Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2010-019 April 17, 2010

Instruction-Level Execution Migration

Omer Khan, Mieszko Lis, and Srinivas Devadas

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Instruction-Level Execution Migration

Omer Khan[1], Mieszko Lis[1], and Srinivas Devadas
Massachusetts Institute of Technology, Cambridge, MA

***Abstract*—We introduce the Execution Migration Machine (EM^2), a novel data-centric multicore memory system architecture based on computation migration. Unlike traditional distributed memory multicores, which rely on complex cache coherence protocols to move the data to the core where the computation is taking place, our scheme *always* moves the computation to the core where the data resides. By doing away with the cache coherence protocol, we are able to boost the effectiveness of per-core caches while drastically reducing hardware complexity.**

To evaluate the potential of EM^2 architectures, we developed a series of PIN/Graphite-based models of an EM^2 multicore with 64 x86 cores and, under some simplifying assumptions (a timing model restricted to data memory performance, no instruction cache modeling, high-bandwidth fixed-latency interconnect allowing concurrent migrations), compared them against corresponding directory-based cache-coherent architecture models. We justify our assumptions and show that our conclusions are valid even if our assumptions are removed. Experimental results on a range of SPLASH-2 and PARSEC benchmarks indicate that EM^2 can significantly improve per-core cache performance, decreasing overall miss rates by as much as 84% and reducing average memory latency by up to 58%.

I. Introduction

In the last few years, the steady increases in processor performance obtainable from higher and higher clock frequencies have come to a dramatic halt: there is simply no cost-effective way to dissipate so much power. Instead, recent development has favored multicore parallelism: commodity processors with four or even eight cores on a single die have become common, and existing technology permits many more; indeed, general-purpose single-die multiprocessors with as many as 64 cores are already commercially available [1]. Even larger multicores have been built [2, 3], and pundits confidently predict thousands of cores per die by the end of the decade [4]. Quite simply, multicores *scale*.

Designing a scalable memory subsystem for a multicore, however, remains a major concern. Increasing the number of concurrent threads requires a large aggregate memory bandwidth, but off-chip memory bandwidth is severely constrained by the number of pins on the package: a conundrum known as the *off-chip memory bandwidth wall* [4, 5]. To address this problem, multicores integrate large private and shared caches on chip: the hope is that large caches can hold the working sets of the active threads, thereby reducing the number of off-chip memory accesses. Private caches, however, require cache coherence, and shared caches do not scale beyond a few cores: even today, the large 32MB last-level cache in recent Intel® 8-core processors is split physically into tiles distributed across the chip [6], and accessing remote cache lines is significantly slower than accessing local ones.

Since shared caches do not scale, many private caches are the only practical option in large-scale multicores. In practice, this means some form of memory coherence, as the success of alternate programming paradigms based on exposing core-to-core communication to the programmer has been limited to scientific computing and other niches where performance or power considerations warrant the increased programming complexity. The key question, then, is: how can we provide the illusion of shared memory in a way that scales to thousands of cores?

Bus-based cache coherence, which provides the illusion of a single, consistent memory space, clearly does not scale beyond a few cores. Directory-based cache coherence is not subject to the electrical limitations of buses, but requires complex states and protocols for efficiency even in today's relatively small multicores. Worse yet, directory-based protocols can contribute significantly to the already costly delays of accessing off-chip memory because data replication limits the efficient use of cache resources. Finally, the area costs of keeping directory entries are a large burden: if most of the directory is kept in off-chip memory, accesses will be too slow, but if the directory is stored in a fast on-chip memory, evictions from the necessarily limited directory cause thrashing in the per-core caches, also decreasing performance.

Yet on-chip multicores provide a tremendous opportunity for optimization in the form of abundant interconnect bandwidth. Even existing electrical on-chip interconnect networks offer terabits per second of cross-

[1] Equal contributors.

section bandwidth [7] with latencies growing with the diameter of the network (i.e., as the square root of the core count in meshes), and emerging 3D interconnect technologies enable high-bandwidth, low-latency on-chip networks [8]. Optical interconnect technology, which offers high point-to-point bandwidth at little latency and with low power, is fast approaching miniaturization comparable to silicon circuits, with complete ring lasers no larger than 20µm^2 [9]; multicore architectures featuring an on-chip optical interconnect have been proposed [10, 11], but have so far been based on traditional cache-coherent memory architectures.

In this manuscript, we take the view that future multicore architectures will feature thousands of computation cores and copious inter-core bandwidth. To take advantage of this, we propose to do away altogether with the latencies and implementation complexities of cache coherence protocols. Instead, we argue, each core should be responsible for caching a segment of the address space; when the thread running on a given core refers to an address resident in some other core, the computation itself must move by having the two cores swap execution contexts. Supported by extensive simulations running the SPLASH-2 and PARSEC benchmark suites on both our architecture and a traditional cache-coherence architecture with an equivalent interconnect network, we make the case that the complete absence of data sharing among caches, far from limiting performance, actually *improves* cache efficiency by evicting fewer cache lines on average and increasing the effective size of the combined on-chip cache.

The novel contributions of this paper are:

1. We introduce execution migration at the instruction level, a simple architecture that provides a coherent, sequentially consistent view of memory without the need for a cache coherence protocol.
2. We evaluate our scheme on actual applications in a current x86-based shared memory system: our functional memory subsystem model is built on PIN/Graphite [12, 13] and runs a set of SPLASH-2 [14] and PARSEC [15] benchmarks with the correct output.
3. We show that, assuming a high-bandwidth, low-latency interconnect, on-chip cache hierarchy miss rate under execution migration improves many-fold (e.g., from 4.4% to 0.5% with 80KB of caches per core), and, as a result, average memory access latencies significantly improve (e.g., 16.5 to 6 cycles/access with 80KB of caches per core).
4. We describe how, provided a scalable interconnect network, EM^2 elegantly scales to thousands of cores while significantly reducing silicon area compared to a traditional cache-coherent design.

Benchmark	Low Performance Network	High Performance Network
Blackscholes	≤ 80 KB	EM^2 outperforms cache coherence at all cache sizes we tested
LU (non-contiguous)	≤ 32 KB	
LU (contiguous)	Never	
Swaptions	≤ 32 KB	
Water	≤ 144 KB	

Table 1: Per-core cache sizes at which EM^2 outperforms cache coherence for a low-performance interconnect network (packet latency of 50 cycles) and a high-performance network (packet latency of 5 cycles). With a slow network, EM^2 outperforms cache-coherent designs only when small cache sizes combined with data sharing cause high miss rates in the latter, but with a fast network, EM^2 performed better on all of the cache sizes we tested.

