Asymmetry-aware execution placement on manycore chips

Alexey Tumanov, Joshua Wise, Onur Mutlu, Gregory R. Ganger
Carnegie Mellon University

ABSTRACT

Network-on-chip based manycore systems with multiple memory controllers on a chip are gaining prevalence. Among other research considerations, placing an increasing number of cores on a chip creates a type of resource access asymmetries that didn't exist before. A common assumption of uniform or hierarchical memory controller access no longer holds. In this paper, we report on our experience with memory access asymmetries in a real manycore processor, the implications and extent of the problem they pose, and one potential thread placement solution that mitigates them. Our user-space scheduler harvests memory controller usage information generated in kernel space on a per process basis and enables thread placement decisions informed by threads' historical physical memory usage patterns. Results reveal a clear need for low-overhead, per-process memory controller hardware counters and show improved benchmark and application performance with a memory controller usage-aware execution placement policy.

General Terms

manycore scheduling, Network-on-Chip, operating system

1. INTRODUCTION

Modern manycore platforms organize on-chip cores in a grid connected with a "Network-on-Chip" (NoC) interconnect (e.g., Figure 1). New hardware characteristics such as this pose an array of interesting research challenges for the design and architecture of operating systems. These challenges are further exacerbated by the continuing growth of the number of cores on the die and, correspondingly, the diameter of the 2D NoC. With the increasing need for memory bandwidth on manycore systems, multiple memory controllers (MC) are placed on the die around the periphery of the grid. As a result, each core may incur a different memory access latency depending on which MC it accesses and its relative distance to that controller.

This raises an important question: can intelligent placement of execution threads on the network improve system performance and better satisfy per-application performance requirements? If so, OS designers should concern themselves with initial placement, careful monitoring, and runtime migration of threads in response to a number of runtime characteristics to mitigate the effects of variable memory access latency on performance. One of these characteristics is the thread's DRAM usage—the degree to which it utilizes each MC—and its sensitivity to memory access latency.

Variability of memory access latency is, thus, an important aspect to consider when designing an operating system scheduler or a virtual machine placement scheduler in a manycore datacenter—an argument we motivate with preliminary results obtained on the 64-core Tilera processor [1]. To our knowledge, the effect of this variability on a real system has not seen extensive study before. We share our experience addressing this challenge on a 8x8 core system running a TILE Linux port and present motivating evidence of as much as 14% variability in application performance caused by NoC routing distance variation. We show promising signs of asymmetry-aware execution placement advantages, full realization of which, however, calls for a holistic cooperative effort of multicore hardware and OS architects.

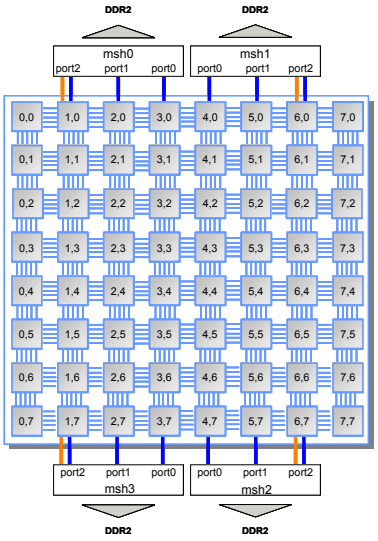

Figure 1: *TilePro64* interconnect architecture [1].

2. RELATED WORK

Much work was done on traditional NUMA and its effects on thread scheduling [6]. We argue that Networks-on-Chip are fundamentally different from traditional NUMA systems. Indeed, sharing of resources is a lot more extensive and fine-grained in NoC systems and spans the entire cache/memory system. There's also no longer a notion of strict memory hierarchies nor a sharp difference in latency between *local* and *remote* memory node access. Instead, NoCs and the use of multiple MCs on a chip result in a multi-dimensional continuum of NoC-induced memory access latencies. We thus make a key observation that the traditional NUMA concept of local vs. remote memory controller access breaks down in the context of NoC-based manycore architectures. Whereas previously the primary concern was binary (local vs. remote access), the access latencies we observe are now more continuously variable as a function of static and

dynamic factors. In this paper, we focus on one such factor, namely the Manhattan core-to-MC distance. Recent work by Awasthi et al. [2] considers interconnect architectures that, by their design, fall into a more hierarchical NUMA category (e.g., quad-socket quad-core), and chooses a memory page placement-centric solution. We focus on continuously variable access latency interconnects, exemplified by mesh-based manycore chips, such as a 64 core TilePro64 [1], and depart from [2] with our execution placement-centric solution. Data-to-execution and execution-to-data placement strategies could also be combined.

