous allocation cannot exceed 8 kB if it needs to be within the *same bank and controller*. Instead, our allocator allows medium allocations ($8KB < size \leq 64KB$) to span multiple banks while larger ones ($> 64KB$) even span multiple controllers.

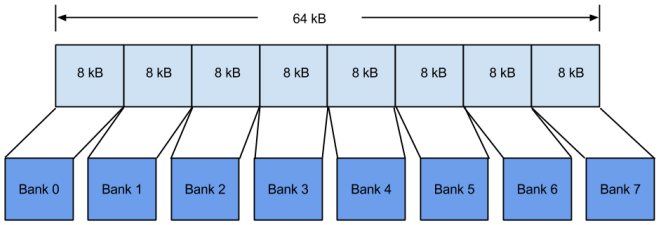

Fig. 3. Bank split up in a 64 kB page

4 Evaluation

The evaluation platform used is a Tilera TILEPro-64 with 64 tiles [13]. Each tile has 16+8 kB private L1I+D cache(s), a 64 kB private L2 cache and a soft L3 cache of 5 MB, which is created by combining the private L2 caches of all tiles (see Fig. 2). There are 4 memory controllers, each capable of independent operation. Each controller can support up to 32 banks (4 ranks and 8 banks per rank). The configuration we utilize has 8 GB of DDR2 RAM, 2 GB per controller. Address hashing [8] is enabled to enhance the number of available banks.

4.1 Address Mapping

Address translation is straight forward. The physical address has 36 bits [8]. Per the documentation and configuration register values, address hashing is per-

formed to increase bank availability, which distributes a page at cache line granularity among the available tiles. Bits 13, 14, 15 are used to determine banks and bits 34 and 35 are used to determine the controller (in combination covering all 32 banks).

4.2 Experimental Setup

To measure bank-level contention, we use two OpenMP benchmarks. The entire processor is divided into 4 quadrants with 16 tiles each, except quadrants 3 and 4 with only 12 tiles each as the bottom-most 8 tiles are reserved by Tilera's SDK in our setup. The primary aim of dividing tiles into quadrants is to restrict the access to memory via the closest memory controller. This allows us to remove controller contention across quadrants and to minimize NoC contention so that we can focus on measuring bank level contention. This also helps if we want to create pathological worst case scenarios, i.e., tiles from each quadrant accessing all four controllers to generate excessive memory contention. The execution time is indicative of contention. We refer to the controller-restricted policy as "colored allocation", i.e., choosing a page located at the closest controller to minimize NoC path length and this contention. The selection of c colors indicates that threads access controller-local banks shared by as few threads as possible depending on the configuration. Colored allocation results in lower execution time and better memory bandwidth compared to non-colored allocation. Non-colored, using tmc_alloc_map() from the Tilera Multicore Components (TMC) library, is subsequently referred to as the default allocator, which observes controller locality and uses address hashing (see Section 4.1) affecting banks, only, i.e., same or different banks may be chosen in an indeterminate manner.

During the entire experiment, threads were pinned to the respective tiles to prevent them from migrating and causing unintended interference. Benchmark data is explicitly allocated from the bank/controller-aware allocator using coloring.

5 Experiments And Results

Let us assess the impact of bank-aware allocation on performance. We use execution time to measure the performance impact of latency, bandwidth and contention. By running the same benchmark in multiple tile(s) and using an OpenMP enabled benchmark (where OpenMP constructs map to POSIX threads), the impact of bank-aware and controller-aware allocation is shown. The *x-axis* of the plots denotes the i^{th} experiment/execution (experiment instance) and the *y-axis* shows overall memory bandwidth (MB per second) or execution time (seconds) depending on the plot.