The performance of EM^2 is tightly coupled to available network resources and, given sufficiently large caches, a directory-based cache-coherent architecture outperforms EM^2 on a low-performance, high-latency network even though it suffers more cache hierarchy misses. Table 1 shows the minimum cache sizes needed by an equivalent directory-based cache-coherent architecture to outperform EM^2 on a few sample benchmarks. With a low-performance network (50-cycle per-message latency), context migrations in EM^2 are expensive, and large per-core caches allow the cache coherent architecture to reduce main memory accesses and perform better; with a high-performance network (5-cycle per-message latency), however, the significant reduction in cache hierarchy misses in EM^2 balances out the cost of context migrations and EM^2 outperformed cache-coherent architecture with all of the cache sizes we tested. For example, the *swaptions* benchmark performed better under EM^2 when caches were 32KB or less per core; on a high-performance network EM^2 always performed better.

The remainder of this paper is organized as follows: Section II below reviews related research; in Section III we delineate the operation of execution migration, and in Section IV describe the effects and architectural tradeoffs versus directory-based cache coherence. Section V outlines our experimental methodology and Section VI compares real-world application performance of execution migration against a traditional cache-coherence scheme using a detailed architectural simulator. Section VII offers concluding remarks and outlines future research.

II. RELATED WORK

A. Computation migration

Migrating computation to the locus of the data is not itself a novel idea. Hector Garcia-Molina in 1984 introduced the idea of *moving processor to data* in memory bound architectures [16]. In recent years migrating execution context has re-emerged in the context of single-chip multicores. Michaud shows the benefits of using execution migration to improve the overall on-chip cache capacity and utilizes this for migrating selective sequential programs to improve performance [17]. Computation spreading [18] splits thread code into segments and assigns cores responsible for different segments, and execution is migrated to improve code locality. Kandemir presents a data migration algorithm to address the data placement problem in the presence of non-uniform memory accesses within a traditional cache coherence protocol [19]. This work attempts to find an optimal data placement for cache lines. A compile-time program transformation based migration scheme is proposed in [20] that attempts to improve remote data access. Migration is used to move part of the current thread to the processor where the data resides, thus making the thread portion local. This work shows that computation migration puts far less stress on the network than shared memory counterpart. Our proposed execution migration machine is unique among the previous proposed works because we completely abandon data sharing (and therefore do away with cache coherence protocols). Instead, we propose to rely solely on execution migration to provide coherence and consistency.

B. Data placement in distributed memories

The paradigm for accessing data is critical to shared memory parallel systems; Table 2 shows the four possible configurations. Two of these (moving data to computation) have been explored in great depth with many years of research on cache coherence protocols. Recently several data-oriented approaches have been proposed to address the non-uniform access effects in traditional and hybrid cache coherent schemes. An OS-assisted software approach is proposed in [21] to control the data placement on distributed caches by mapping virtual addresses to different cores at page granularity. When adding affinity bits to TLB, pages can be remapped at runtime [5, 21]. The CoG [22] page coloring scheme moves pages to the "center of gravity" to improve data placement. The O^2 scheduler [23], an OS-level scheme for memory allocation and thread scheduling, improves memory performance in distributed-memory multicores by keeping threads and the data they use on the same core.

Hardware page migration support was exploited in

Move Computation to Data (C2D)	Move Data to Computation (D2C)	Resulting Architecture
No	No	Uni-processor model
No	Yes	Traditional cache coherence protocol
Yes	No	EM^2
Yes	Yes	Hybrid schemes that rely on cache coherence

Table 2: Different paradigms of distributing data and computation, and the resulting architectures.

PageNUCA and Micro-Pages cache design to improve data placement [24, 25]. All these data placement techniques are proposed for traditional cache coherent or hybrid schemes. EM^2 can only benefit from improved hardware or OS-assisted data placement schemes. Victim Replication [26] creates local replicas of data to reduce cache access latency, thereby, adding extra overhead to improve drawbacks of traditional cache coherence protocol.

Execution migration not only enables EM^2, but it has been shown to be an effective mechanism for other optimizations in multicore processor. [27] migrates the execution of critical sections to a powerful core for performance improvement. Core Salvaging [28] exploits inter-core redundancy to provide fault tolerance via execution migration. Thread motion [29] exchanges running threads to provide fine-grain power management.

III. EM^2: THE EXECUTION MIGRATION MACHINE

The essence of traditional cache coherence in multicores is bringing *data* to the locus of the *computation* that is to be performed on it: when a memory instruction refers to an address that is not locally cached, the instruction stalls while the cache coherence protocol brings the data to the local cache and ensures that the address can be safely shared (for loads) or exclusively owned (for stores). Execution migration turns this notion on its head, bringing the *computation* to the locus of the *data*: when a memory instruction requests an address not cached by the current core, the execution context (current program counter, register values, etc.) moves to the core where the data is cached.

In this scheme, the physical address space in the system is divided among the cores, for example by striping (see Figure 1), and each core is responsible for *caching* its region of the address space; thus, each address in the system is assigned to a unique core where it may be cached. (Note that this arrangement is independent of how the off-chip memory is accessed,

and applies equally well to a system with one central memory controller and to a hypothetical system where each core has its own DRAM). When the processor executes a memory access for address *A*, it must

1. compute the "home" core for *A* (e.g., by masking the appropriate bits);
2. if the current core is the home,
   a. forward the request for A to the cache hierarchy (possibly resulting in a DRAM access);
3. if the home is elsewhere,
   a. interrupt the execution of the current core (as for a precise exception),
   b. migrate the architectural state to the core that is home for *A*,
   c. resume execution on the new core, forwarding the request for *A* to its cache hierarchy (and potentially resulting in a DRAM access).

Because each address can be accessed in at most one location, many operations that are complex in a traditional cache-coherent system become very simple: sequential consistency and memory coherence, for example, are trivially ensured, and locking reduces to preventing other threads from migrating to a specific core.

This basic sketch intentionally leaves a broad range of design choices. For example, migration (step 3.b above) could preempt the execution on the target core or be subject to scheduling; similarly, the context currently executing on the target core could be either kept on the same core or transferred elsewhere. While we defer the question of the best precise migration algorithm to future research, we focused our investigation in this paper on two simple models: one where the two contexts are swapped, and another where the migrated thread simply moves to the destination core and shares the computational resources with the threads already present there (see Section V.D). Similarly, the questions of dividing the address space among the cores (step 1 above) and finding the best assignment of virtual to physical addresses also potentially offer interesting tradeoffs; in this paper, we evaluated two simple striping schemes based on cache-line size and the operating system page size (see Section V.B).

IV. DISCUSSION

Although at first blush migrating the execution context on every memory access to a non-local region of memory might seem expensive, a careful analysis of a traditional directory-based cache coherence protocol, supported by experimental data, reveals that migration can in fact outperform cache coherence.

