
DATA MARSHALING FOR MULTICORE SYSTEMS

DIVIDING A PROGRAM INTO SEGMENTS AND EXECUTING EACH SEGMENT AT THE CORE BEST SUITED TO RUN IT CAN IMPROVE PERFORMANCE AND SAVE POWER. WHEN CONSECUTIVE SEGMENTS RUN ON DIFFERENT CORES, ACCESSES TO INTERSEGMENT DATA INCUR CACHE MISSES. DATA MARSHALING ELIMINATES SUCH CACHE MISSES BY IDENTIFYING AND MARSHALING THE NECESSARY INTERSEGMENT DATA WHEN A SEGMENT IS SHIPPED TO A REMOTE CORE.

M. Aater Suleman
University of Texas
at Austin

Onur Mutlu
Carnegie Mellon
University

**José A. Joao
Khubaib**

Yale N. Patt
University of Texas
at Austin

••••• Several parallel programming paradigms split the code into segments that run on different cores. Such models include Accelerated Critical Sections,¹ producer-consumer pipelines, computation spreading,² Cilk,³ and Apple's Grand Central Dispatch.⁴ We refer to these models collectively as *staged execution*. In staged execution, each segment is executed on its home core. The home core of a segment is the core best suited to run that segment. When a core encounters a new segment, it ships the segment to its home core. The home core buffers the execution request from the requesting core and processes it in turn. The criteria commonly used to choose a segment's home core include performance criticality, functionality, and data requirements (for example, critical sections that are on the program's critical path run best at the fastest core).¹

Although staged execution can improve locality,^{1,5,6} increase parallelism,⁷ and leverage heterogeneous cores,⁸ it yields only a limited performance benefit when a segment accesses intersegment data—that is, data that the previous segment generated. Because each segment runs on a different core,

accessing intersegment data generates cache misses, which reduces performance.

To reduce cache misses for intersegment data, we propose Data Marshaling, a mechanism to identify and send the intersegment data required by a segment to its home core. We used a key observation to design Data Marshaling: the generator set (the set of instructions that generate intersegment data) stays constant throughout the execution. The compiler uses profiling to identify the generator set, and the hardware marshals the data these instructions have written to the next segment's home core. Our approach has three advantages. First, because the generator instructions are statically known, Data Marshaling doesn't require any training. Second, the requesting core starts marshaling the intersegment cache lines as soon as the execution request is sent to the home core, which makes data transfer timely. Third, because Data Marshaling identifies intersegment data as any data written by a generator instruction, Data Marshaling can marshal any arbitrary sequence of cache lines. Data Marshaling requires only a modest 96 bytes of storage per core, one new ISA instruction, and compiler support. (For information on

Research related to Data Marshaling

Researchers have explored the areas of hardware prefetching and operating system and compiler techniques to improve locality. We briefly describe the past work most relevant to Data Marshaling.

Hardware prefetching

We can broadly classify hardware prefetchers as prefetchers that target regular (stream and stride) memory access patterns¹ and those that target irregular memory access patterns.² Prefetchers that handle only regular data can't capture misses for intersegment data because intersegment cache lines don't follow a regular stream and stride pattern, but instead are scattered in memory. Furthermore, prefetchers are ill-suited for prefetching intersegment data because the number of cache misses required for training such prefetchers is often more than all of the intersegment data (an average of 5 cache lines in Accelerated Critical Sections and 6.8 cache lines in pipeline workloads). Thus, by the time prefetching begins, most of the cache misses have already been incurred.

Data Marshaling doesn't have these disadvantages. It requires minimal on-chip storage, can marshal any arbitrary sequence of intersegment cache lines, and starts marshaling as soon as the next code segment ships to its home core, without requiring any training. Our baseline uses an aggressive stream prefetcher³ and the reported improvements are on top of this aggressive prefetcher.

Reducing cache misses

Yang et al. show that pushing (versus pulling) cache lines to the core can save off-chip cache misses for linked data structures.⁴ In contrast, Data Marshaling saves on-chip misses for any sequence of intersegment data. Hossain et al. propose DDCache, where the producer pushes a cache line to all the sharers of the line when one sharer requests the line.⁵ DDCache is orthogonal to Data Marshaling; the former improves shared data locality, while the latter improves private (intersegment) data locality.