Fig. 4 depicts the execution time (y-axis) of the NAS IS OMP benchmark for 32 threads (8 threads per quadrant) for 15 runs. We obtain close to a 20% performance benefit with colored allocation, which improves as colors increase from four per controller (each color/bank shared by 2 threads) to eight (completely

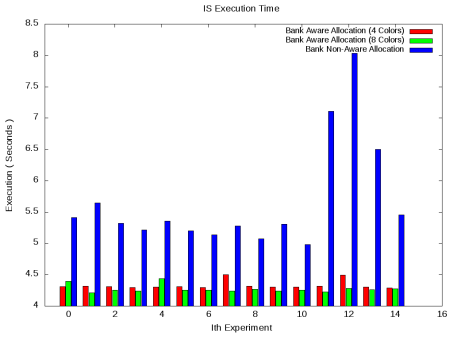

Fig. 4. NPB IS OMP (Class A)

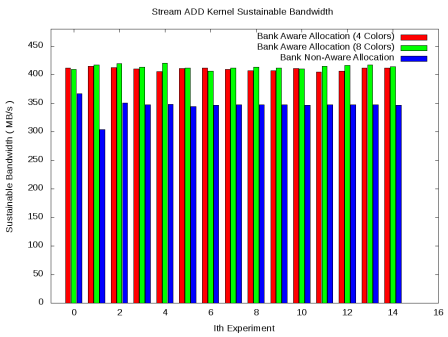

Fig. 5. Stream Add Bandwidth

thread-private banks). We also observe significant performance variations when bank non-aware allocation is used. Fig. 5 depicts the bandwidth on y-axis reported by the STREAM benchmark with 32 threads and 8 threads per quadrant. We observe a significant increase in sustainable bandwidth for the ADD kernel from the STREAM benchmark for 15 runs. Bank non-aware allocation results in higher variation again (for 1 of 15 experiments).

To further understand NoC contention and bank contention, composite benchmark executions (for bubblesort from the Mälardalen benchmark suite) with varied number of threads and memory traces were obtained. Bubblesort is the most memory bound of all integrated benchmarks. Composite benchmark runs are reported in terms of iterations. To get reliable data and account for outliers, multiple iterations are run per execution and experiments are repeated (reported as instances). Fig. 6 shows the overall maximum execution time per experiment of the composite benchmark for 32 threads and 2 pages per thread. We observe that colored allocation always results in the best performance isolation. The two pathological worst case scenarios, tasks in each quadrant accessing all quadrants (labeled as circular allocation) or tasks restricted to a particular controller, always lead to the worst performance. This highlights the criticality of NoC contention and controller contention. Controller-local allocation is slightly slower than bank-aware allocation within a controller (colored allocation) as the latter reduces serializations of accesses.

The difference between 2, 4 and 8 color allocations is noticeable. This is due the fact that even though we are simulating two and four bank configuration(s) by wrapping around bank indices when populating free lists at user level, at hardware level we still have 8 banks, which is a higher bank level parallelism than can be exploited by a four or two color scheme.

To further analyze the impact of colored allocation, the following two sets of experiments were performed:

1. Constant memory footprint and variations in the number of threads.
2. Constant number of threads and variable memory footprint per thread.

Fig. 7 shows the overall maximum execution time per experiment/execution of the composite benchmark for 32 threads and 10 pages per thread. As we

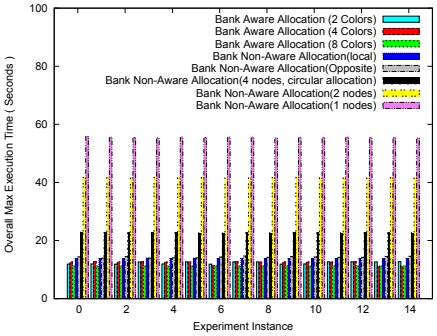

Fig. 6. Composite Benchmark (32 Threads, 2 Pages per Thread)

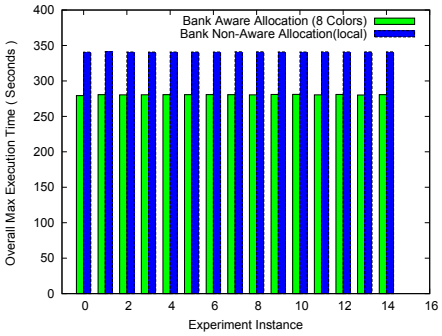

Fig. 7. Composite Benchmark (32 Threads, 10 Pages per Thread)

compare the plots in Fig. 6 with Fig. 7, we observe an increase in execution time by a factor of 21, even though the input size increased five-fold. This is mainly due to memory boundedness of the latter vs. L2-boundedness of the former experiment. An improvement of 17% to 20% over non-bank aware controller-local allocation was observed on average.