A. Costs of directory-based cache coherence

Last-level cache misses in a directory-based cache-coherence scheme incur three significant costs: the latency of potentially retrieving the data from off-chip memory, the latencies associated with the directory protocol itself, and decreased cache effectiveness due to data sharing and directory size limits. For example, in a last-level cache miss under a simple MSI directory protocol,

1. the last-level cache must contact the relevant directory;
2. if *A* is not cached in the directory, the directory must (a) potentially evict another directory entry, contacting all sharers of that entry and waiting for their invalidate responses, and (b) retrieve the data for *A* from off-chip memory;
3. if *A* is already in the directory and the request is for exclusive access, or if *A* is exclusively held by another core, the directory must contact all sharers and wait for their invalidate responses;
4. finally, the directory must respond to the requesting cache with the cache line data for *A*.

The communication cost and the latency of the off-chip memory access, while significant, are dwarfed by the potential deleterious effect on private caches. When an already full directory services a request for a new address (step 2 above), it must replace an existing entry and *invalidate its address in all processor caches* even though the caches themselves did not need to evict the line. Perversely, growing the per-core caches (or adding more processors) without significantly increasing the

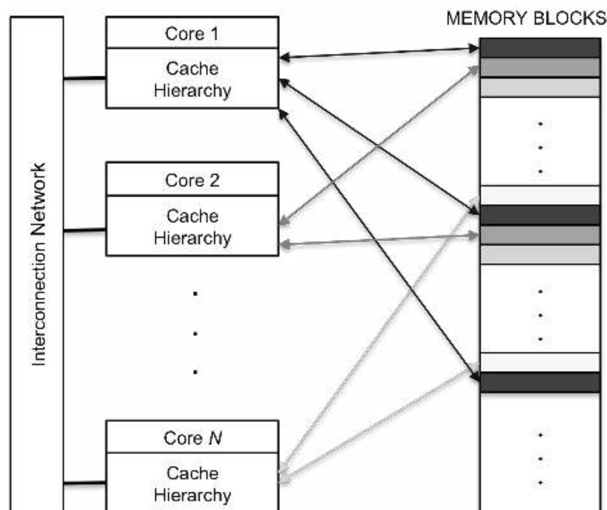

Figure 1: Address-based cache distribution in EM². Each cache (left) is responsible for caching a specific, unique region of main memory (right). In our experiments, main memory is divided into 64-byte or 4KB blocks assigned to consecutive caches; the assignment wraps around and block *N*+1 is again assigned to the first core.

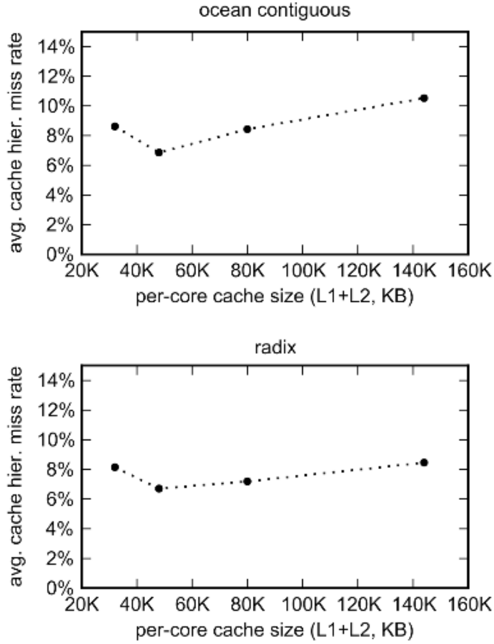

Figure 2: Impact of directory size on cache performance on two SPLASH-2 benchmarks. When caches grow too large in relation to the directory, frequent evictions from the directory lead to cache thrashing. The results show total cache hierarchy misses per memory access in a 64-core cache-coherent model with 5 memory controllers and a 64KB directory for each controller (see Section V.B for configuration details).

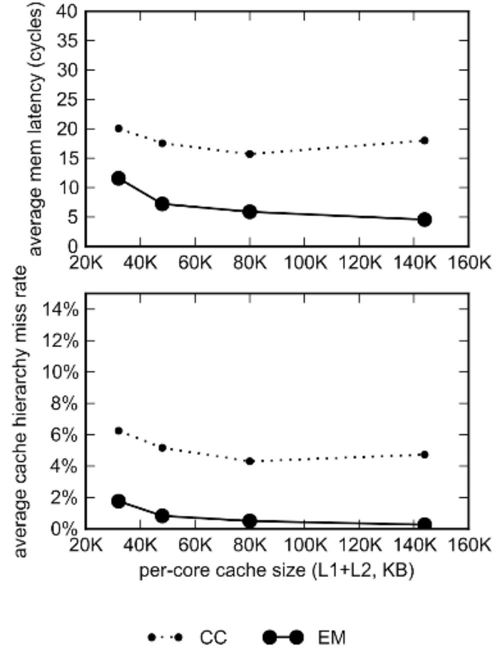

Figure 3: Reduction in memory latency in execution migration (EM) vs. cache coherence (CC) is due to a much lower cache hierarchy miss rate. (The figure shows an average over all benchmarks).

directory size only compounds the problem and increases the cache miss rate, as the larger caches hold more unique addresses and cause more directory evictions. Indeed, 6 of the 16 applications we tested suffered *worse* performance on a realistic cache-coherent 64-core system when the per-core cache size increased. Figure 2 shows two examples of this effect: on both benchmarks, the number of cache hierarchy misses per memory access in the system *increased* when per-core caches grew beyond 48KB.

The magnitude of this effect is application-dependent and the selection of an appropriate directory configuration is not straightforward; at worst, each directory may have to grow as much as all processor caches combined, clearly an unrealistic scenario. If directory limits can impede performance at 64 cores, what will we do when we get to 1,000 cores?

In addition, directory-based cache-coherent multicores suffer from other secondary effects. Most directly, directory sizes needed to retain good performance—especially as core counts and cache capacities grow—use significant area and power, which could instead be allocated to more cores or larger caches. The complex cache and directory controller logic requires area and power as well as significant verification effort. At an architectural level, an implementation of directory-based cache coherence forms an intricate system with many complex interactions, making it difficult to reason about and evaluate design tradeoffs.

In the end, all of these costs stem from one central feature of cache coherence: each cache line may be shared among many cores. This presents a significant opportunity, as eliminating sharing can result in improved performance, complexity, silicon area, and power.

B. *Performance of execution migration*

On most workloads, execution migration significantly improves memory performance: in our benchmarks simulating a realistic 64-core architecture with 5 memory controllers and various per-core cache sizes on a high-performance on-chip interconnect (see Section V.B), the number of off-chip memory accesses in EM^2 decreased by 75%–89% relative to the cache coherent architecture. As shown in Figure 3, the improved memory performance is directly attributable to the significant reduction in last-level cache misses. In turn, this is caused by (a) a significant increase in effective cache capacity when compared to the cache-coherent architecture because each address is cached in at most one location, and (b) the consequent longer lifetime of cache lines in the absence of cache evictions caused by such external requests as exclusive-access requests from other cores or directory evictions.

