Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2010-030 June 12, 2010

EM2: A Scalable Shared-Memory Multicore Architecture

Omer Khan, Mieszko Lis, and Srini Devadas

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

EM^2: A Scalable Shared-memory Multicore Architecture

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

Abstract

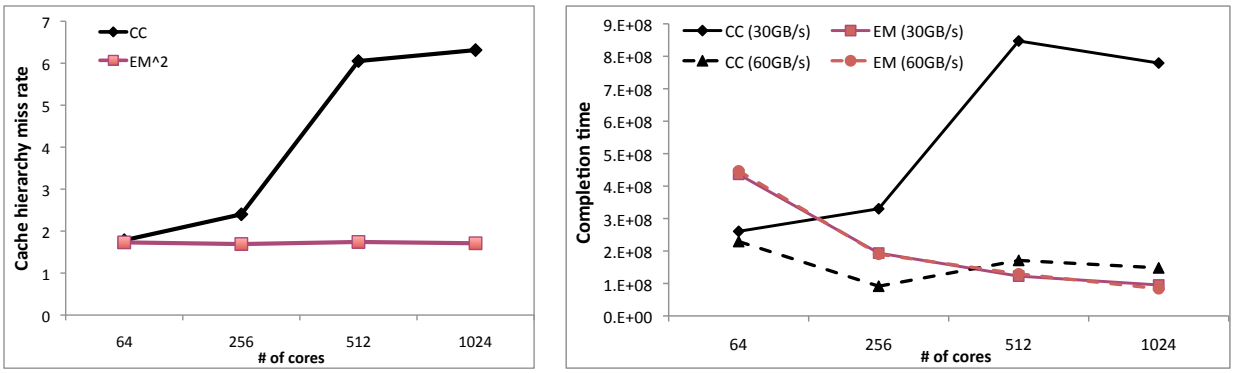
We introduce the Execution Migration Machine (EM^2), a novel, scalable shared-memory architecture for large-scale multicores constrained by off-chip memory bandwidth. EM^2 reduces cache miss rates, and consequently off-chip memory usage, by permitting only one copy of data to be stored anywhere in the system: when a thread wishes to access an address not locally cached on the core it is executing on, it migrates to the appropriate core and continues execution. Using detailed simulations of a range of 256-core configurations on the SPLASH-2 benchmark suite, we show that EM^2 improves application completion times by 18% on the average while remaining competitive with traditional architectures in silicon area.

I. Introduction

In the last few years, the reign of higher and higher frequency unicores has given way to multicore parallelism: four to eight general-purpose cores on a die are now commonplace [1], and designs with many more cores are not far behind [2], [3], [4]. Pundits confidently predict thousands of cores by the end of the decade [5].

Multicore scalability is, however, critically constrained by the *off-chip memory bandwidth wall* [5], [6]: while on-chip transistor counts grow at $\sim 59\%$ for each technology generation and allow more and more cores on a single die, the package pin density—and with it, the off-chip memory bandwidth—increases at a much slower rate of $\sim 5\%$ [7], making performance *worse* as the number of cores on the chip grows (Figure 1). To address this problem, today's 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

[1]equal contributors



(a) cache hierarchy miss rate (%)

(b) application completion time (cycles)

Fig. 1. The effect of off-chip memory bandwidth on multicore performance in the SPLASH-2 radix sort benchmark on 16,000,000 keys (i.e., optimized for 1000-core threads). With 30GB/s of memory bandwidth (solid lines), performance in a cache-coherent shared-memory system (CC) actually *worsens* beyond 64 cores (a): more core means more shared addresses and higher cache miss rates (b), and the memory bandwidth must be doubled for reasonable performance (dashed lines). In contrast, the EM^2 architecture we propose here steadily performs better as the core count grows: because caches do not share data, more addresses can be cached in total, and less memory bandwidth is needed. Multicore simulations using the PIN-based simulator Graphite (see Section III).

not scale beyond a few cores [1]. Many private caches are the only practical option for the predicted 1000-core single chip processors. Exposing the core-to-core communication to software for managing coherence and consistency between caches has limited applicability; therefore, hardware must provide some level of shared memory support to ease programming complexity. Snoop-based cache coherence does not scale beyond a few cores. Directory-based hardware cache coherence requires complex states and protocols for efficiency. Worse, directory-based protocols can contribute to the already costly delays of accessing off-chip memory because data replication limits the efficient use of cache resources.

Since memory bandwidth will not scale, preserving good performance requires significant reductions in off-chip memory access rates. In this paper we introduce the Execution Migration Machine (EM^2) architecture, which addresses this problem by ensuring that no writable data are ever shared among caches: when a thread needs access to an address cached on another core, the system migrates the computation's execution context to the core where the memory is (or allowed to be) cached and continues execution there. Since migrations are handled transparently by the hardware, EM^2 requires only small changes in the operating system and no changes whatsoever in user application code.

In a traditional architecture, memory performance is often determined by the size of *each* core cache and the number of shared addresses; this scales poorly as the core count grows. In an EM^2 system, on

the other hand, memory performance follows the *combined* size of *all* core caches: because each address is only cached in one location, the total number of cached addresses is much higher, which, in turn, allows significantly lower cache miss rates and better overall performance (Figure 1).

The novel contributions of this paper are:

1) We introduce execution migration at the instruction level, an elegant architecture that provides a coherent, sequentially consistent view of memory.
2) We show that migration latency can be mitigated by a multithreading architecture and introduce a novel migration algorithm.
3) We evaluate our architecture on actual multithreaded applications in a current x86-based shared memory system: our functional memory subsystem model is built on PIN/Graphite [8], [9] and runs a set of SPLASH-2 [10] benchmarks with the correct output.
4) We show that using a low-latency mesh interconnection network, on-chip cache miss rate under execution migration improves many fold when compared to traditional directory-based cache coherent system, and, as a result, application completion time improves on an average of 18%.
5) We show that using a realistic on-chip cache hierarchy and an electrical interconnection network, EM^2 elegantly scales to hundreds of cores in a bandwidth-constrained single-chip multicore.