Other proposals improve shared data locality by inserting software prefetch instructions before the critical section.^{6,7} Such a scheme can't work for intersegment data because prefetch instructions must execute as part of the next code segment, close to the actual use of the data, which likely makes the prefetches untimely.

Other related work

Remote Procedure Calls, used in networking, require the programmers to marshal the input data with the RPC request.⁸ This is similar to Data

Marshaling, which marshals intersegment data to the next code segment's home core. However, unlike RPC, Data Marshaling is solely for performance (that is, not required for correct execution), doesn't require programmer intervention, and applies to instances of staged execution that don't resemble remote procedure calls.

References

1. 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)*, ACM Press, 1990, pp. 364-373.
2. E. Ebrahimi, O. Mutlu, and Y.N. Patt, "Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems," *Proc. IEEE 15th Int'l Symp. High-Performance Computer Architecture (HPCA 09)*, IEEE Press, 2009, pp. 7-17.
3. 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)*, IEEE Press, 2007, pp. 63-74.
4. C.-L. Yang and A.R. Lebeck, "Push vs. Pull: Data Movement for Linked Data Structures," *Proc. 14th Int'l Conf. Supercomputing (ICS 00)*, ACM Press, 2000, pp. 176-186.
5. H. Hossain, S. Dwarkadas, and M.C. Huang, "DDCache: Decoupled and Delegable Cache Data and Metadata," *Proc. 18th Int'l Conf. Parallel Architectures and Compilation Techniques*, IEEE Press, 2009, pp. 227-236, doi:10.1109/PACT.2009.24.
6. P. Trancoso and J. Torrellas, "The Impact of Speeding up Critical Sections with Data Prefetching and Forwarding," *Proc. 1996 Int'l Conf. Parallel Processing*, vol. 3, IEEE Press, pp. 79-86, doi:10.1109/ICPP.1996.538562.
7. P. Ranganathan et al., "The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems," *Proc. 24th Ann. Int'l Symp. Computer Architecture (ISCA 97)*, ACM Press, 1997, pp. 144-156.
8. A.D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Computer Systems*, vol. 2, no. 1, 1984, pp. 39-59.

other proposals, see the "Research related to Data Marshaling" sidebar.)

Data Marshaling is a general framework that can reduce the number of intersegment data misses in any staged execution paradigm. Our evaluation across 12 critical-section-intensive applications shows that Data

Marshaling eliminates almost all intersegment data misses, improving performance by 8.5 percent over an aggressive baseline with Accelerated Critical Sections (ACS) and a state-of-the-art prefetcher, at an area budget equivalent to 16 small cores. Similarly, our evaluation across nine pipelined

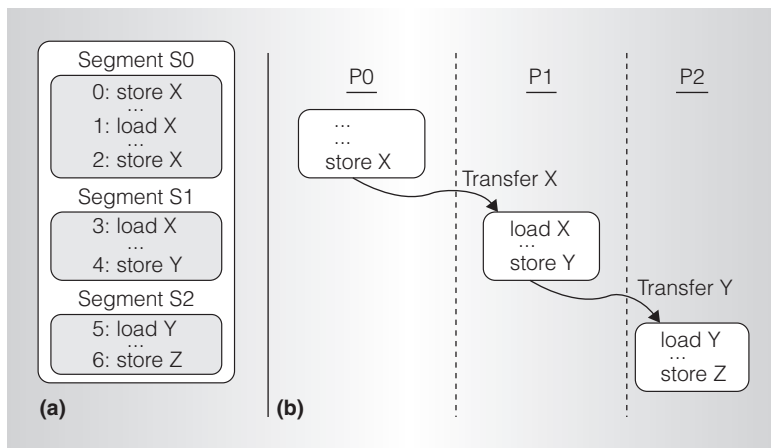


Figure 1. A piece of code with three segments: source code (a) and staged execution (b). Segments S0, S1, and S2 execute on cores P0, P1, and P2 respectively.

workloads shows that Data Marshaling improves performance by 14 percent at an area budget of 16 small cores, achieving 96 percent of the potential performance possible from eliminating intersegment cache misses.