Critically, as the number of cores on a die grows, the performance advantage of execution migration

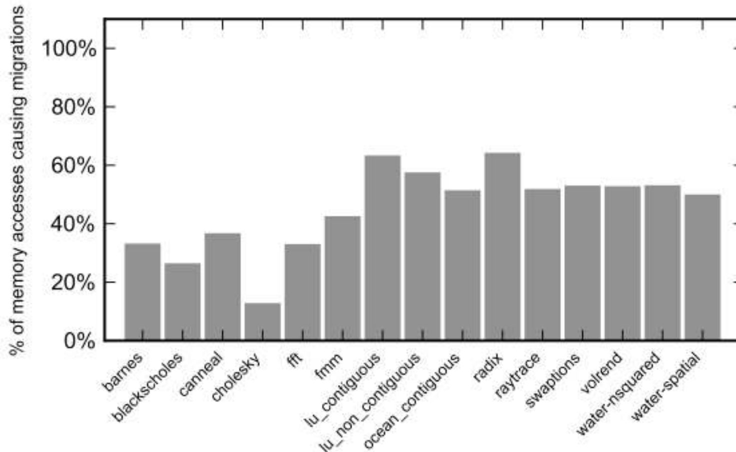

Figure 4: Minimum required execution migration rate for various benchmarks. Specific implementations (e.g., *swap*) may have a higher migration rate.

architecture increases. While the performance of a cache coherence scheme is limited on the one hand by the number of sharers per cache line (and the consequent invalidates caused by exclusive-access requests) and on the other hand by directory sizes (and the consequent invalidates caused by directory evictions), cache miss performance in execution migration depends directly on the effective point-to-point bandwidth and latency provided by the on-chip interconnect, and is much easier to reason about.

C. Costs of execution migration

Since in most workloads memory instructions occur every few cycles and migrations can be frequent: for example, for one of the migration policies we evaluated (*one-way*, see Section V.D), an average of 45% of memory hierarchy accesses in the benchmarks triggered migrations (Figure 4). This, however, is under an OS model that assumes a cache-coherent architecture, and does not allocate memory pages appropriately for an EM^2 architecture. Thus, for example, the stack area and the working set are likely to be allocated in different cores, causing frequent migrations between the two, especially with the heavy stack utilization of an x86 architecture. Efficient page allocation under EM^2 is, however, beyond the scope of this paper a subject of further research.

The main memory access cost incurred by execution migration architecture is that of transferring an execution context to the home cache for the given address. Per-migration bandwidth requirements, although larger than the cache line required by cache-coherent designs, are not prohibitive by on-chip standards: in a 32-bit x86 processor, the relevant architectural state amounts, including a TLB and an instruction cache line, to about 2 Kbits [29]. Although on-chip networks today are not generally designed to carry that much data, on-chip communication scales well; indeed, a migration network is easily scaled by simple replication because all transfers have the same size. Furthermore, execution migration is uniquely poised to take advantage of the high bandwidth, low latency, and low power potential of quickly maturing on-chip optical interconnect technologies [10, 11].

Another potential cost of execution migration is the loss of some instruction locality: when an execution context is moved, the instruction cache in the destination core might not contain the instructions for the transferred thread. In our model, we mitigate this effect by including one 64-byte instruction cache line in the 2Kbit execution context that is migrated between cores (discussed in Section V.B below). While further discussion of instruction caching in execution migration falls outside of the scope of the present paper, we note that (a) many numerically intensive applications (including most SPLASH-2 and PARSEC benchmarks) run the same instructions in each thread, and the instructions cached are likely to be similar, and (b) instruction caches store read-only data and therefore do not require cache coherence logic, and instruction data can easily be replicated by, for example, transmitting the current cache line along with the execution context as we do in our model—over time, instruction caches on each core will store the instructions that operate on the data cached in the same core.

V. METHODS

A. Modeling methodology

We use Pin [12] and Graphite [13] to model the proposed EM^2 architecture. Pin enables runtime binary instrumentation of parallel programs, including the SPLASH-2 [14] and PARSEC [15] benchmark sets we use for evaluation, while the Graphite program analysis pintool provides models for a tile-based core, memory subsystem and network. Graphite provides the

infrastructure to intercept and modify the memory references and present a uniform, coherent view of the application address space to all threads; this allows us to maintain functional correctness in our EM^2 architecture models.

In this paper we do not model non-memory instructions or the memory effects of the instruction cache; since we do not model instruction delay, we also do not model the timing effects of execution other than memory latency. This choice allows us to focus on the data-centric component impact of our architecture on a generic multicore processor.

For the interconnect, we chose to model a fixed-latency high-bandwidth network model where all messages experience the same latency, which allows us to reason cleanly about the role of the interconnect in the memory system performance; consequently, we did not model congestion in the interconnect network. Indeed, this is not an unreasonable assumption. On the one hand, maturing optical interconnect technologies enable high-bandwidth, low-latency communication, and have reached miniaturization levels required for CMOS integration [9]. On the other hand, the technology for high-bandwidth electrical interconnect is already available, and our requirements are not far beyond the capabilities of existing NoC interconnects. For example, the mesh network of the 1GHz TILE64™ multicore processor provides 1.28Tbps of bandwidth to each core [7]; this translates to a bandwidth of 0.32Kbit/cycle one-way in each of the four ports, and means that a 2Kbit execution context can leave (and another arrive at) the core every 6–7 cycles. With an average of 45% of memory accesses causing migrations (cf. Section IV.C), a rate of one memory access every 3–4 processor cycles can be maintained. While this back-of-the-envelope calculation is necessarily approximate (and does not consider, for example, network congestion), it clearly shows that our bandwidth requirements are technologically feasible.

B. System configurations

We ran our experiments using a set of SPLASH-2 and PARSEC applications: *FFT, Radix, Water, Ocean, LU, FMM, Barnes, Volrend, Raytrace, Cholesky, Blackscholes, Swaptions* and *Canneal*; the remaining benchmarks from the two suites cannot run because of the Graphite system limitations in handling certain system calls. Each application was run to completion and used the recommended input set.

For each benchmark, we simulated a 64-core processor with six different memory subsystem configurations and four cache configurations (Table 3). While we concentrated on comparing EM^2 with a realistic cache-coherent MSI design with five memory controllers and 64KB directories for each controller ("CC realistic" in the figures), we reasoned that the complex design-specific interactions between directories and core caches might obscure the true potential of the cache-coherent paradigm, and repeated all experiments with an idealized version with a memory controller on each of the 64 cores and 512KB directories ("CC ideal" in the figures); the total memory bandwidth in the system remained she same (64 GB/s). The simulated application memory space is striped across the memory controller based on either cache line granularity (64 bytes) or OS-page granularity (4KB).