In the remainder of this paper, we describe the details of the EM^2 architecture (Section II), characterize the design space using SPLASH-2 benchmarks inside the multicore x86 simulator Graphite [8] and discuss the implications on hardware and operating system design as well as application optimization (Sections III and V). Finally, we review related work (Section VI) and outline future research directions (Section VII).

II. THE EM^2 ARCHITECTURE

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.

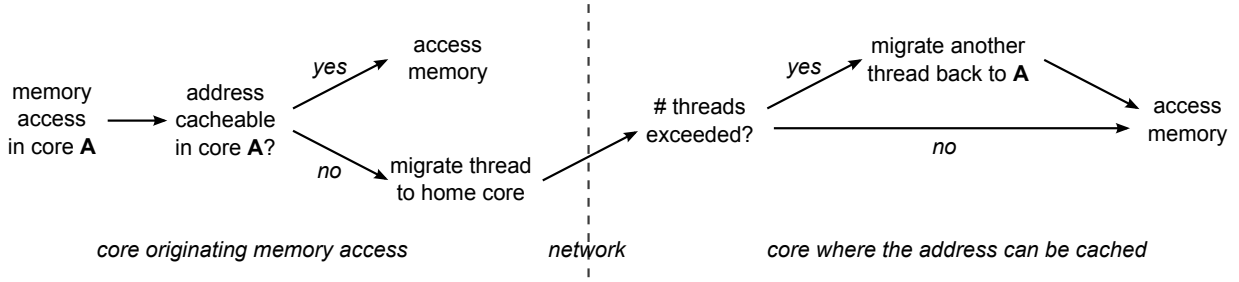


Fig. 2. In an EM^2 architecture, memory accesses to addresses not assigned to the local core cause the execution context to be migrated to the core.

A. *Execution migration*

The physical address space in the system is divided among the cores, for example by striping (see Figure 4), 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 *core hit*),
    a) forward the request for *A* to the cache hierarchy (possibly resulting in a DRAM access);
3) if the home is elsewhere (a *core miss*),
    a) interrupt the execution of the current core (as for a precise exception),
    b) migrate the microarchitectural 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.

The core miss cost incurred by an EM^2 architecture is dominated by transferring an execution context to the home cache for the given address. Per-migration bandwidth requirements, although larger than those required by cache-coherent designs, are not prohibitive by on-chip standards: in a 32-bit x86 processor, the

relevant architectural state amounts, including TLB, to about 1.5 Kbits [11]. Although on-chip electrical networks today are not generally designed to carry that much data, on-chip communication scales well; indeed, a migration network is easily scaled by replication because all transfers have the same size. In this section, therefore, we focus on the migration algorithm itself.

Figure 2 shows the proposed migration framework. On a core miss (at say core A), the hardware initiates an execution migration transparent to the operating system. The execution context traverses the on-chip interconnect and, upon arrival at the home core (say core B), is loaded into the core B and the execution continues. In a single-threaded core microarchitecture, this results in a SWAP scheme, where the thread running on the core B is evicted and migrated to core A. While this ensures that multiple threads are not mapped to the same core and requires no extra hardware resources to store multiple contexts, the context evicted from core B may well have to migrate back to core B at its next memory access. In addition, the SWAP scheme also puts significantly more stress on the network: not only are the migrations symmetrical, the thrashing effect may well increase the frequency of migrations. For this reason, we relax the SWAP algorithm by allowing each core to hold multiple execution contexts (cf. Section II-B), and resorting to evictions only when the number of hardware contexts running at the target core would exceed available resources.

Overall, the cost of a memory access under EM^2 is driven by the cost of off-chip memory accesses due to cache misses and the cost of migrations due to core misses:

$$rate_{core\_hit} \times (rate_{cache\_hit} \times cost_{cache\_hit} + rate_{cache\_miss} \times cost_{DRAM\_access}) + \quad (1)$$
$$rate_{core\_miss} \times (cost_{migration} + rate_{cache\_hit} \times cost_{cache\_hit} + rate_{cache\_miss} \times cost_{DRAM\_access})$$

While $cost_{DRAM\_access}$ is relatively constrained, we can optimize performance by improving the other variables. Thus, in the following sections we show how our architecture improves $rate_{cache\_hit}$ when compared to a cache-coherent baseline, discuss several ways to keep $rate_{core\_hit}$ high, and argue that today's interconnect technologies keep $cost_{migration}$ sufficiently low to ensure good performance.

B. Multithreaded core

EM^2 deploys a multithreading architecture [12], primarily to enable efficient pipelining of execution migrations. A multithreaded core, shown in Figure 3, contains architectural state (program counter, register file and virtual memory translation cache) for multiple threads, and at any given cycle, a core fetches instructions from one of the threads. This architecture is well suited for EM^2 because it is more tolerant of long latency operations and effective for hiding the serialization effects of multiple threads contending

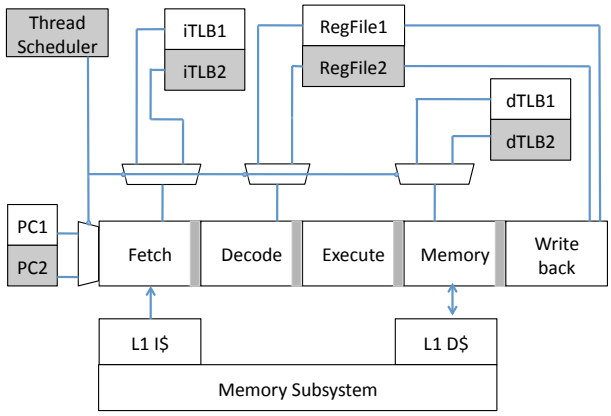


Fig. 3. The architecture of a single core two-threaded core in EM^2. Differences from a single-threaded pipeline are highlighted.

on a core. When instruction-level parallelism becomes a bottleneck, multiple issue slots can be used. Although a wide issue multithreaded core is more efficient for exploiting parallelism, it comes at the cost of increased implementation complexity, hardware and power overheads; while we characterize and discuss the choice of multithreading and issue-slots for the proposed EM^2 architecture in subsequent sections, we chose a 2-way multithreaded core with single issue-slot because it offered best performance and area cost tradeoffs.