In the architecture and hardware design communities, packet-switched on-chip networks [7] have recently received widespread attention, with research focused on topology, routing, flow control, energy management, and quality of service. Almost all of this research is on improving the hardware design. While this work has considered hardware support for providing quality of service, fairness, and performance isolation in the presence of multiple applications sharing the NoC [2, 9, 11, 13, 19], none of these efforts have investigated OS-level issues that arise from the sharing of the NoC.

Recent research provided hardware support for partitioning bandwidth and capacity in shared hardware resources, e.g. caches [18], memory controllers [12,17], and on-chip networks [9, 11, 13]. These mechanisms can be used as building blocks by a manycore operating system. However, all of the proposals were evaluated without consideration for OS design. Similarly, OS-level techniques that aim to reduce contention in the memory system [10, 23] assume that the hardware provides no support. Based on empirical evidence in current NoC-based systems, such as the TILEPro64 processor, our vision departs from hardware-only or OS-only approaches and is more in line with Jeff Mogul's call for OS-driven innovation in computer architecture [15]. We believe a simultaneous investigation of hardware and OS mechanisms driven by the needs of the OS at the higher level and characteristics of the network-on-chip based hardware at the lower level can lead to more scalable and efficient manycore operating systems.

Lastly, Boyd-Wickizer et al. [5] present an evaluation of Linux scalability to manycores. They mention, but do not explore the inter-core interconnect as one source of scalability bottlenecks. Barrelfish [3], Corey [4], and fos [22] propose more scalable OS designs for manycore systems but they 1) assume no hardware support, and 2) do not focus on the challenges brought about by on-chip networks.

3. VARIABLE MEMORY ACCESS LATENCY

We start with a set of motivating experiments that validate our prediction of NoC-latency induced variability in execution thread performance. Inspired by this, we move on to our preliminary efforts to construct a mechanism for runtime memory access profiling and reactive placement of execution threads based on their memory utilization characteristics. As the overhead of instrumenting virtual memory subsystem motivates the need for hardware support, we demonstrate the promise of asymmetry-aware thread placement in our comparison of preliminary performance results to the *overhead* placement policy.

3.1 System Setup

All of our development, measurement, and experimentation was carried out on a real manycore system, namely the

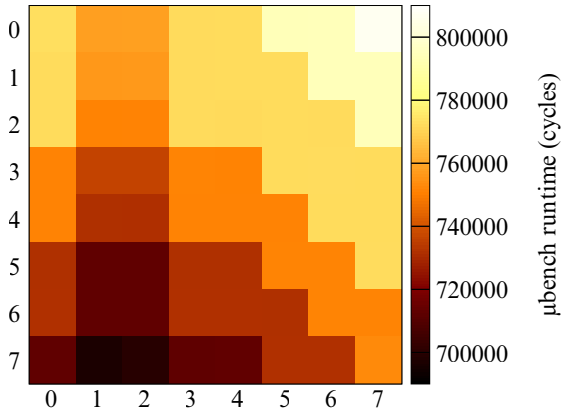

Figure 2: Microbenchmark heatmap. Axes are core rows & columns.

TILExpressPro-64 board. The TilePro64 [1] is a grid of 8 by 8 TILE processors arranged in an on-chip two-dimensional network (Figure 1). This board was installed in a standard host Linux system, and benchmarks were run using the board's on-board frequency sources. We worked with the board at the hypervisor level as well as on top of the Tilera Linux port.

3.2 Motivating Experiments

We begin by running a microbenchmark directly on top of the hypervisor to confirm that access latency to any given memory controller varies with the placement of an execution task. Our microbenchmark consists of a simple execution kernel that performs a read/modify/write/flush cycle on each cache line in a large block of memory allocated from one of the four available memory controllers. It is run on each tile in sequence. Benchmark runtime is measured in cycles and ranges from 700000 to 800000, depending on the core location. In Figure 2, the bottom left MC was accessed, and the heatmap reveals a gradual increase of the benchmark's runtime with the placement distance away from the controller. A spread of as much as 14% is observed, directly attributable to the NoC propagation latency for a *single* execution thread unhampered by NoC link contention. Higher latency variation is possible, especially in the presence of competition for NoC links, which will have an amplifying effect on latency asymmetry.