For experimental case 1, we fixed the memory footprint to 10 pages per thread and varied the number of threads from 8 to 32 in steps of 8, i.e., increments of 2 threads per quadrant. We observe that the difference between colored allocation and bank non-aware controller-local allocation keeps increasing until 32 threads, and then remains close to 60 seconds (see Figures 6, 7, and 8). This is due to the fact that each controller has 8 banks per rank, i.e., for up to 32 threads we were able to restrict each thread in the quadrant to a separate bank. However, as we exceed 8 threads per quadrant, more than one thread accesses a given bank leading to increased contention. Now, as we reduce the number of threads to 8, non-colored controller-local allocation provides good performance most of the times (see Fig. 9). There are two possible reasons for this behavior.

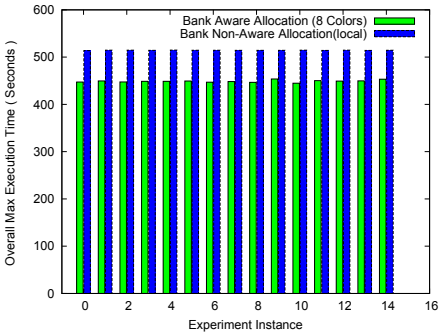

Fig. 8. Composite Benchmark (48 Threads, 10 Pages per Thread)

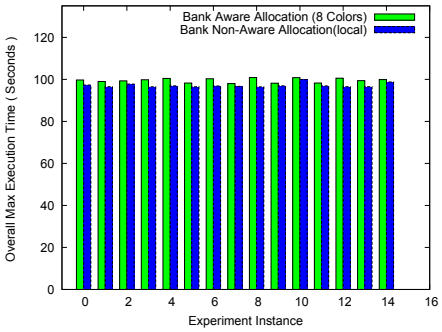

Fig. 9. Composite Benchmark (8 Threads, 10 Pages per Thread)

One is bank address hashing. In case of non-bank aware controller-local allocation, we perform 8 kB allocations using Tilera's API that allocates at 64 kB (page size) granularity. Effectively, we only access the first 8 kB of each page, for which the bank is randomized due to bank address hashing. Our colored allocation restricts thread 0 in each quadrant to bank 0 of the closest controller and thread 1 to bank 1 of the closest controller, effectively restricting accesses to 2 banks. But in case of non-colored allocation, due to bank address hashing, the first 8 kB of the page (which we access) can be any bank (from 0-4), so there is a high probability of more bank level parallelism. The second, less probable reason is that the number of threads per controller is less than the number of MDN ports per controller. There are two threads per quadrant and 3 MDN ports per controller. Therefore, any delay due to serialization at ports is avoided.

Table 1 shows the standard deviation of the execution times of IS and sustainable bandwidth reported by STREAM over 15 executions. We can observe that bank-aware allocation results in tighter bounds for IS and STREAM.

Table 1. Standard Deviations for IS and STREAM

Benchmark	**4 Banks**	**8 Banks**	**Controller-local**
NAS IS OMP	0.06824	0.06272	0.866052
STREAM	2.950072	3.919421	12.63999

Table 2 shows the standard deviation over 15 experiments corresponding to the plots of each experiment having *5 iterations* for Composite Benchmarks. We observe increased jitter compared to controller-local allocation except for 8 banks. We can also observe that as we increase the number of threads from 32 to 48, the jitter decreases as we change the number of threads from 8 to 32, which is due to increased bank-level parallelism. However, as we increase the number for threads to 48, the increased NoC contention and bank sharing subsume the performance improvement. For controller-local allocation, a higher level of serialization exist as we increase the memory footprint, which causes lower performance at reduced jitter.