The problem

Figure 1a shows a piece of code with three segments: S0, S1, and S2. S0 is S1's previous segment and S2 is S1's next segment. At the end of S0 is S1's initiation routine, and at the end of S1 is S2's initiation routine. Figure 1b shows this code's execution on a chip multiprocessor (CMP) with three cores (P0 to P2) with the assumption that the home cores of S0, S1, and S2 are P0, P1, and P2, respectively. After completing S0, P0 runs S1's initiation routine, which inserts a request for S1's execution in P1's work queue. Thus, P0 is S1's requesting core. P1 dequeues the entry, executes segment S1, and enqueues a request for S2 in P2's work queue. Processing at P2 is similar to processing at P0 and P1.

We define *intersegment data* as the data a segment requires from its previous segment. In Figure 1, *X* is S1's intersegment data because *X* is produced by S0 and consumed by S1. Intersegment data locality is high in models where consecutive segments run on the same core: the data that the first segment generated remains in the local cache until the next segment can use it. However, in staged execution, accesses to intersegment data

incur cache misses and the data is transferred from the requesting core to the home core via cache coherence. In Figure 1, P1 incurs a cache miss to transfer *X* from P0's cache and P2 incurs a cache miss to transfer *Y* from P1's cache. These cache misses limit staged execution's performance.

Unfortunately, hardware prefetching is ineffective for intersegment data because the data access patterns are irregular and the number of cache lines transferred is too small to train a hardware prefetcher.

Data Marshaling

Let's first define new terminology. The *generator* is the last instruction that modifies a segment's intersegment data. For example, in Figure 1a, the "store" on line 2 is *X*'s generator because no other stores to *X* exist between this store and S1's initiation. The *generator set* is the set of all generator instructions. We observed that a program's generator set is often small and doesn't vary during the program's execution or across input sets. This implies that any instruction previously identified as a generator very likely remains a generator. Thus, a cache line written by a generator is very likely to be intersegment data that the following segment requires—and therefore a good candidate for data marshaling. From this observation, we assume that every cache line written by an instruction in the generator set is intersegment data and will be needed during the execution of the next segment. We can split Data Marshaling's functionality into three parts.

- *Identification.* Data Marshaling uses a profiling algorithm to identify the generator set. The profiling algorithm executes the program as a single thread and, for every cache line, records the last instruction that modified the cache line. When an instruction accesses a cache line that was last modified by an instruction in the previous segment, the profiling algorithm adds the last instruction to write to that cache line to the generator set. The compiler marks the generator instructions using a *Generator* prefix.
- *Recording.* We augment each core with a Data Marshaling Unit. The DMU

contains a circular, combining buffer called Marshal Buffer. When the core retires a generator instruction, the physical address of the cache line written by the instruction (assumed to be intersegment data) is enqueued in the Marshal Buffer.

- *Marshaling.* When the core has finished executing the current segment, it executes a new *Marshal* instruction that triggers data marshaling. For each line address in the Marshal Buffer, the DMU accesses the local L2 cache and, if the coherence state is exclusive or modified, transfers the cache line to the fill buffer of the home core's L2 cache. Data Marshaling marshals data into L2 caches (not L1) because our experiments show that not saving L1 misses is a better trade-off than allowing pollution of the L1 caches with marshaled data. The Data Marshaling transaction acts like an unsolicited cache-to-cache transfer (we describe this in detail in our ISCA 2010 paper⁹).

Data Marshaling's main overhead is the storage that the Marshal Buffer requires, which amounts to 96 Bytes per core—16 (6-Byte) entries. Data Marshaling can increase the instruction footprint because it adds the Generator prefixes and Marshal instructions. However, this increase is only marginal. Overall, Data Marshaling has a low overhead and its performance benefit is high, as we will show next.

Evaluation

Although Data Marshaling is a general mechanism suitable for any staged execution paradigm, we designed and evaluated two concrete examples: ACS and pipeline parallelism. ACS accelerates critical sections by executing them at a remote large, high-performance core. Data Marshaling can reduce the number of cache misses of the large core by marshaling the required data from the small core to the large core. In pipeline parallelism, all data that is moved from one stage to the next is intersegment data, and Data Marshaling can marshal it.