C. Instruction cache

Since the migrated execution context does not contain instruction cache entries, the target core might not contain the relevant instruction cache lines and a thread might incur an instruction cache miss immediately upon migration. To evaluate the potential impact of this phenomenon, we compared L1 instruction cache miss rates for EM^2 and the cache-coherent architecture; simulations on a range of benchmarks showed an average instruction cache miss rate of 0.19% under EM^2 as compared to 0.27% for the cache-coherent architecture (data not shown). Instruction cache performance is, in fact, better under our EM^2 implementation because a thread's instructions are split among instruction caches according to the location of the data they access.

D. Data placement

Careful assignment of address ranges to the various core caches can address the two factors that determine the performance of an EM^2 design: (a) off-chip memory access cost, and (b) pauses in execution

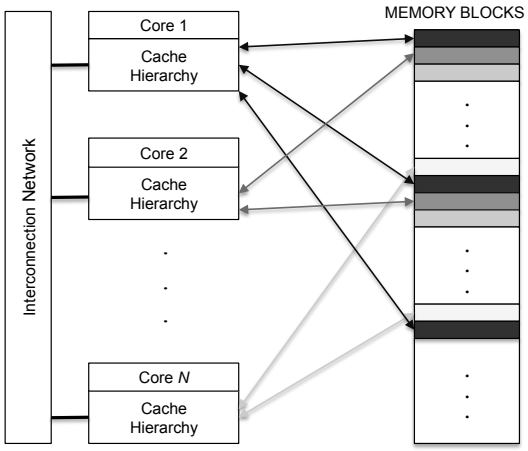


Fig. 4. A simple address-striped cache distribution in EM^2. Each cache (left) is responsible for caching a specific, unique region of main memory (right). In this scheme, main memory is divided into blocks (e.g., the size of an OS page) assigned to consecutive caches; the assignment wraps around and block $N+1$ is again assigned to the first core.

due to migrations. On the one hand, spreading frequently used addresses evenly among the core caches ensures that more addresses are cached in total, reducing cache miss rates and, consequently, off-chip memory access frequency; on the other hand, keeping addresses accessed by the same thread in the same core cache reduces migration rate and network load.

Unlike in a traditional architecture, however, where the OS schedules threads to run on specific cores, threads under EM^2 automatically migrate among cores as needed by their memory accesses without involving the OS. Instead, the operating system controls thread-to-core distribution indirectly via the existing virtual memory mechanism: when a virtual address is first mapped to a physical page (e.g., on program load or on first access), an EM^2-aware operating system can choose where the relevant page should be cached by mapping the virtual page to a physical address range assigned to a specific core.

Figure 4 shows a simple assignment where the address space to be cached is striped, perhaps in page-sized blocks, among the available core caches; to assign a virtual page to a physical address range, the OS masks out a few bits from the virtual address to compute the target core and maps the virtual page to the next available range cached by the desired core. The STRIPED scheme assigns consecutive 4 KB pages to consecutive cores: addresses 0x0000–0x0fff are assigned to core 0, 0x1000–0x1fff to core 2, and so on. The assignment wraps around, so, in a 256-core system, addresses 0xff000–x0fffff would reside in core 255, while 0x100000–0x100fff would again be assigned to core 0.

Since the OS knows which thread caused the page fault, more sophisticated heuristics are possible:

for example, the OS might map the page to the thread's "home" core, taking advantage of data access locality to reduce the migration rate while keeping the threads spread among cores. The ORIGINAL scheme, which also uses 4 K pages, assigns to each thread a "home" core: for example, on a 256-core system, each thread in a 256-thread application might be assigned to a separate core. Using the existing virtual memory subsystem, page-to-core mapping is delayed until some thread actually reads or writes an address on the given page; on the first access, the page is mapped to the accessing thread's home core and remains there for the duration of the simulation. With this heuristic, the ORIGINAL scheme aims to keep each thread on its home core for as much of its running time as possible.

Although a detailed analysis of possible data placement algorithms is beyond the scope of this paper, the present discussion warrants a few observations. First, an optimal algorithm will treat addresses differently based on whether they are private to a thread (in which case they OS might try to keep the pages together), or shared among several threads (in which case the OS might try to distribute the pages to keep one core from being swamped). Indeed, one might even imagine systems where page-to-core assignment is periodically rebalanced by the operating system (e.g., by flushing relevant TLB and cache entries and copying the memory pages, or by adding another layer of address translation at the memory controller). Read-only data present another design choice: in computations where data is never modified after some point during execution (e.g., data read from an input file), such data can be "frozen" and shared among core caches without the need of a coherence protocol; depending on the sharing patterns in the application, sharing read-only data might give the OS even more fine-grained control over the tradeoff between limiting cache misses and reducing migration rates. Finally, as with any memory system, performance depends on the way application accesses its data structures and the way these structures are laid out in memory: for example, an EM^2-aware compiler might attempt to assign shared and private data to different pages and facilitate an effective data placement by the OS.

While we characterize and discuss the choice of the STRIPED versus ORIGINAL schemes for the proposed EM^2 architecture in subsequent sections, we chose the ORIGINAL scheme because it significantly lowered the execution migrations from 66% for STRIPED to 38%.

E. Operating system support

The programmer's view of an EM^2 is no different from any other shared memory design, and so porting an existing OS to EM^2 is no different from porting to any processor architecture. The relevant differences include:

- Multiprocessing operating systems schedule threads *per processor*, and, indeed, threads can often

Parameter	Settings
# Cores	256
Multithreading	2 threads, 1 issue-slot
L1 data cache per core	16 KB
L2 cache size per core	64 KB
Network	Mesh with XY routing
Network hop latency	1 cycle
Migration scheme	original, 4K page size
VM page size	4 K
Total directory size	∼1.5 MB
Coherence protocol	MSI
Memory bandwidth	30 GB/s
Off-chip latency	75 ns

TABLE I
SYSTEM CONFIGURATIONS USED

request assignment to a specific core. In EM^2 this is not directly possible, since threads migrate among cores as required by the memory address being accessed without any OS involvement; instead, the OS can only control at *system* granularity which threads are currently being scheduled to run and which are not.

- While an existing virtual page mapping scheme can work in EM^2 without changes, greater care in mapping virtual to physical pages can offer performance benefits (cf. Section II-D); an OS optimized for EM^2 might map pages carefully to ensure good cache utilization and minimize migration rates.