We then verify the continued presence of latency variation when running a real application atop the Linux port. GCC's front end, `cc1`, was used to process a large source file on each of the available cores in the 8x8 grid, while the task was bound to a single memory controller. Remarkably, a heatmap similar to 2 was produced, with the corresponding scatter plot in Figure 3 showing a clear linear relationship between the Manhattan distance from the memory controller to the execution thread and its runtime. In fact, the observed variation in application runtime between the closest and the furthest point of execution was also 14%, matching our $\mu-$benchmark results. Based on this, it becomes clear that real applications can benefit from execution thread placement optimizations in the network-on-chip. The extent of that benefit will depend on the application's memory access patterns. We expect memory intensive and memory latency sensitive (i.e., those that exhibit low memory

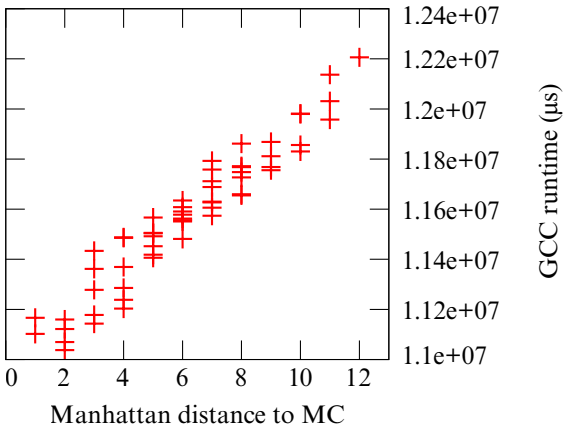

Figure 3: Manhattan distance to MC vs. runtime

parallelism) applications to benefit the most. Such applications are most sensitive to interference in the network and MCs [8]. We leave full analysis of application classes and their sensitivity to memory access latency asymmetries for future work.

4. OUR SOLUTION

4.1 Placement Algorithm

Here we briefly summarize our initial algorithm for MC usage and location-aware placement of execution threads. The basic idea is to have the operating system keep track of each task's access counts to each MC at fixed time intervals and use those counts to place tasks on the grid such that the average latency to any memory controller from each task is minimized. The placement algorithm is based on the core concept of a weight vector, $\vec{w}(t)$, learned from the physical page access patterns to each memory controller for each monitored process. We collect a vector of page access counts, with one element per on-chip memory controller, by instrumenting the virtual memory subsystem. The count vector is then normalized to produce an L_1-normal unit weight vector used to calculate the (x, y) coordinates for the monitored thread as follows (where (x_i, y_i) are the coordinates of each MC):

$$Coord(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix} = \begin{bmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \end{bmatrix} \begin{bmatrix} w_0(t) \\ w_1(t) \\ w_2(t) \\ w_3(t) \end{bmatrix}$$

We refer to this placement strategy as *weighted geometric* and compare it against three others. First, the *baseline* thread placement is the default "tile" architecture Linux scheduling policy with no MC stats collection. Second, the *overhead* policy is identical to *baseline*, but with MC usage tracking turned on. Third, the *brute force* placement strategy is a search performed over a space of available cores minimizing the cost function, defined as follows:

$$\text{costf}(x, y, \vec{c}(t)) = \sum_i (|x - x_i| + |y - y_i|) * c_i$$

where (x, y) are the coordinates of the candidate tile, c_i is the i^{th} element of the access count vector $\vec{c}(t)$ associated with memory controller i, and (x_i, y_i) are the corresponding memory controller coordinates on the 2D grid.

Once computed, the placement map dictates the migration of threads to their coordinates. Conflicts in optimal core selection were infrequent and were resolved using a gradient descent method to probe directly adjacent candidate cores.

4.2 MC Usage Collection Mechanism

The placement calculation for monitored threads is predicated on the knowledge of $\vec{c}(t)$. In the absence of programmable performance counters that could expose to the operating system per-process memory controller usage statistics, we use the virtual memory subsystem to maintain a discretized running log of which pages on the system have been accessed. We do this by repeatedly setting the PTEs of all pages in a monitored process to be unreadable. As the system continues its execution, the monitored process generates page faults when accessing those pages. Our mechanism intercepts these page faults and determines which faults were caused by our interposition. The latter set of faults then contributes to access counters for corresponding memory controllers, thereby updating $\vec{c}(t)$. The usage of the page fault handler to estimate memory controller accesses foreshadows significant overhead associated with this collection mechanism. Furthermore, in the absence of hardware based MC access counters, our best effort is to track the number of page faults triggered by *periodic* PTE resets, which serves as a proxy to the real count of MC accesses.