We evaluated ACS on an asymmetric CMP (ACMP) with one large core and

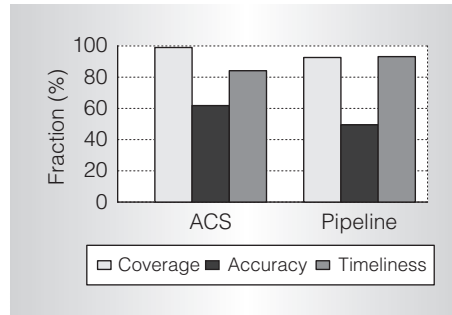


Figure 2. Data Marshaling's coverage, accuracy, and timeliness.

many small cores, and we evaluated the pipeline workloads on a symmetric CMP (SCMP) with all small cores. We assumed that a small core takes unit area and the large core takes as much area as four small cores. Our ISCA 2010 paper includes details of our methodology.⁹

Coverage, accuracy, and timeliness

We measure the effectiveness of Data Marshaling in reducing intersegment data misses using three metrics: coverage, accuracy, and timeliness. Coverage is the fraction of intersegment data cache lines that Data Marshaling identified. Accuracy is the fraction of marshaled lines that the home core actually uses. Timeliness is the fraction of useful marshaled cache lines that reach the home core before they're needed. We don't consider a marshaled cache line that's in transit when the home core requests it to be timely according to this definition, but it can provide performance benefit by reducing L2 miss latency.

Figure 2 shows the coverage of Data Marshaling for ACS and pipeline workloads. Data Marshaling will likely detect all intersegment lines because it optimistically assumes that every instruction that once generated intersegment data always generates intersegment data. As a result, its coverage is almost 100 percent for ACS and above 90 percent for pipeline parallelism.

Figure 2 also shows Data Marshaling's accuracy. Because Data Marshaling optimistically assumes that all lines that the generator instructions modify are intersegment data, it could have low accuracy. We find that, despite our optimistic assumptions, more than 50 percent of the marshaled cache

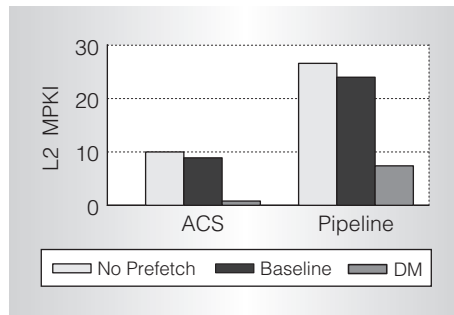


Figure 3. L2 cache misses per kilo-instruction (MPKI) due to intersegment data. (DM is Data Marshaling.)

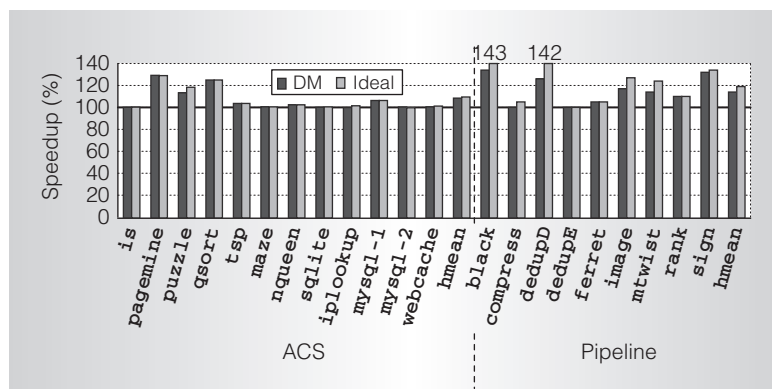


Figure 4. Data Marshaling's performance at an area budget of 16 small cores.

lines are useful. Marshaling useless lines can cause cache pollution and interconnect contention only if the number of marshaled cache lines is high. We find that Data Marshaling transfers an average of only 5 cache lines, more than 50 percent of which are useful on average, which causes minimal cache pollution and interconnect contention.

Figure 2 also shows Data Marshaling's timeliness. We find that more than 80 percent of the useful marshaled cache lines are timely. Because coverage is close to 100 percent, timeliness directly corresponds to the reduction in intersegment data cache misses.