Table 2. Standard Deviation for Composite Benchmark Execution

threads,pages	2banks	4banks	8banks	ctrl-local	opposite-ctrl	1node	2nodes	circular-alloc
32,2	0.39	0.57	0.02	0.07	0.22	0.12	0.042	0.036
8,10	–	–	1.13	0.67	–	–	–	–
32,10	–	–	0.38	0.19	–	–	–	–
48,10	–	–	2.23	0.38	–	–	–	–

6 Related Work

Jeong et al. [11] propose to optimize power consumption, increase memory throughput and improve performance by means of bank-level partitioning but do not consider multiple controllers or NoC contention. Instead of actual hardware, the evaluation is based on a simulator. Fourth, the effect of the NUMA allocation policy is not considered. Park et al. [14], another software-based approach, propose to increase memory bandwidth and reduce cross interference in a multicore

system by dedicating multiple dedicated banks to different cores. Page allocation within the dedicated banks is randomized to reduce row-buffer conflicts and to further minimize unwarranted cross-interference. Due to the implicit assumption of more banks than cores, this approach excels when the number of banks is much larger than the number of cores. It can also be observed that none of the approaches focus specifically on improving worst-case execution time (WCET).

Since multicore systems have multiple memory channels, memory channel partitioning is one potential solution to improve performance isolation. Muralidhara et al. [15] propose an application-aware memory channel partitioning. They consider partitioning memory channels and prioritized request scheduling to minimize the interference between memory-bound tasks from cache bound tasks and CPU bound tasks. Apart from software based approaches, there are multiple works on improving predictability of memory controllers. PRET [16] employs a private banking scheme, which eliminates bank sharing interference. Wu et al. [17] take a similar approach but both approaches differ in scheduling policy and page policy. There are several other works closely related to our work [18], [19], [20]. AMC [20] focuses on improving the tightness of WCET estimation by reducing interference via bank interleaving and reducing inferences using a close-page policy. The drawback of this proposal is that instead of treating banks as resources, it treats memory as a resource. Akesson et al. [18] present a similar approach to guarantee a net bandwidth and provide an upper bound on latency. They use bank interleaving to increase the available bandwidth. For bounding latency, they use a Credit-Controlled Static-Priority arbiter [21]. The drawback of this approach is also is same as that of AMC.

Goossens et al. [19] presents a proposal to improve the average performance of hard and soft real time system(s) on a FRT (firm real-time) controller without sacrificing hard/firm real time guarantees. It utilizes bank interleaving and proposes a new conservative closed page policy to maintain locality within a certain window. The drawback of this approach is that it does not eliminate bank conflicts completely.

Caches also impact performance and there are several studies regarding the same at both hardware and software levels [22], [23], [24], [25], [26], [27], [28], [29], [30]. The basic idea behind software-based approaches is cache coloring.

Buono et al. [31] experiment with different Tilera allocations for their FastFlow framework that provides an abstraction for task-based programming.

In contrast to the above references, our work focuses on the impact of NoC contention for bank-aware and controller-aware allocation using multi-threaded codes. This makes it significantly different from prior studies and does not allow a direct comparison.

7 Conclusion

In this paper, we presented a prototype bank-aware and controller-aware user space allocator for increased memory performance on the Tilera architecture. It restricts tiles to access memory of specific banks in addition to a specific

controller, thus minimizing bank sharing while balancing and enhancing available bandwidth and performance.

Using this allocator, we performed experiments with the STREAM and the NAS IS OMP benchmarks. Based on our results, we conclude that bank-aware allocation improves performance, increases memory bandwidth utilization, and reduces NoC contention, the latter of which has not been studied before and is becoming a problem for large-scale multicores with many memory controllers.

Acknowledgment

Tilera Corporation provided technical support of the research. This work was funded in part by NSF grants 1239246 and 1058779 as well as a grant from AFOSR via Securboration.