Furthermore, we add a toggle mechanism through Linux's `/proc` interface to enable or disable MC access statistics collection on a per-process basis and to read out the collected statistics at placement decision time. The sampling callback fires at 20Hz when enabled. The user-space execution thread scheduler subsequently polls collected MC usage statistics at its own parameterizable rate and makes thread placement decisions in accordance with the chosen policy.

4.3 Lessons Learned and Observations

First, through a series of experiments, we demonstrate that the overhead of MC access count approximation overshadows the benefits of MC location- and usage-aware adaptive scheduling. The experiment consisted of 60 trials for each of the placement policies, with the distribution for the slowest, median, and fastest tasks captured by the corresponding boxplots in Figures 4 and 5. In Figure 4, the workload consists of 1, 2, 4, 8, or 16 copies of the off-the-shelf STREAM [14] benchmark, and tracks the copy bandwidth of each executing task. As the number of tasks (and the associated page faulting overhead) grew from one to 16, the *weighted* policy consistently underperformed the *baseline*. In the case of 16 tasks, the loss of performance due to our prototype was 5%. In fact, a comparison based on median task performance reveals up to 10.5% performance degradation of *overhead* from the *baseline*, indicating there is a tangible loss due to the page-fault-based MC statistics collection mechanism. At the same time, *weighted* outperformed *overhead* by a 5% increase in performance, demonstrating a clear opportunity to improve performance with asymmetry-aware task placement.

While in Figure 4 we explore the effect of our scheduler on the memory bandwidth performance of managed threads, we measure its runtime performance in Figure 5. The latter reports the amount of time taken to complete a striding