Finally, we assume a fixed-latency network with 5 cycles for communication between any two cores. EM^2 requires more network bandwidth per message than cache coherence, since the execution context (such architectural state as registers, TLB, and an instruction cache line, about 2Kbits in an x86 [29]) is larger than a cache line (perhaps 64 bytes). Since we postulate a high-bandwidth network, we assumed enough bandwidth that the larger context messages will not incur extra latency; to characterize the effect of a less powerful network, however, we repeated our experiments in a model where latencies correspond to the message sizes and EM^2 has latency 4× larger than the cache-coherent architecture.

C. Measurements

We collected the experimental results using a homogeneous cluster of machines. Each machine within the cluster has an Intel® Core™ i7-960 Quad-Core (HT enabled) running at 3.2GHz with 6GB of PC3-10600 DDR3 DRAM. These machines run Debian Linux with kernel version 2.2.26 and all applications were compiled

MemoryConfigurations	CC-realistic	CC-ideal	EM^2
#Memory Controller Ports (MC)	5	64	5
Bandwidth per MC (GB/sec)	13	1	13
Directory Entries	8192	65536	—
Directory Associativity	8	32	—
Total Directory Size	320 KB	32 MB	—
Memory Striping Granularity	• on cache line size (64 Bytes) • on page size (4096 Bytes)		
Network Topology	64-core point-to-point fixed-latency		

Cache Configurations (Private L1 and L2)	Cfg 1	Cfg 2	Cfg 3	Cfg 4
L1$	16 KB, 2-way, 1 cycle access			
L2$ Size	16 KB	32KB	64KB	128KB
L2$ Associativity	4	4	4	4
L2$ Access latency	2	3	4	6
Total Cache Size Per Core	32 KB	48 KB	80 KB	144 KB

Table 3: Memory system configurations used in experiments.

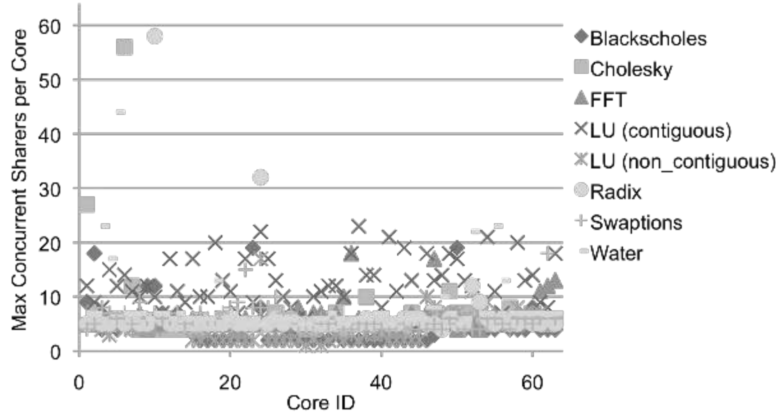

Figure 5: The maximum number of threads concurrently executing on any core in the one-way migration scheme. Most cores never have more than 8 threads executing at the same time.

with *gcc* version 4.3.2.

For each simulation run, we tracked the cache hierarchy miss rates, perceived memory latencies, and, for the EM^2 simulations, migration rates; we averaged per-core numbers weighted by the total memory access count for each core. In figures where data is aggregated over all benchmarks, we averaged per-benchmark data with each benchmark given equal weight to reflect a varied computation load.

D. *Migration algorithm*

We use two migration algorithms for our experiments. In the ***swap*** scheme, when the computation context migrates from, say, core *A* to core *B*, the context in *B* is moved to *A* concurrently. On the one hand, this ensures that multiple threads are not mapped to the same core, and requires no extra hardware resources to store multiple contexts. On the other hand, the context that originated in *B* may well have to migrate back to *A* at its next memory access, causing a thrashing effect. Swap also puts significantly more stress on the network: not only are the migrations symmetrical, the thrashing effect may well increase the frequency of migrations.

The ***one-way*** scheme assumes that multiple contexts can be mapped to any single core and, therefore, would only perform the migration from *A* to *B*. This reduces the strain on the network and the total number of migrations, but requires hardware resources at core *A* to store multiple contexts. While it may appear that *A* might now have to alternate among contexts and become a computational bottleneck, observe that threads executing on *A* are also accessing the memory cached by *A*, and would be subject to additional migrations in the swap scheme.

In the one-way scheme, the number of threads mapped to a single core becomes an important consideration, as it affects both the hardware resources required to store the contexts and the computational power required at each core. In our experiments cores mostly did not exceed 8 concurrent threads at any given time (see Figure 5). While a more detailed evaluation of this design space is beyond the scope of this paper, we observe that including multiple computational units on each core, as in simultaneous multithreading [30], could potentially allow all threads to run in parallel; for example, one might imagine a design where one-way migrations are the default and swaps are used whenever a maximum number of threads mapped to a core exceeds its computational resources.

VI. PERFORMANCE EVALUATION

A. *Memory performance*

This section discusses per benchmark results. Comparisons are drawn based on three data points: two cache coherence implementations and an EM^2 implementation (cf. Section V); unless otherwise noted, the results reflect 4KB main memory striping and a high-performance interconnect.

Figure 6 illustrates that in EM^2 the cache hierarchy miss rate generally directly determines memory performance. For water-nsquared, which has good spatial locality, this is also the case in the ideal cache-coherent model with enormous directories, but under the realistic CC model memory performance suffers from the directory size bottleneck; since EM^2 can cache more addresses in total and is not encumbered by a directory, it performs better. Volrend suffers in both CC regimes because extensive sharing reduces cache utilization; since the per-core caches in EM^2 never share, the total number of addresses cached at any given time is much larger and leads to better performance.