III. METHODS

We use Pin [9] and Graphite [8] to model the proposed EM^2 architecture. Pin enables runtime binary instrumentation of parallel programs, including the SPLASH-2 [10] benchmark sets we use for evaluation, while Graphite models a tile-based core, memory subsystem, and network, as well as ensures functional correctness.

The settings used for the various system configuration parameters are summarized in Table I and any deviations are noted when results are reported. In experiments comparing EM^2 against cache coherence, the parameters for both were identical, except for (a) the memory directories which are not needed for

Component	Count	Total Area (mm^2)	Details
EM^2 extra context	256	1.63	1.5Kbits per context
EM^2 router	256	15.67	5-Read, 5-Write ports; 256x20 bits
Coherence router	256	3.78	5-Read, 5-Write ports; 64x20 bits
Directory	8	3.55	256 KB cache; 16-way associative
L2 Cache	256	41.39	128 KB
		26.65	64 KB
		13.28	32 KB
		10.21	16 KB

TABLE II
AREA ESTIMATES

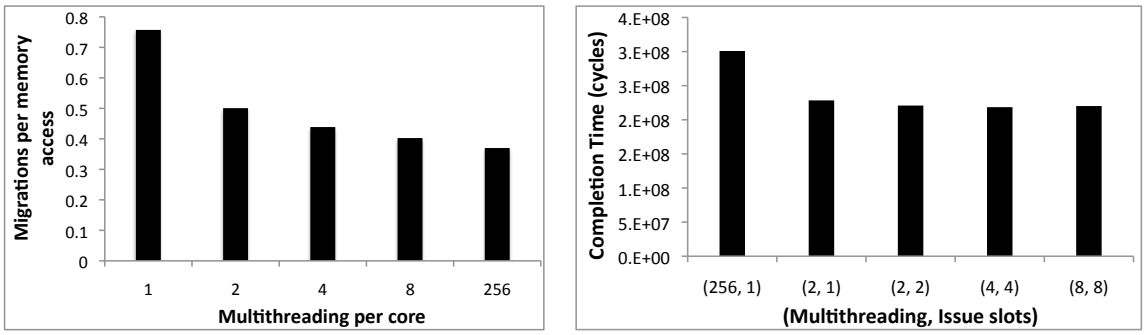
EM^2 and were set to sizes recommended by Graphite on basis of the total cache capacity in the simulated system, (b) the 2-way multithreaded cores which are not needed for cache coherent system, and (c) the interconnection network.

A. On-chip interconnect

Experiments were performed using Graphite's model of a mesh network with XY routing and fixed per-hop latency. For cache-coherent architecture, each packet is partitioned into 64-bit flits. As EM^2 requires higher bandwidth due to larger size of execution context (architecture registers and TLB entries account for 1.5 Kbits in an x86 [11]), we chose a 256-bit flit size. The choice of 256-bit flit size is a function of packet injection rate for EM^2 and the area overhead of additional buffers and crossbar in the router; the characterization and discussion sections describe the tradeoffs in more detail. Since modern network-on-chip routers are pipelined [13], we model a one-cycle per-hop latency [14]; for EM^2, we model a pipeline fill delay of 6 cycles to account for the six-flit context size, while for cache-coherence we optimistically assume a 0-cycle fill delay.

B. Area estimation

We assume 32nm process technology and use CACTI [15] to estimate the area requirements of the proposed on-chip caches and interconnect routers. To estimate the overhead of extra hardware context in the 2-way multithreaded core for EM^2, we used Synopsys Design Compiler [16] to synthesize the extra logic and storage. The area numbers used in this paper are summarized in Table II.



(a) # migrations per memory access

(b) application completion time (cycles)

Fig. 5. Performance of EM^2 with different core microarchitectures. With a single-threaded, single-issue core, each migration to a core must be balanced by a migration out of that core, which significantly affects performance; keeping more than one thread in each core—even when issuing at most one instruction per cycle—lowers the migration rate and significantly improves performance. We selected (2, 1) based on these results.

C. Measurements

Our experiments used a set of SPLASH-2 applications: FFT, LU-CONTIGUOUS, LU-NON-CONTIGUOUS, OCEAN, RADIX, RAYTRACE, and WATER-N^2; 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 the number of cores used, except as otherwise noted. For each simulation run, we tracked the total completion time, the parallel work's completion time, the percentage of memory accesses causing cache hierarchy misses, and the percentage of memory accesses causing migrations.

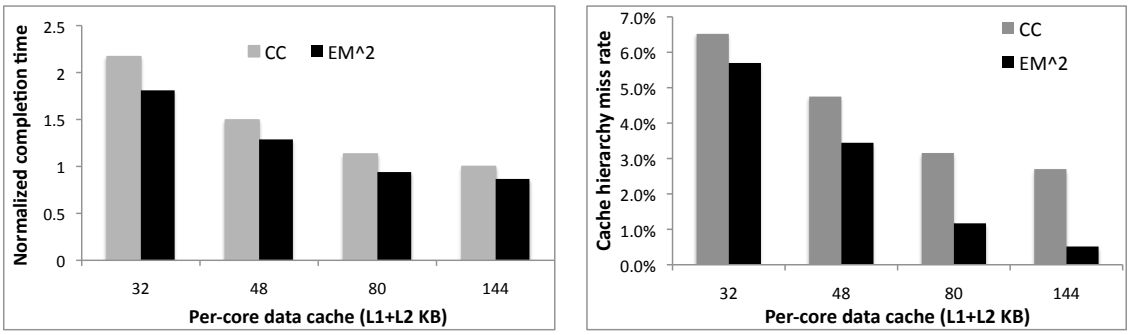
To exclude differences resulting from relative scheduling of Graphite threads, data were collected using a homogeneous cluster of machines. Each machine within the cluster has an Intel® Core™ i7-960 Quad-Core (HT enabled) running at 3.2 GHz with 6 GB of PC3-10600 DDR3 DRAM. These machines run Debian GNU/Linux with kernel 2.2.26 and all applications were compiled with gcc 4.3.2.

IV. CHARACTERIZATION

In order to select a performance-cost tradeoff suitable for EM^2, we explored the design space via running our SPLASH-2 benchmark set under a range of design parameter choices.