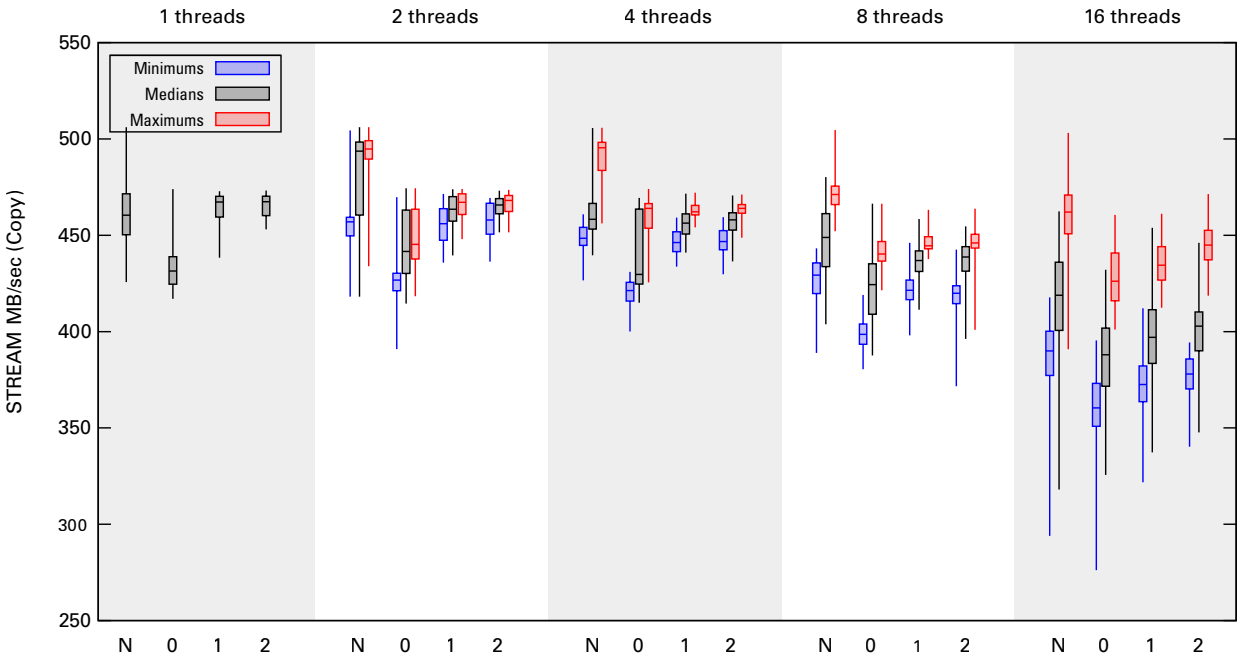

Figure 4: Memory bandwidth performance (MB/s). Each group of 4 box plots corresponds to 1, 2, 4, 8, or 16 instances of the benchmark being executed. Within each group, we compare *baseline*(N), *overhead*(0), *weighted*(1), and *brute force*(2). Higher is better.

read/modify/write pattern benchmark, similar to the one used for Figure 2, which we call bench2. Again, we observe some overhead compared to the *baseline*, but the runtime performance is improved in all cases for both asymmetry-aware placement strategies we evaluated (*weighted* and *brute force*) as compared to the *overhead* policy.

These results call for programmable MC access counters exposed to the manycore OS by the architecture. Their availability will not only eliminate the overhead of MC stats collection, but will also provide a stable, reliable stream of real MC usage data instead of its statistical surrogate, which may be inaccurate. In fact, we believe MC access counters are but a small piece of the architectural performance profiling support needed for a NoC-based OS. Other programmable counters can keep track of contention for different NoC links, access counts for distributed L2 cache banks, and occupancy levels of different L2 cache banks. Such counters can allow the NoC-based OS to make intelligent, contention- and locality-aware execution placement and migration decisions.

5. LIMITATIONS & FUTURE WORK

Given our results, we describe two limitations imposed by our statistics collection mechanism and propose some schemes that mitigate these effects. We also consider other improvements to Network-on-Chip systems enabled as a direct result of our work.

5.1 Statistics Collection Limitations

We find two primary problems imposed by the way we collect MC usage data. First, there is a fairly substantial performance overhead from our prototype. For a single-threaded instance of our microbenchmark, the median execution time increases from 5.3 seconds to 5.8 seconds (a 9% performance degradation). In many cases, our assignment algorithm is capable of winning back much of that performance loss (and it follows that a more optimal statistics collection scheme would see a strict gain), but the degradation is still substantial.

Second, although our data relocation algorithm is not dependent on high-precision usage data, the MC usage statistics provide an approximate view of how physical memory is used by an application. It more effectively captures the size of the working set on an MC and uses that as a proxy for utilization. However, this approach does not capture a full picture of memory access *patterns*—for instance, a workload with high temporal locality may have a lesser need for low memory latency than a workload prone to cache thrashing.

One way to resolve the latter problem, albeit at the potential expense of more overhead, would be to use the TILE architecture's software-loaded TLB functionality to more directly probe page access. By artificially shrinking the size of the TLB, we can cause TLB misses to act as a more accurate proxy for cache misses, providing more precise data to an application that wishes to optimize the system. This would be an effective means for further experimentation on a live system, though such a drastic reduction of TLB capacity would almost certainly not be appropriate for a production system, thus calling for memory access instrumentation in hardware. A simple effort would be to augment each core with a counter that increments on each access to an MC (and how many cycles the core has spent blocked on access to that MC). While this would enable simple application locality optimizations, it would not provide detailed informa-