References

1. W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.
2. *Programming The Tile Processor*, Tilera, "http://www.tilera.com/".
3. *Application Libraries Reference Manual*, Tilera, "http://www.tilera.com/".
4. "Tilera processor family," www.tilera.com.
5. "Intel xeon phi," https://www-ssl.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-datasheet.html, April,2015.
6. "Single-chip cloud computer," blogs.intel.com/research/2009/12/sccloudcomp.php.
7. "Adapteva processor family," www.adapteva.com/products/silicon-devices/e16g301/.
8. *Tile Processor I/O Device Guide*, Tilera, "http://www.tilera.com/".
9. *Tile Processor User Architecture Overview*, Tilera, "http://www.tilera.com/".
10. H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, vol. 356, 2014.
11. M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez, "Balancing dram locality and parallelism in shared memory cmp systems," in *International Symposium on High Performance Computer Architecture*, 2012, pp. 1–12.
12. L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *International Conference on Parallel Architectures and Compilation Techniques*, 2012, pp. 367–376.
13. *Tile Processor User Architecture Reference*, Tilera, "http://www.tilera.com/scm".
14. H. Park, S. Baek, J. Choi, D. Lee, and S. H. Noh, "Regularities considered harmful: forcing randomness to memory accesses to reduce row buffer conflicts for multi-core, multi-bank systems," in *ACM SIGPLAN Notices*, vol. 48, no. 4, 2013, pp. 181–192.
15. S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *International Symposium on Microarchitecture*, 2011, pp. 374–385.

16. J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "Pret dram controller: Bank privatization for predictability and temporal isolation," in *International conference on Hardware/software codesign and system synthesis*, 2011, pp. 99–108.
17. Z. P. Wu, Y. Krish, and R. Pellizzoni, "Worst case analysis of dram latency in multi-requestor systems," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, 2013, pp. 372–383.
18. B. Akesson, K. Goossens, and M. Ringhofer, "Predator: a predictable sdram memory controller," in *International conference on Hardware/software codesign and system synthesis*, 2007, pp. 251–256.
19. S. Goossens, B. Akesson, and K. Goossens, "Conservative open-page policy for mixed time-criticality memory controllers," in *Conference on Design, Automation and Test in Europe*, 2013, pp. 525–530.
20. M. Paolieri, E. Quiñones, F. J. Cazorla, and M. Valero, "An analyzable memory controller for hard real-time cmps," *Embedded Systems Letters, IEEE*, vol. 1, no. 4, pp. 86–90, 2009.
21. B. Åkesson, L. Steffens, E. Strooisma, K. Goossens *et al.*, "Real-time scheduling of hybrid systems using credit-controlled static-priority arbitration," in *RTCSA*, 2008.
22. S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *International Conference on Parallel Architectures and Compilation Techniques*, 2004, pp. 111–122.
23. K. J. Nesbit, J. Laudon, and J. E. Smith, "Virtual private caches," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 57–68, 2007.
24. J. Liedtke, H. Hartig, and M. Hohmuth, "Os-controlled cache predictability for real-time systems," in *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, 1997, pp. 213–224.
25. J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, 2008, pp. 367–378.
26. X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *European conference on Computer systems*, 2009, pp. 89–102.
27. L. Soares, D. Tam, and M. Stumm, "Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer," in *International Symposium on Microarchitecture*, 2008, pp. 258–269.
28. X. Ding, K. Wang, and X. Zhang, "Srm-buffer: an os buffer management technique to prevent last level cache from thrashing in multicores," in *Conference on Computer systems*, 2011, pp. 243–256.
29. B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, "Outstanding paper award: Making shared caches more predictable on multicore platforms," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, 2013, pp. 157–167.
30. R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, 2013, pp. 45–54.
31. D. Buono, M. Danelutto, S. Lametti, and M. Torquati, "Parallel patterns for general purpose many-core," in *Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2013, pp. 131–139.