A. Sharing & serialization

Because a thread remains inside a core it has migrated to until it reaches a memory access to memory cached in another core, several threads may potentially wish to execute on the same core at the same



(a) application completion time, normalized to 144 K CC

(b) cache hierarchy miss rate (%)

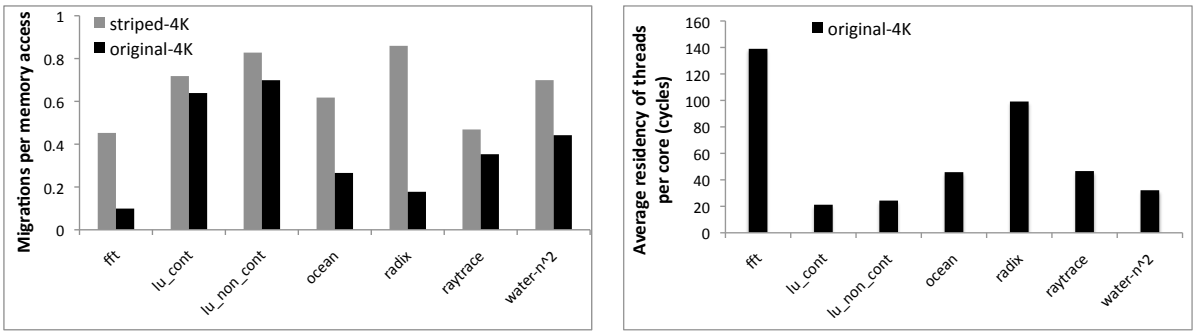
Fig. 6. Effect of cache sizes on performance using parameters from Table I.

time. To address this problem, EM^2 permits multiple threads to run in a single pipeline in a round-robin fashion (see Section II-B); when the permitted number of threads in a single core has been reached, an incoming migration evicts a thread from the core and sends it back to its own origin. On the one hand, each thread residing in a core pipeline requires additional resources (register files, TLBs, etc.); on the other hand, increasing migration frequency by evicting threads adds load to the network and slows down computation.

To investigate the relevant performance tradeoffs we examined system performance under different core multithreading and issue-slot configurations. As Figure 5a illustrates, the migration rate per memory access decreased steadily as the number of threads allowed to reside on a single core increased because evictions became less frequent: in the extremes, allowing only one thread per core induced an evict migration for every data-driven migration, while the 256-thread configuration exhibited no evictions. The wall-clock performance results (Figure 5b) show that improved migration rates are partially counterbalanced by serialization effects: when many threads share one pipeline, each thread may have to wait many cycles while other threads run. Overall, the configuration with 2-way multithreading and single issue-slot per core offered the best tradeoff and was used for the remaining experiments.

B. Cache size

We used the same benchmark set to select a suitable per-core data cache size. While adding cache capacity improves cache utilization and therefore performance for both our cache-coherent baseline and our EM^2 design, cache miss rates are much lower for EM^2 and equivalent performance can be achieved with smaller caches: for example, EM^2 performance with 80 KB data caches matches the performance of the baseline with 144 KB data caches (Figure 6). For the remaining experiments, we chose the 80 KB



(a) Migration rates for two data placement schemes

(b) Number of cycles a thread is pinned to a particular core

Fig. 7. Page-to-core mapping strategy can significantly affect the migration rate. For all benchmarks, the ORIGINAL scheme (which assigns accessed pages to a thread's home core) outperforms the STRIPED scheme (which stripes the address space among the caches on page boundaries). Page sizes in this experiment were 4 KB.

cache configuration as a reasonable tradeoff point. We revisit the cache size tradeoffs in the discussion section where area versus performance considerations are discussed.

C. Data placement

In an EM^2 architecture, migrations are caused by memory references to addresses that are assigned to remote cores, and careful assignment of address ranges to cores can reduce the rate of thread migration and thus improve performance (Section II-D). While choosing to relegate an exhaustive study of data placement algorithms to future work, we sought to evaluate the possible impact on performance by comparing two allocation schemes.

Figure 7a shows that the effect of data placement on the migration rate is not only significant, but also application-dependent. For all applications, the ORIGINAL scheme outperforms striping, sometimes by a wide margin (e.g., FFT and RADIX). Intuitively, the ORIGINAL heuristic performs better because it allocated memory accessed by a given thread to the thread's home core, and therefore keeps the thread executing there most of the time; naturally, the effect is greater for private data than for shared data, and different applications benefit to varying degrees.

D. Network

Data placement directly affects the network capacity required for EM^2: with a lower per-memory-access migration rate, the less frequent migrations put less load on the network. We reasoned that because on *average* each of the two threads possibly executing on the core stays on the core for just under 60

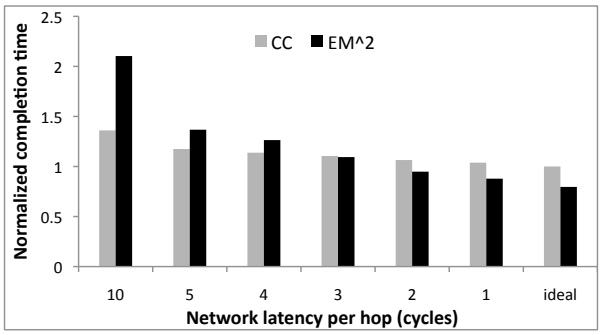


Fig. 8. Application completion time, normalized to an ideal point-to-point network CC. Network design affects EM^2 much more than an equivalent cache-coherent design: while EM^2 outperforms an equivalent cache-coherent design on a network with 1-, 2- and 3-cycle per-hop latency, the trend reverses for a 4-cycle per-hop latency, and EM^2 does significantly worse when each hop takes 10 cycles. The network used here is a 2D mesh with XY routing; classical network-on-chip routers employ pipelines with three to four stages [13] and for a migration network used by EM^2 fewer pipeline stages are possible.

cycles (cf. Figure 7b), there would be, on average, one migration from that core at most every 30 cycles. To balance the interconnect bandwidth (and buffer area) needed to support that rate against network congestion effects, we chose to split the 1.5-KBit execution context packet into six 256-bit flits, a figure which overprovisions network bandwidth 5 $\times$.