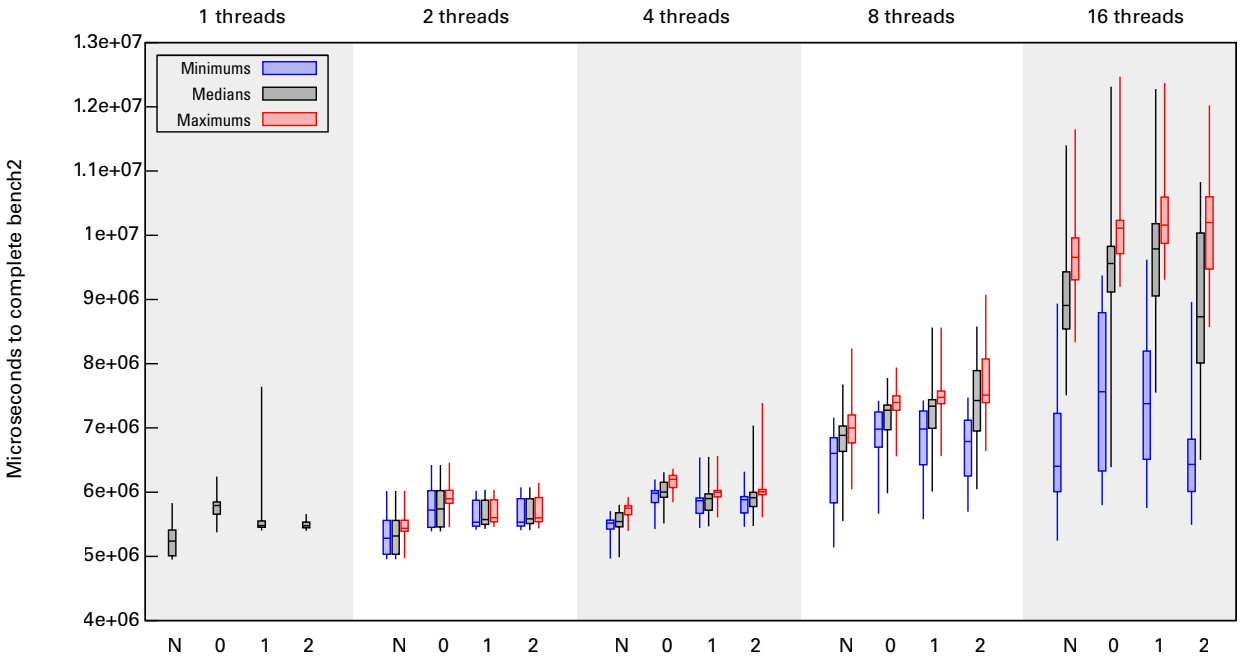

Figure 5: Runtime benchmark performance(us). Each group of 4 box plots corresponds to 1, 2, 4, 8, or 16 instances of the benchmark being executed. Within each group, we compare *baseline*(N), *overhead*(0), *weighted*(1), and *brute force*(2). Lower is better.

tion about utilization of individual pages within the working set.

For the full benefit of software-optimized application placement, it may be useful to collect statistics on specific pages that frequently miss in cache. An individual CAM updated in parallel with memory requests could easily store this information. A more advanced scheme could attach this data to the TLB and write back to a buffer alongside the page table.

5.2 Related Challenges

Precise collection of memory access data opens up new opportunities for improving the performance of Network-on-Chip systems. We discuss three concepts thus motivated.

Optimizing for programmable NUCA environments: While we focused on variable memory access latency in this paper, we recognize that other causes of variability in data access latency exist. For instance, on the TILE architecture, access latency to cached blocks can be extremely variable, because the third-level cache is distributed across *all* nodes on the many-core chip. This is known as a Non-Uniform Cache Architecture (NUCA). It is possible to pin caches for individual pages to be "homed" to specific nodes, but to date the only application of this feature has been to home caches to the node on which a process executes. This provides the undesirable choice between either a large cache with variable and high access latency, or a low latency cache that is substantially smaller than what could be accomplished by aggregating the cache from multiple nodes.

A memory instrumentation scheme that can count cache misses to each page would aid in choosing more effective placements of caches. Our *weighted geometric* scheme could be similarly used to choose home nodes for each page, homing lesser-used pages further away (but, on average, closer than a random placement would provide). Such a scheme could also influence application placement such that heavy users of cache are placed further from each other, allowing the most physically-adjacent cache to be available to each application.

Optimizing for application performance requirements: In a typical datacenter, multiple heterogeneous applications [20] and virtual machines will share the many-core chip resources. Extensive sharing of on-chip resources (caches, on-chip network, memory bandwidth and capacity) among applications with differing performance, SLA, and QoS requirements creates a new problem for both datacenter and manycore operating systems: how do we design the OS in such a way that each application's requirements are satisfied while the system's throughput is still maximized [21]? The problem is difficult because existing multicore hardware does not provide mechanisms for performance isolation or fairness to different applications/VMs within shared hardware resources [10, 16, 17]. Some applications can be starved or denied service for long time periods [16]. The priorities set by the OS cannot be enforced by the shared hardware resources because these resources are not designed to be aware of application priorities. And, application performance becomes very dependent on what other applications are running on the same chip [9, 16, 17]. The solution we present in this paper is currently oblivious to such heterogeneity, but can be adapted to account for priorities and weighting application placement preferences accordingly. Our technique could, thus, be used to improve memory access latency for more demanding (and, therefore, higher priority) applications, while simultaneously allowing

lower priority (e.g., management) tasks to continue to execute on the same network.