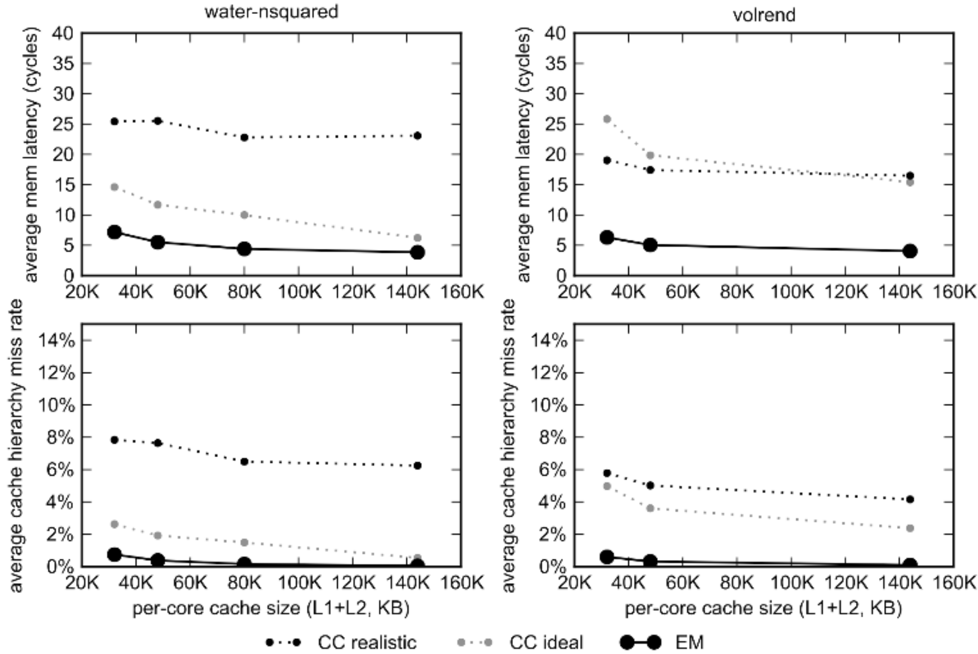

Figure 6: Execution migration significantly reduces the cache hierarchy miss rate, and consequently outperforms either cache coherence model. Note that the cache miss rates for both EM schemes are the same, and the difference in latency is only due to migration frequency.

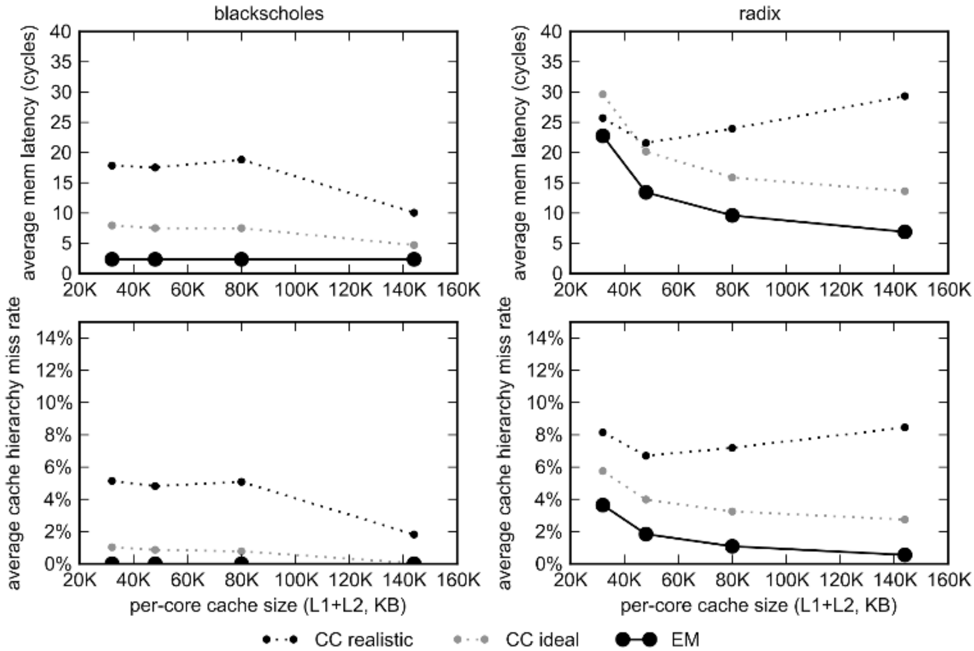

Figure 7: Blackscholes (left) relies heavily on local caches, and memory performance is low on the five-memory-controller CC model until a large cache can hold the working set. Radix (right) under realistic CC performs *worse* with bigger caches because directory cache evictions cause thrashing. (Cache miss rates are the same for both EM schemes).

The blackscholes benchmark in Figure 7 is highly parallel with a relatively small primary working set (64K), and performs well with ideal CC and EM^2. The radix sort kernel showcases how directory sizes can significantly limit performance: the realistic CC model performs *worse* as the per-core caches grow (Figure 7). Since the main working set does not fit in the cache, increasing local cache sizes allows threads to cache more unique addresses; this, in turn, leads to frequent capacity-based evictions from the directory (and,

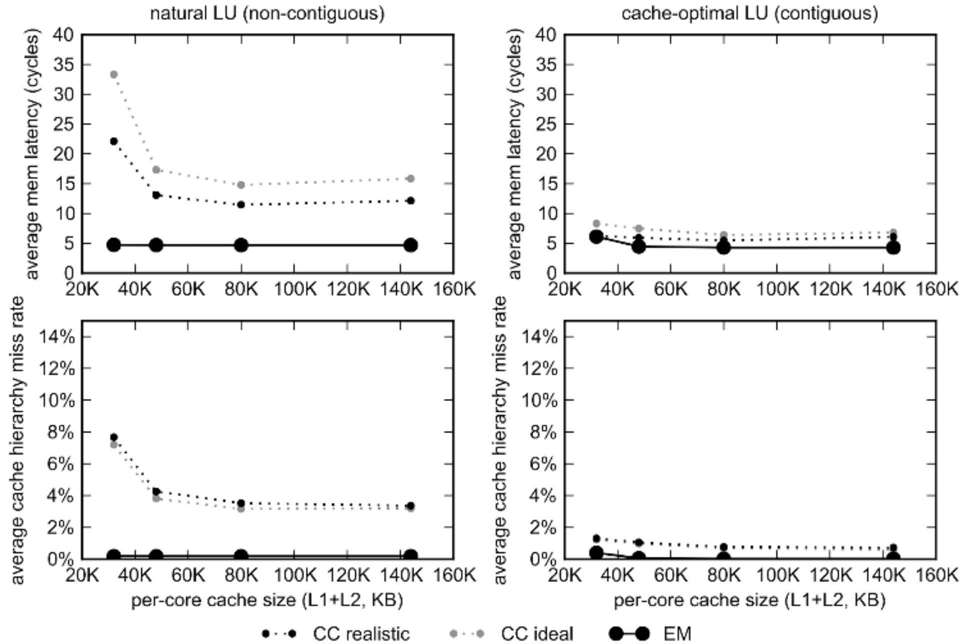

Figure 8: EM^2 reduces the need to optimize code for parallel cache performance: while a cache-optimal implementation of LU decomposition performs well in both CC and EM^2, only the latter also allows a more straightforward implementation to perform equally well. (Cache miss rates are the same for both EM schemes).

consequently, the local caches). When directory space is very large (ideal CC), performance improves steadily with cache size, and is even better in EM^2 as cache lines are never shared and a large portion of the working set can be cached.