In addition to reducing the migration rate via intelligent data placement, migration performance can be improved by decreasing migration latency: that is, improving the interconnect itself. To quantify the network performance effects, we ran our benchmark set on a mesh on-chip network with XY routing under various per-hop latencies, as well as an "ideal" point-to-point network where each transfer takes three cycles (load, transfer and unload); while we have not modeled network congestion explicitly in our model due to dramatic slowdowns in simulation speed, we left headroom for significant congestion by overprovisioning network bandwidth 5 $\times$ and doubling the per-hop latencies to 2 cycles. Since even a cache hit under EM^2 may require a migration, the performance of this architecture is significantly affected by network latency; cache coherent architectures, on the other hand, employ the network only on cache misses, and are less affected (Figure 8). Nevertheless, for reasonable per-hop latencies ranging from 1 cycle (little network congestion) to 3 cycles (significant network congestion), the EM^2 architecture outperforms the cache coherent baseline.

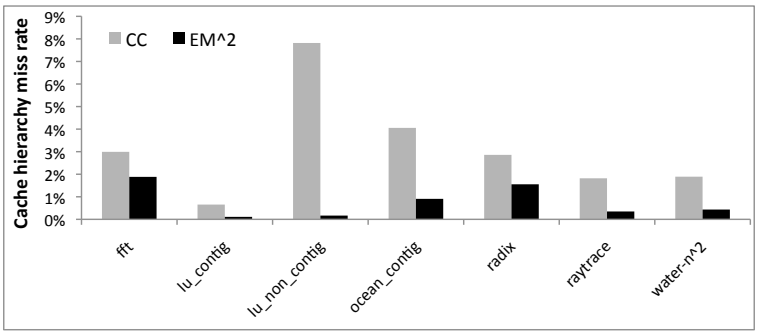


Fig. 9. Cache miss rates under EM^2 and a cache-coherent design on a variety of benchmarks. In most cases, cache miss rate under EM^2 is much lower because storing each cache line in only one location effectively increases the number of cache lines available.

V. Discussion

A. Cache miss rates

The performance benefits of the EM^2 architecture are mainly due to lowered cache miss rates, which, in turn, limits the number of cycles spent waiting for off-chip memory accesses. Because the improvement arises from not storing addresses in many locations, the benefits naturally depend on the memory access pattern of the application (Figure 9): for example, the FFT benchmark does not exhibit much sharing and cache miss rate under EM^2 is 2% lower than the under the cache-coherent baseline design, while for the LU decomposition benchmarks, cache miss rates under EM^2 are negligible.

The two SPLASH-2 implementations of LU decomposition (LU-NON-CONTIGUOUS and LU-CONTIGUOUS) provide a good illustration of the weaknesses inherent in cache coherence. The original version of LU-NON-CONTIGUOUS was written without regard to the details of cache coherence protocols, and, under the baseline architecture, exhibits a significant amount of false read-write sharing [17]: this causes a lot of cache misses, even though the data set would otherwise fit in the cache. In the LU-CONTIGUOUS version, on the other hand, data layout was painstakingly hand-optimized to remove false sharing, and the cache miss rate is lower by $13\times$. Since our EM^2 design does not permit any sharing, cache lines are almost never evicted and the cache miss rates are negligible for both implementations.

Such variations aside, cache miss rates were significantly lower under EM^2 than in the cache-coherent baseline architecture. As a result, the EM^2 design can be expected to scale to many more cores than an equivalent cache-coherent architecture while keeping the pressure on the off-chip memory bandwidth low. Alternately, for a smaller chip where memory bandwidth may not be the overriding concern, EM^2

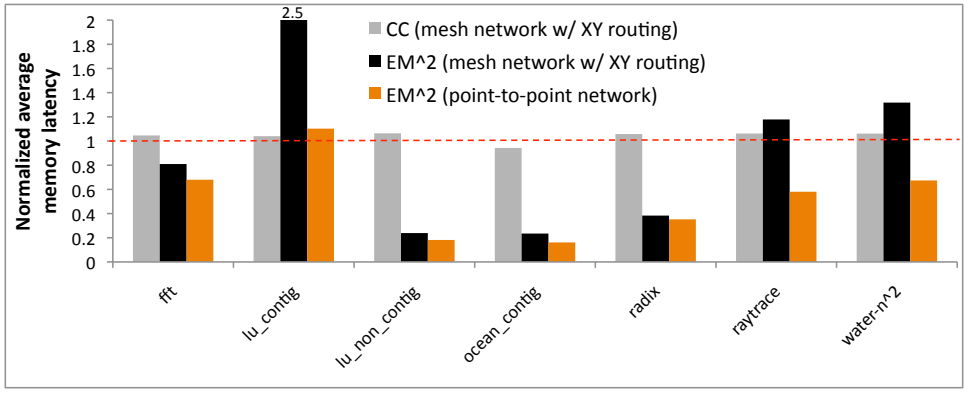


Fig. 10. Average memory latency for different network configurations, normalized to ideal point-to-point network CC.

can ship with smaller caches, thus potentially reducing the necessary package size and cost.

B. Network-dependent performance

Because, depending on the application memory access patterns, migrations in an EM^2 architecture may occur frequently, the interconnect network is a much more important design point for EM^2 than for a cache coherent architecture. In particular, applications that have a lot of read-only sharing but fit in per-core caches do not cause much cache coherence traffic in a traditional design, but may be subject to many migrations under EM^2.

Figure 10 illustrates the average memory access latency in our cache-coherent baseline and the corresponding EM^2 design under different network designs. While EM^2 performs better for most benchmarks even under a 2D mesh network with XY routing, the two versions of LU decomposition again offer an instructive example. Although cache miss rates for both versions under EM^2 are negligibly small (see Section V-A), both must pay the cost of core-to-core execution context migrations; while the abysmal memory performance of LU-NON-CONTIGUOUS under cache coherence can be outperformed by EM^2 with nearly any network, the cache-coherence-optimized LU-CONTIGUOUS version requires a low-latency network to compete under EM^2.