Improving power efficiency: Although we primarily focused on raw *performance*, real environments are increasingly becoming bound by *power density* as a limit on how many systems can be simultaneously active in a datacenter. In that light, a useful measurement is not only raw performance per mm^2, but also performance per Watt. Although our experimental setup was not capable of such measurements, we believe that our results should show an improvement in power as well. As transistors shrink, NoC devices will move from being dominated by dynamic power on gates to being dominated by static power and wire capacitance power. The latter is proportional to Manhattan interconnect distances, which we track and optimize for.

6. CONCLUSION

Variable Network-on-Chip memory access latencies can significantly affect application performance in NoC-based manycore systems in ways fundamentally different from traditional NUMA interconnect systems. This paper proposes explicit scheduling of execution threads so as to mitigate memory access latency variation. We depart from previous work by migrating execution closer to the data it uses as opposed to inter-controller memory page migration. In our implementation, we rely on instrumentation of the virtual memory subsystem for the purposes of memory controller access statistics collection on a per-address space basis. This approach helps motivate the need for programmable, low-overhead, per-process memory controller access counters, which will substantially reduce the overhead associated with execution behavior measurement. By reporting on our experience constructing a solution without them, we show that their availability will enable the OS scheduler to assess threads' runtime dependence on specific memory nodes, adjust their placement accordingly, and improve their performance as a result.

7. ACKNOWLEDGMENTS

The authors wish to thank Tilera Corporation, for generously providing a TilePro64 board for research at Carnegie Mellon, and Professor David A. Eckhardt for lab space and hardware hosting. We thank the member companies of the PDL Consortium (Actifio, APC, EMC, Emulex, Facebook, Fusion-IO, Google, HP, Hitachi, Huawei, Intel, Microsoft, NEC Labs, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, VMware, Western Digital) for their interest, insights, feedback, and support. This research is supported in part by Intel, as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC), and by an NSERC Postgraduate Fellowship.

8. REFERENCES

[1] TILEPro processor family: TILEPro64 overview, 2013. `http://www.tilera.com/products/processors/TILEPro_Family`.

[2] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers. *PACT*, 2010.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. In *SOSP*, 2009.

[4] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *OSDI*, 2008.

[5] S. Boyd-Wickizer, A. Clements, Y. Yandong Mao, A. Pesterev, M. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *OSDI*, 2010.

[6] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *ASPLOS*, 1994.

[7] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. *DAC-38*, 2001.

[8] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *Proc. of HPCA'13*, 2013.

[9] R. Das, O. Mutlu, T. Moscibroda, and C. Das. Application-Aware Prioritization Mechanisms for On-Chip Networks. *MICRO-42*, 2009.

[10] A. Fedorova, M. I. Seltzer, and M. D. Smith. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In *PACT*, 2007.

[11] B. Grot, S. Keckler, and O. Mutlu. Preemptive Virtual Clock: A Flexible, Efficient, and Cost-effective QOS Scheme for Networks-on-Chip. *MICRO-42*, 2009.

[12] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *International Symposium on Microarchitecture (MICRO-43)*, 2010.

[13] J. Lee, M. Ng, and K. Asanovic. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. *ISCA-35*, 2008.

[14] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[15] J. Mogul. SIGOPS to SIGARCH: "Now it's our turn to push you around". In *OSDI, Research Vision Session*, 2010.

[16] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security Symposium*, 2007.

[17] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *International Symposium on Microarchitecture (MICRO-39)*, 2006.

[18] K. J. Nesbit, J. Laudon, and J. E.Smith. Virtual Private Caches. In *International Symposium on Computer Architecture (ISCA-34)*, 2007.

[19] G. Nychis, C. Fallin, T. Moscibroda, and O. Mutlu. Next Generation On-Chip Networks: What Kind of Congestion Control Do We Need? *HotNets*, 2010.

[20] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. of ACM SoCC*, 2012.

[21] A. Tumanov, J. Cipar, M. A. Kozuch, and G. R. Ganger. alsched: algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proc. of ACM SoCC*, 2012.

[22] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43:76–85, April 2009.

[23] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*, 2010.