Reduction in intersegment data cache misses

Figure 3 shows the number of L2 cache misses per kilo-instruction (MPKI) for intersegment data. We compare the MPKI for a system with hardware prefetching turned off (no prefetch), our baseline system with

hardware prefetching turned on but without Data Marshaling (baseline), and a system with both prefetching and Data Marshaling. Hardware prefetching reduces the number of cache misses for intersegment data only marginally in both ACS and pipeline workloads. In contrast, Data Marshaling substantially reduces the number of these cache misses. For ACS, Data Marshaling almost eliminates these misses because coverage and timeliness are both high. In pipeline workloads, Data Marshaling reduces the number of cache misses for intersegment data by 69 percent.

Performance

Figure 4 shows the speedup over the baseline (without Data Marshaling) for Data Marshaling and an unrealistic configuration (Ideal) that artificially converts all intersegment cache misses into hits. We evaluate all three configurations at an equal area budget of 16 small cores. For ACS workloads, Data Marshaling provides the highest speedup on pagemine, puzzle, and qsort because these workloads have high contention for critical sections; thus, Data Marshaling can speed up the critical path by saving intersegment data cache misses. Data Marshaling's speedup is 8.5 percent on average when the area budget is 16 small cores. Data Marshaling's performance is within 1 percent of the Ideal performance, showing that Data Marshaling achieves almost all of its potential benefit.

The execution time of a pipelined program is always dictated by its slowest stage. Thus, Data Marshaling's impact on overall performance depends on how much it speeds up the slowest stage. At 16 cores, Data Marshaling improves performance by more than 20 percent in five out of nine workloads. On average, Data Marshaling provides a 14-percent speedup over the baseline, which is 96 percent of the potential.

Figure 5 shows Data Marshaling's average speedup over baseline as the number of cores increases. For ACS, the contention for critical sections increases with the number of cores, which increases the benefit of eliminating cache misses inside critical sections. Consequently, Data Marshaling's benefit increases as the number of cores increases

from 16 to 32 and 64. For pipeline workloads, Data Marshaling's speedup increases to 16 percent with a larger area budget of 32, which is higher than its speedup at 16 cores (14 percent).

Other studies and results

Our ISCA 2010 paper further analyzes Data Marshaling's performance trade-offs and evaluates how its performance benefit is sensitive to important architectural parameters.⁹ We show that Data Marshaling's benefit increases if we increase the interconnect hop latency or L2 cache size. Both of these parameters increase the effect of intersegment data cache misses on overall performance, thereby making Data Marshaling more useful. Data Marshaling becomes even more effective if we increase the number of entries in the Marshal Buffer.

We conclude that Data Marshaling is effective for improving both ACS and pipelined workload performance, with increasing benefit as the number of cores increases.

Contributions and impact

Data Marshaling reduces the number of cache misses incurred during remote execution. The overhead of these cache misses is the single most important cost associated with remote execution, and it's often why programmers avoid remote execution despite its well-known benefits. Fundamentally, programmers use remote execution only if

$$\text{Benefit of remote execution} > \text{Migration cost}$$

Data Marshaling significantly reduces migration cost and can therefore make remote execution feasible in many cases where it had been infeasible. We envision that Data Marshaling will significantly impact parallel programming models, CMP architectures, and future academic and industrial research.

Impact on parallel programming paradigms

Today's parallel programming paradigms are designed to improve intersegment data locality. For example, the most common programming model, work sharing, always executes all instructions in a loop iteration at the same core. Even task-parallel models, which split work into finer-grain tasks, operate such that when a thread finishes a

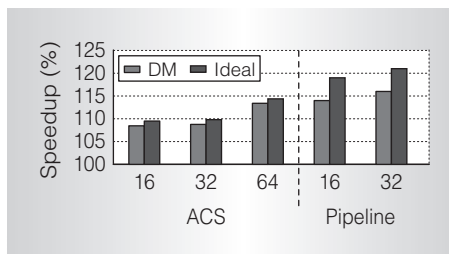


Figure 5. Data Marshaling's performance as the area budget increases.

segment, it enqueues the next segment in the local core's work queue, even though a more suitable core for the segment might be available. Enqueueing is done at the local core instead of a more suitable remote core that might be free in order to exploit the fact that inputs to this new segment are likely to be present in the local cache. If the segment were shipped to a remote core, these inputs would cause cache misses. Data Marshaling avoids these cache misses, thus enabling a change in the programmer's mindset regarding intersegment data transfers. The programmer can more aggressively decide to execute the next segment remotely in a core that is more suitable (instead of the local core) because the effect of intersegment data transfer on performance is not high. We therefore expect Data Marshaling to make future parallel programming more data-centric and fine grained.