The mesh (XY) network model we used for our experiments can be implemented today: as discussed in Section III-A, with pipelined network switches, a one-cycle per-hop latency is reasonable when congestion is not significant. But the point-to-point model we used is not unreasonable: optical network technology has already achieved levels of miniaturization [18] that would permit the kind of low-latency, high-bandwidth on-chip interconnects we contemplate [19]. In either case, it is clear that EM^2 provides a

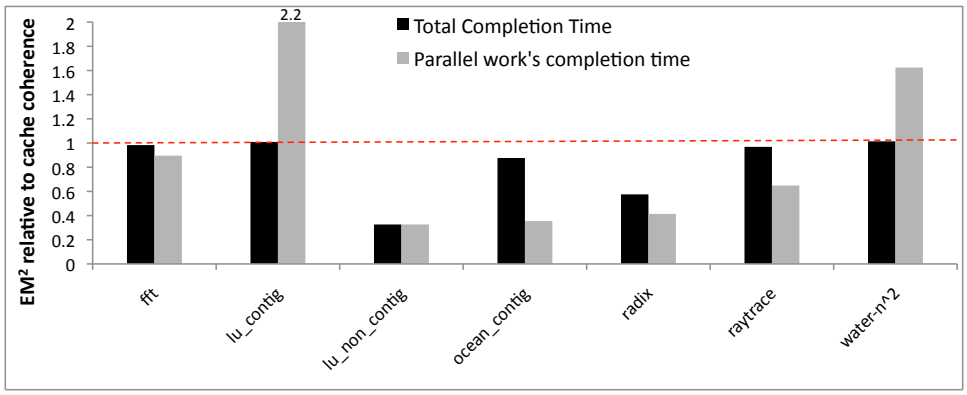


Fig. 11. Overall completion time and parallel work's completion time using parameters from Table I

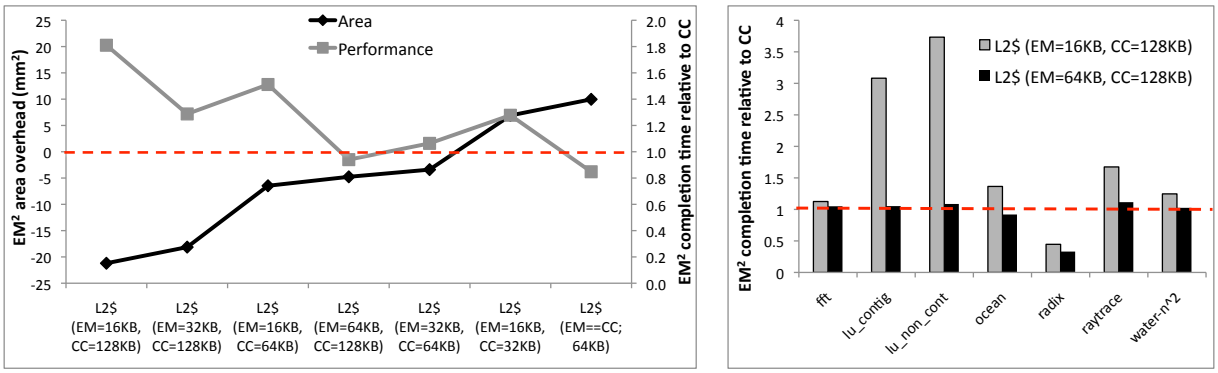
driver for future on-chip network design.

C. Overall performance

Many of the benchmarks we used to evaluate our system follow a three-stage pattern: in the first stage, a data set is allocated and initialized in a single thread; in the second, threads are spawned and perform the parallel computation of interest; finally, in the third stage, the threads are joined and the final results are collated and reported in the original single thread. Because benchmarks by necessity consist of relatively small compute kernels, the single-threaded stage can form a significant part of the total running time.

On the one hand, the system architect might be designing a system that runs one time-critical, compute-intensive application, a common paradigm in scientific computing; on the other hand, the multicore might be intended for a multitasking system with many semi-independent threads where throughput is key, a situation common in server applications where a completed thread would simply be replaced by the next queued request. In the first case, the important performance characteristic is the end-to-end running time including the single-threaded sections; in the second, an architect might focus on the performance of only the parallel section of each benchmark.

Consequently, in Figure 11, we report both metrics. The end-to-end completion time is always better under EM^2; the parallel completion time is better for most benchmarks, the exceptions being LU-CONTIGUOUS and WATER-N^2, which combine a low cache-miss rate under cache coherence with a high migration rate under EM^2.



(a) Area versus performance at various data cache configurations

(b) Per-benchmark completion time results

Fig. 12. Highlights EM^2 competitiveness at several data cache size tradeoffs.

D. Area vs. performance

Like any architecture, EM^2 offers tradeoffs between area and performance: Figure 12a illustrates how different per-core L2 cache size choices affect silicon area and performance as compared to a cache-coherent baseline design. For example, since EM^2 relies more heavily on the interconnect network, the improved performance of EM^2 comes at the cost of a larger estimated silicon area when both architectures are given 64 KB of cache.

When area is of paramount concern, EM^2 offers advantages because it requires smaller cache sizes for the same performance as cache coherence (cf. Section IV-B). Although in the extreme case of 16 KB L2 caches for EM^2 vs. 128 KB caches for the baseline the significant area savings come at a cost of an average performance drop (Figure 12a), a more detailed analysis (Figure 12b) reveals that this drop is mostly due to only two benchmarks, and other benchmarks perform competitively or even significantly better, and suggests that the architectural tradeoff may be reasonable.

Critically, the area and performance of EM^2 are competitive with cache coherence at a number of design points: in fact, an EM^2 design with 64 KB L2 caches outperforms a cache-coherent design with 128 KB L2 caches while taking up less silicon area: all benchmarks perform competitively on EM^2, and our motivating example RADIX significantly outperforms cache coherence even under all configurations (Figure 12b).