The SPLASH-2 suite includes two implementations of matrix LU decomposition: one written in a more straightforward style, and one where blocks accessed by the same thread are allocated contiguously to optimize parallel cache performance. Again, the underlying problem is sharing, and, under a cache-coherent architecture, the performance benefits of the restructured (contiguous) implementation are impressive: cache miss rates drop from 4%–8% to nearly zero (Figure 8). Since EM^2 eliminates the sharing of data among different cores, cache miss rates are low and overall memory performance is very good.

It's worth stressing that EM^2 allowed good performance regardless of how much effort was expended in optimizing the benchmark itself for cache performance. While program optimization for the memory hierarchy will always be important in a small number of critical applications, EM^2 often offers good performance without the expense of optimization, and, when optimization is necessary, provides an architecture where reasoning about cache performance is much easier.

B. Impact of data striping

To determine how the pattern in which main memory is distributed across cores affects the performance of EM^2, we repeated all benchmarks with stripe sizes of 4,096 bytes (equivalent to a common OS page size), 64 bytes (a common cache line size), and 16,384 bytes. The results for 16,384-byte striping closely matched those for 4096-byte striping, and we omit them. The results for 64-byte and 4096-byte striping are summarized in Figure 9: the 4,096 stripe size allows one-way EM^2 to take advantage of spatial locality in memory references and keep threads mapped to cores for longer; in the swap version, threads are evicted by incoming migrations and the migration rate remains higher. On the other hand, if caches are small (on the order of 32KB total), any non-trivial shared data structures that fit in one 4096-byte page are cached in the same cache with 4096-byte striping and are subject to more evictions; with 64-byte striping, they are distributed among many caches and evictions are less frequent.

C. Impact of network latency

While throughout the paper we assume a network with bandwidth sufficient to deliver the 2Kbit context migrations of EM^2 as quickly as the 64-byte messages of a cache coherence protocol, we also modeled a network where bandwidth is low and context migrations take 4× more cycles than cache coherence messages, as much as the difference in message sizes. Figure 10 shows that even under low-bandwidth conditions one-way EM^2 outperforms the directory-based cache-coherence version.

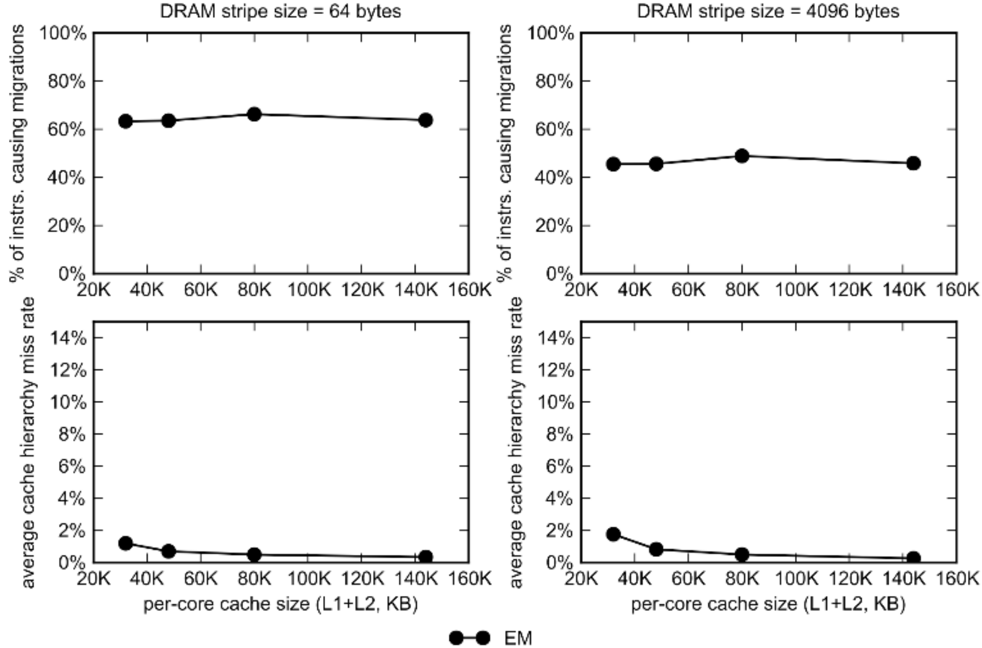

Figure 9: Spatial locality allows the one-way EM^2 model to keep threads mapped to cores they need to access for longer periods when the main memory is striped in larger blocks. (The figure shows an average over all benchmarks; cache miss rates are the same for both schemes).

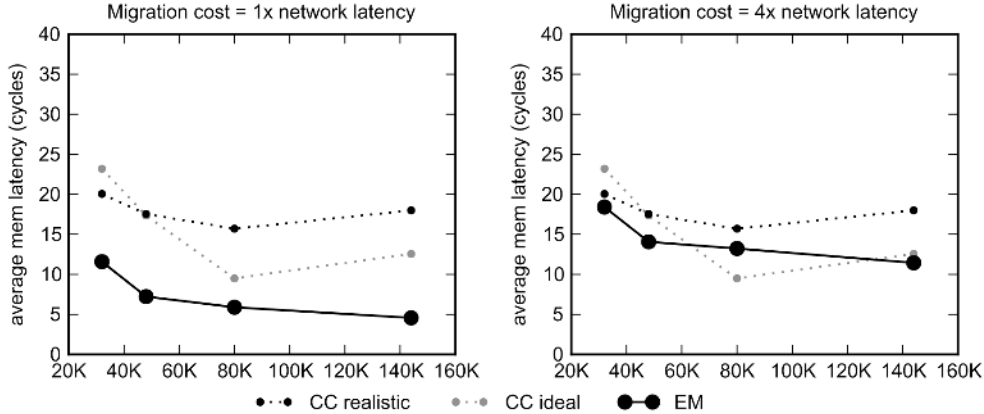

Figure 10: Impact of network latency on EM^2. (The figure shows an average over all benchmarks).

D. Silicon area advantages of EM^2

In general, even compared to "ideal" cache coherence, EM^2 requires significantly smaller cache resources to achieve the same memory latencies: for example, Figure 11 shows that even an "ideal" cache coherence design requires 144KB of caches per core to match the performance of EM^2 with only 80KB of per-core cache, which translates to significant silicon area and power savings.

VII. CONCLUSIONS

In this paper, we introduced EM^2, an instruction-level execution migration-based memory scheme for large-scale multicore processors. EM^2 provides a coherent, sequentially consistent view of a uniform address space without the need for complex cache coherence protocols and the associated silicon area, while reducing the perceived cost of memory accesses: for example, on a set of SPLASH-2 and PARSEC benchmarks, EM^2 reduced cache miss rates by 84% and decreased memory latencies by 58% on average.