Effect on data-centric parallel programming. Future paradigms can focus less on improving the *intersegment* data locality and more on improving the *intra-segment* data locality. Consequently, programmers can "specialize" even homogeneous cores for certain segments. For example, programmers can write code in a data-centric style where data is pinned to a core and all segments accessing that data execute on that core. Such a model can improve locality and avoid data synchronization, eliminating the overhead of expensive lock acquire and release operations.

Effect on fine-grain parallel programming. A major overhead of fine-grain parallelism is communication of intersegment data. Data Marshaling largely lowers this overhead,

which can help programmers and compilers exploit parallelism at a finer granularity. Finer-grain segments can unveil more parallelism, thereby improving system performance.

Effect on adaptive staged execution programs. Another advantage of Data Marshaling is that it can allow hardware to estimate the cost of a task migration by knowing the number of cache lines it will need to migrate for each task (that is, the number of lines in the Marshal Buffer). This can not only enable software to make informed task migration decisions but also help the hardware or the runtime system make task migration decisions without programmer intervention. As a result, programs can be more adaptive, and programmers won't have to decide whether to migrate a task.

Impact on CMP architectures

By reducing the cost of task migration, data marshaling helps programs better utilize heterogeneous cores. For example, we showed that Data Marshaling improved performance of ACS, an asymmetric multicore architecture. Hardware manufacturers can thus tile heterogeneous processing cores specialized for particular code segments and expect software to exploit them.

Data Marshaling can also broaden the spectrum of heterogeneous or special-purpose cores and units that chip designers generally consider for on-chip integration. Without Data Marshaling, the overhead of migrating work is high—so, chip designers consider a unit for on-chip integration only if it provides orders of magnitude improved performance and power over general-purpose cores. Data Marshaling relaxes this constraint by lowering the migration overhead, allowing hardware and software to experiment with new special-purpose units (such as field-programmable gate array [FPGA]-like programmable accelerators) and exploit untapped opportunities.

By reducing the cost of remote execution, Data Marshaling can further encourage sharing of special-purpose units. Chip manufacturers can provide only a few special-purpose units of each type and cores can efficiently share these remote units using Data Marshaling to transfer the data back and forth.

We believe that Data Marshaling will encourage the following new research topics:

- Because Data Marshaling lets software direct the hardware to marshal a cache line to any core in the system, future research can explore how to exploit this capability to speed up interthread communication—even in nonstaged execution paradigms.
- Future research can explore other uses of Data Marshaling's hardware support, such as pushing data to a core that's polling.
- Data Marshaling enables fine-grain remote execution, so future research can explore what kind of units can use this ability and how software can leverage these units.
- Because Data Marshaling lets runtime systems estimate the cost of migrating a code segment, future research may be able to develop runtime systems that can adapt staged execution programs using this feedback.
- We've shown that Data Marshaling is effective on two execution paradigms; future research can explore using Data Marshaling for any paradigm that resembles staged execution.
- We implemented Data Marshaling using a combined hardware/software approach that requires support from the compiler and the instruction set architecture (ISA); future research can investigate a hardware-only approach that can work with existing programs without requiring recompilation.
- Although we used a simple, optimistic algorithm for identifying intersegment data, we expect future research to result in more accurate algorithms to improve Data Marshaling.
- Although we only investigate marshaling cache lines between consecutive segments, future research can remove this constraint.

Data Marshaling improves locality of intersegment data in staged execution, thereby overcoming a major limitation of staged execution. We believe that Data Marshaling

is a promising approach and we hope that it will influence future CMP designs and enable future research in new parallel programming paradigms.

MICRO

Acknowledgments

We thank the HPS Research Group for its feedback and suggestions. We also thank the Cockrell Foundation, Intel, and the Carnegie Mellon University CyLab for their support, along with the Texas Advanced Computing Center for providing computing resources. M. Aater Suleman was supported by an Intel PhD fellowship.