Move computation to data (C2D)	Move data to computation (D2C)	Resulting architecture
No	No	Uniprocessor
No	Yes	Traditional cache coherence
Yes	No	EM^2
Yes	Yes	Hybrid schemes that rely on cache coherence

TABLE III

DIFFERENT PARADIGMS OF DISTRIBUTING DATA AND COMPUTATION, AND THE RESULTING ARCHITECTURES

VI. 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 [20]. 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 [21]. Computation spreading [22] 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 [23]. This work attempts to find an optimal data placement for cache lines. A compile-time program transformation based migration scheme is proposed in [24] 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 scheme allows programmer to express when migration is desired. 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 III 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 [25] 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 [6], [25]. The CoG [26] page coloring scheme moves pages to the "center of gravity" to improve data placement. The O^2 scheduler [27], 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 PageNUCA and Micro-Pages cache design to improve data placement [28], [29]. 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 [30] 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. Suleman et al [31] propose migrating the execution of critical sections to a powerful core for performance improvement. Core Salvaging [32] exploits inter-core redundancy to provide fault tolerance via execution migration. Thread motion [11] exchanges running threads to provide fine-grain power management.

VII. Conclusions and future work

We have introduced the EM^2, a scalable architecture for large-scale shared-memory multicores. Our approach significantly lowers off-chip memory access rates, instead taking advantage of the much faster on-chip interconnect to achieve average performance gains of 18% while keeping silicon area competitive.

EM^2 offers three natural avenues to improving performance: (a) lowering migration rates, (b) increasing migration rate performance, and (c) optimizing applications for execution migration (cf. Formula 1 in Section II-A). These neatly outline our future plans: we will aim to lower migration rates with better data placement algorithms, optimize migration performance via careful network interconnect design, and investigate OS and application optimizations appropriate in the context of execution migration.

References

[1] S. Rusu, S. Tam *et al.*, "A 45nm 8-core enterprise Xeon® processor," in *A-SSCC*, 2009, pp. 9–12.

[2] S. Bell, B. Edwards *et al.*, "TILE64 - processor: A 64-Core SoC with mesh interconnect," in *ISSCC*, 2008, pp. 88–598.

[3] S. R. Vangal, J. Howard *et al.*, "An 80-Tile Sub-100-W TeraFLOPS processor in 65-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 43, no. 1, pp. 29–41, 2008.

[4] T. R. Halfhill, "Looking beyond graphics," In-Stat Whitepaper, 2009.

[5] S. Borkar, "Thousand core chips: a technology perspective," in *DAC*, 2007, pp. 746–749.

[6] N. Hardavellas, M. Ferdman *et al.*, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *ISCA*, 2009, pp. 184–195.

[7] "Assembly and packaging," *International Technology Roadmap for Semiconductors*, 2007.

[8] J. E. Miller, H. Kasture *et al.*, "Graphite: A distributed parallel simulator for multicores," in *HPCA*, 2010, pp. 1–12.

[9] M. M. Bach, M. Charney *et al.*, "Analyzing parallel programs with pin," *Computer*, vol. 43, pp. 34–41, 2010.

[10] S. Woo, M. Ohara *et al.*, "The SPLASH-2 programs: characterization and methodological considerations," in *ISCA*, 1995, pp. 24–36.

[11] K. K. Rangan, G. Wei *et al.*, "Thread motion: fine-grained power management for multi-core systems," in *ISCA*, 2009, pp. 302–313.

[12] R. Alverson, D. Callahan *et al.*, "The tera computer system," 1990.

[13] W. J. Dally and B. Towles, *Principles and practices of interconnection networks*. Morgan Kaufmann, 2003.

[14] A. Kumar, P. Kundu *et al.*, "A 4.6tbits/s 3.6ghz single-cycle noc router with a novel switch allocator," in *in 65nm CMOS, ICCD*, 2007.

[15] S. Thoziyoor, J. H. Ahn *et al.*, "A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies," in *ISCA*, 2008, pp. 51–62.

[16] www.synopsys.com, "Synopsys design compiler."

[17] S. C. Woo, J. P. Singh *et al.*, "The performance advantages of integrating block data transfer in cache-coherent multiprocessors," vol. 29, no. 11, pp. 219–229, 1994.

[18] M. T. Hill, H. J. S. Dorren *et al.*, "A fast low-power optical memory based on coupled micro-ring lasers." *Nature*, vol. 432, no. 7014, pp. 206–9, 2004.

[19] J. Liu, J. Psota *et al.*, "Atac: A manycore processor with on-chip optical network," *MIT-CSAIL-TR-2009-018*.

[20] H. Garcia-Molina, R. Lipton *et al.*, "A massive memory machine," *IEEE Trans. Comput.*, vol. C-33, pp. 391–399, 1984.

[21] P. Michaud, "Exploiting the cache capacity of a single-chip multi-core processor with execution migration," in *HPCA*, 2004, pp. 186–195.

[22] K. Chakraborty, P. M. Wells *et al.*, "Computation spreading: employing hardware migration to specialize CMP cores on-the-fly," in *ASPLOS*, 2006, pp. 283–292.

[23] M. Kandemir, F. Li *et al.*, "A novel migration-based NUCA design for chip multiprocessors," in *SC*, 2008, pp. 1–12.

[24] W. C. Hsieh, P. Wang *et al.*, "Computation migration: enhancing locality for distributed-memory parallel systems," in *PPOPP*, 1993, pp. 239–248.

[25] S. Cho and L. Jin, "Managing distributed, shared L2 caches through OS-Level page allocation," in *MICRO*, 2006, pp. 455–468.

[26] M. Awasthi, K. Sudan *et al.*, "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," in *HPCA*, 2009, pp. 250–261.

[27] S. Boyd-Wickizer, R. Morris *et al.*, "Reinventing scheduling for multicore systems," in *HotOS*, 2009.

[28] M. Chaudhuri, "PageNUCA: selected policies for page-grain locality management in large shared chip-multiprocessor caches," in *HPCA*, 2009, pp. 227–238.

[29] K. Sudan, N. Chatterjee *et al.*, "Micro-pages: increasing DRAM efficiency with locality-aware data placement," *SIGARCH Comput. Archit. News*, vol. 38, no. 1, pp. 219–230, 2010.

[30] M. Zhang and K. Asanović, "Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *ISCA*, 2005, pp. 336–345.

[31] M. A. Suleman, O. Mutlu *et al.*, "Accelerating critical section execution with asymmetric multi-core architectures," in *ASPLOS*, 2009, pp. 253–264.

[32] M. D. Powell, A. Biswas *et al.*, "Architectural core salvaging in a multi-core processor for hard-error tolerance," in *ISCA*, 2009, pp. 93–104.