While EM^2 offers significant savings in silicon area compared to a directory-based cache-coherent architecture and an EM^2 design can therefore offer more computation resources or larger caches in the same area, we conservatively assume equivalent cache sizes and number of cores for the cache-coherent architectures we compare against. Further research will quantify the savings and determine how to best allocate them.

Throughout this paper we have assumed a high-bandwidth, low-latency on-chip interconnect. Although currently there is little demand to optimize network-on-chip designs for such high bandwidths, we argued that they are not beyond the capabilities of today's electrical interconnects. At the same time, the advent of practical on-chip optical interconnect technology promises to make high-bandwidth, low-latency, low-power on-chip networks common. Either way, the high-bandwidth interconnect we require offers a fertile field for future research.

The performance of EM^2 is bounded by the number of migrations per memory access. While we show that even with a relatively high migration rate EM^2 can outperform a directory-based cache-coherent design, reducing migrations would improve overall performance and lower the interconnect network performance demands; this motivates further research into better migration algorithms, appropriate main memory striping schemes, and OS support for keeping all data used by a thread on the same core. Indeed, the unique properties of the EM^2 architecture open up abundant opportunities for operating system techniques and optimizations.

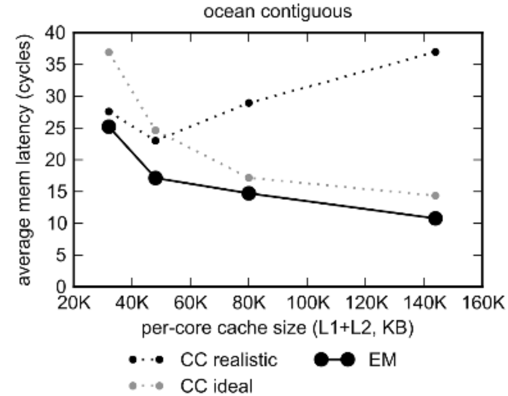

Figure 11: EM^2 requires smaller caches to guarantee the same memory latency on many benchmarks.

REFERENCES

[1] S. Bell et al., "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect," in *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, 2008, p. 88.

[2] S. R. Vangal et al., "An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS," *Solid-State Circuits, IEEE Journal of,* vol. 43, p. 29, 2008.

[3] T. R. Halfhill, "Looking Beyond Graphics," In-Stat Whitepaper2009.

[4] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual Design Automation Conference*, San Diego, California, 2007, p. 746.

[5] N. Hardavellas et al., "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *Proceedings of the 36th annual international symposium on Computer architecture*, Austin, TX, USA, 2009, p. 184.

[6] S. Rusu et al., "A 45nm 8-core enterprise Xeon® processor," in *Solid-State Circuits Conference, 2009. A-SSCC 2009. IEEE Asian*, 2009, p. 9.

[7] D. Wentzlaff et al., "On-Chip Interconnection Architecture of the Tile Processor," *Micro, IEEE,* vol. 27, p. 15, 2007.

[8] D. Park et al., "MIRA: A Multi-layered On-Chip Interconnect Router Architecture," in *Proceedings of the 35th International Symposium on Computer Architecture*, 2008, p. 251.

[9] M. T. Hill et al., "A fast low-power optical memory based on coupled micro-ring lasers," *Nature,* vol. 432, p. 206, 2004.

[10] J. Miller et al., "ATAC: A Manycore Processor with On-Chip Optical Network," MIT CSAIL Technical Report MIT-CSAIL-TR-2009-018, 2009.

[11] N. Kırman et al., "A Power-efficient All-optical On-chip Interconnect Using Wavelength-based Oblivious Routing," in *Proceedings of the 15th international conference on Architectural support for programming languages and operating systems*, Pittsburgh, Pennsylvania, USA, 2006.

[12] M. Bach et al., "Analyzing Parallel Programs with Pin," *Computer,* vol. 43, p. 34, 2010.

[13] J. Miller et al., "Graphite: A Distributed Parallel Simulator for Multicores," in *High Performance Computer Architecture, 2010. HPCA-16. Proceedings. 16th International Symposium on*, 2010, p. 186.

[14] S. C. Woo et al., "The SPLASH-2 programs: characterization and methodological considerations," in *Computer Architecture, 1995. Proceedings. 22nd Annual International Symposium on*, 1995, p. 24.

[15] C. Bienia et al., "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, Toronto, Ontario, Canada, 2008, p. 72.

[16] H. Garcia-Molina et al., "A Massive Memory Machine," *Computers, IEEE Transactions on,* vol. C-33, p. 391, 1984.

[17] P. Michaud, "Exploiting the cache capacity of a single-chip multi-core processor with execution migration," in *High Performance Computer*

Architecture, 2004. HPCA-10. Proceedings. 10th International Symposium on, 2004, p. 186.
[18] K. Chakraborty et al., "Computation spreading: employing hardware migration to specialize CMP cores on-the-fly," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, San Jose, California, USA, 2006, p. 283.
[19] M. Kandemir et al., "A novel migration-based NUCA design for Chip Multiprocessors," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, 2008, p. 1.
[20] W. C. Hsieh et al., "Computation migration: enhancing locality for distributed-memory parallel systems," in *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, San Diego, California, United States, 1993, p. 239.
[21] C. Sangyeun et al., "Managing Distributed, Shared L2 Caches through OS-Level Page Allocation," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, 2006, p. 455.
[22] M. Awasthi et al., "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, 2009, p. 250.
[23] S. Boyd-Wickizer et al., "Reinventing Scheduling for Multicore Systems," in *The 12th Workshop on Hot Topics in Operating Systems (HotOS-XII)* Monte Verità, Switzerland, 2009.
[24] M. Chaudhuri, "PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches," in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, 2009, p. 227.
[25] K. Sudan et al., "Micro-pages: increasing DRAM efficiency with locality-aware data placement," in *Architectural Support for Programming Languages and Operating Systems, 2010. ASPLOS-10. Proceedings. 15th International Conference on*, 2010.
[26] M. Zhang et al., "Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, 2005, p. 336.
[27] M. A. Suleman et al., "Accelerating critical section execution with asymmetric multi-core architectures," in *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, Washington, DC, USA, 2009, p. 253.
[28] M. D. Powell et al., "Architectural core salvaging in a multi-core processor for hard-error tolerance," in *Proceedings of the 36th annual international symposium on Computer architecture*, Austin, TX, USA, 2009, p. 93.
[29] K. K. Rangan et al., "Thread motion: fine-grained power management for multi-core systems," in *Proceedings of the 36th annual international symposium on Computer architecture*, Austin, TX, USA, 2009, p. 302.
[30] S. J. Eggers et al., "Simultaneous multithreading: a platform for next-generation processors," *Micro, IEEE,* vol. 17, p. 12, 1997.