References

1. M.A. Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," *Proc. 14th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (Asplos 09)*, ACM Press, 2009, pp. 253-264.
2. K. Chakraborty, P.M. Wells, and G.S. Sohi, "Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-The-Fly," *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (Asplos 06)*, ACM Press, 2006, pp. 283-292.
3. R.D. Blumofe et al., "Cilk: An Efficient Multithreaded Runtime System," *Proc. 5th ACM Sigplan Symp. Principles and Practice of Parallel Programming*, ACM Press, 1995, pp. 201-216.
4. "Grand Central Dispatch," tech. brief, Apple, 2009; http://images.apple.com/macosx/technology/docs/GrandCentral_TB_brief_20090903.pdf.
5. S. Boyd-Wickizer, R. Morris, and M.F. Kaashoek, "Reinventing Scheduling for Multicore Systems," *Proc. 12th Conf. Hot Topics in Operating Systems*, Usenix Assoc., 2009, no. 21; www.usenix.org/event/hotos09/tech/full_papers/boyd-wickizer/boyd-wickizer.pdf.
6. S. Harizopoulos and A. Ailamaki, "StagedDB: Designing Database Servers for Modern Hardware," *IEEE Data Eng. Bull.*, vol. 28, no. 2, 2005, pp. 11-16.
7. W. Thies, M. Karczarek, and S.P. Amarasinghe, "Streamit: A Language for Streaming Applications," *Proc. 11th Int'l Conf.*

Compiler Construction (CC 02), LNCS 2304, Springer, 2002, pp. 179-196.

8. M. Annavaram, E. Grochowski, and J. Shen, "Mitigating Amdahl's Law through EPI Throttling," *Proc. 32nd Ann. Int'l Symp. Computer Architecture (ISCA 05)*, IEEE CS Press, 2005, pp. 298-309.
9. M.A. Suleman et al., "Data Marshaling for Multi-Core Architectures," *Proc. 37th Ann. Int'l Symp. Computer Architecture (ISCA 10)*, ACM Press, 2010, pp. 441-450.

M. Aater Suleman is a computer architect at Intel. His research interests include chip multiprocessor architectures and parallel programming. He has a PhD in electrical and computer engineering from the University of Texas at Austin. He's a member of IEEE, the ACM, and Eta Kappa Nu.

Onur Mutlu is an assistant professor in the Electrical and Computer Engineering Department at Carnegie Mellon University. His research interests include computer architecture and systems. He has a PhD in electrical and computer engineering from the University of Texas at Austin. He's a member of IEEE and the ACM.

José A. Joao is a doctoral student in Electrical and Computer Engineering at the University of Texas at Austin. His research interests include high-performance microarchitectures and compiler-runtime-architecture interaction. He has a master's degree in electrical and computer engineering from the University of Texas at Austin. He's a student member of IEEE and the ACM.

Khubaib is a doctoral student in electrical and computer engineering at the University of Texas at Austin. His research interests include microarchitecture and high-performance memory system design. He has a master's degree in Electrical and Computer Engineering from the University of Texas at Austin. He's a member of IEEE and the ACM.

Yale N. Patt is the Ernest Cockrell, Jr. Centennial Chair in Engineering at the University of Texas at Austin. His research interests include harnessing the expected

fruits of future process technology into more effective microarchitectures for future microprocessors. He has a PhD in electrical engineering from Stanford University. He coauthored *Introduction to Computing Systems: From Bits and Gates to C and Beyond* (McGraw-Hill, 2003). He's a Fellow of IEEE and the ACM.

Direct questions and comments to M. Aater Suleman, 11932 Rosethorn Dr., Austin, TX 78758; suleman@hps.utexas.edu.


cn Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.

Running in Circles Looking for a Great Computer Job or Hire?



Make the Connection - IEEE Computer Society Jobs is the best niche employment source for computer science and engineering jobs, with hundreds of jobs viewed by thousands of the finest scientists each month - **in Computer magazine and/or online!**

- > Software Engineer
- > Member of Technical Staff
- > Computer Scientist
- > Dean/Professor/Instructor
- > Postdoctoral Researcher
- > Design Engineer
- > Consultant

IEEE  computer society | **JOBS**

<http://www.computer.org/jobs>

IEEE Computer Society Jobs is part of the *Physics Today* Career Network, a niche job board network for the physical sciences and engineering disciplines. Jobs and resumes are shared with four partner job boards - *Physics Today* Jobs and the American Association of Physics Teachers (AAPT), American Physical Society (APS), and AVS: Science and Technology of Materials, Interfaces and Processing Career Centers.