EM^2: A Scalable Shared-memory Multicore Architecture

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

August 27, 2010

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 home core for that data and continues execution. Using detailed simulations of a 256-core chip multiprocessor on the SPLASH-2 benchmarks, we show that EM^2 outperforms directory-based cache-coherence 1.13× on average using a high-bandwidth electrical network and 2× with an optical network. In addition, because of the dramatic reduction in off-chip memory accesses, EM^2 improves energy consumption by 1.3× on average, and the energy-delay product by up to 5.4× over cache coherence.

I. Introduction

Four to eight general-purpose cores on a die are now common across the spectrum of computing machinery [1], and designs with many more cores are not far behind [2], [3]; pundits confidently predict thousands of cores by the end of the decade [4]. Multicore scalability is, however, critically constrained by the *off-chip memory bandwidth wall* [4], [5]: while on-chip transistor counts grow at ca. 59% per technology generation, the package pin density—and with it, the off-chip memory bandwidth—increases at a much slower rate of ca. 5% [6], making performance *worse* as the number of cores on the chip grows. To address this problem, today's multicores integrate large private and shared caches: the hope is that large caches can hold the working sets of the active threads, thereby reducing the number of off-chip memory accesses; shared caches, however, do not scale beyond a few cores [1], and many private caches remain the only practical option for large-scale multicores.

Systems that preserve the programmatically convenient abstraction of a shared memory, therefore, generally implement some form of cache coherence: in large multicores lacking common buses, this usually takes the form of directory-based cache-coherence hardware. Efficient protocols are, however, complex to analyze and implement; worse yet, protocol transactions add delays to the already high cost of accessing off-chip memory, while data replication and directory size constraints limit the efficient use of cache resources and raise the off-chip memory access rates even more.

In this manuscript, we outline the Execution Migration Machine (EM^2) architecture, which addresses this problem by ensuring that no writable data are ever shared among caches. In EM^2, when a thread needs access to an address cached on another core, the hardware efficiently migrates the computation's execution context to the core where the memory is (or is allowed to be) cached and continues execution there. Unlike in a traditional architecture, where memory performance is often determined by the size of *each* core cache and the number of shared addresses, memory performance under EM^2 follows the *combined* size of *all* core caches: eschewing replication allows significantly lower cache miss rates and better overall performance.

Initial simulations for a 256-core chip multiprocessor show that, depending on the application and the available network resources, EM^2 architecture can significantly improve memory access latencies and overall application performance. A simple, easily analyzed design with potential for significant energy savings, EM^2 employs a novel architecture that allows hardware-level, pipelined thread migrations. Although the architecture performs best on a high-bandwidth, low-latency network, such as an on-chip optical interconnect [7], [8], [9], [10], [11], it is also competitive on electrical interconnects available today, and provides a scalable path to the 1000-core era.

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 hardware-based migration algorithm.
3) We show that using a low-latency electrical-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, the average parallel completion time and dynamic energy consumption of applications improve by 1.13× and 1.32× respectively. Even better, using an optical interconnect EM^2 performance improves by 2×.

In the remainder of this paper, we describe the EM^2 architecture (Section II), provide motivation and intuition for EM^2 (Section III), present our simulation methodology (Section IV), characterize the architecture design space (Section V), and evaluate the proposed design across performance and energy tradeoffs (Sections VI). Finally, we review related work (Section VII) and outline future research directions (Section VIII).

II. The EM^2 architecture

The essence of traditional cache coherence (CC) 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 or exclusively owned. EM^2, on the other hand, brings the *computation* to the data: when a memory instruction requests an address not cached by the current core, the execution context (architecture state and TLB entries) moves to the core that is *home* for that data. The physical address space in the system is partitioned among the cores, 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. (This assignment is independent of the number of memory controllers.) When a core 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, requesting *A* from its cache hierarchy (and potentially accessing DRAM).

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 ensured. (For sequential consistency to be violated, multiple threads must observe multiple writes in different order, which is only possible if they disagree about the value of some variable, for example, when their caches are out of sync. Because EM^2 never replicates data, this situation never arises). Atomic locks work trivially, with multiple contexts sequentialized on the core where the lock address is located, and no longer ping-pong among core local caches as in cache coherence.

A. Data Placement

The assignment of addresses to cores affects the performance of an EM^2 design in two ways: (a) directly because context migrations pause thread execution, and (b) indirectly by influencing cache performance. On the one hand, spreading frequently used addresses evenly among the cores 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 traffic.

In an EM^2 architecture, 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, an EM^2-aware operating system chooses where the relevant page should be cached by mapping the virtual page to a physical address range assigned to a specific core. Since the OS knows which thread causes a page fault, more sophisticated heuristics are possible: for example, the OS can 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 (cf. Section V).

B. Migration Framework

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

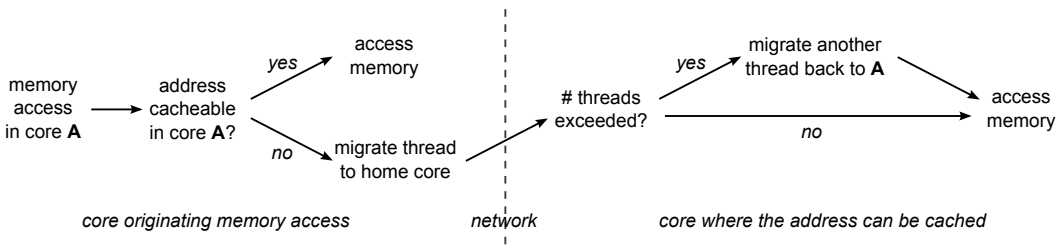


Fig. 1. In an EM^2 architecture, memory accesses to addresses not assigned to the local core cause the execution context to be migrated to the core.

about 1.5 Kbits [12]. Although on-chip electrical networks today are not generally designed to carry that much data in parallel, 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 1 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*. Although 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 creating a core thrashing effect. In addition, the SWAP scheme also puts significantly more stress on the network: not only are the migrations symmetrical, core thrashing 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 V-A), and resorting to evictions only when the number of hardware contexts running at the target core would exceed available resources.

C. Memory Access Cost

Overall, the average memory latency (AML) under EM^2 has three components: cache access (for cache hits and misses), off-chip memory access (for cache misses), and context migration cost (for core misses):

$$AML = cost_{\$access} + rate_{\$miss} \times cost_{\$miss} + rate_{coremiss} \times cost_{contextxfer}$$

While $cost_{\$access}$ mostly depends on the cache technology itself, EM^2 improves performance by optimizing the other variables. In the following sections we show how our architecture improves $rate_{\$miss}$ when compared to a cache-coherent baseline, discuss several ways to keep $rate_{coremiss}$ low, and argue that today's interconnect technologies keep $cost_{contextxfer}$ sufficiently low to ensure good performance.

III. Why EM^2 is not a crazy idea

A. Memory access costs

Intuitively, replacing off-chip memory traffic due to (hopefully infrequent) cache misses with possibly much more frequent on-chip thread migration traffic may not seem like an improvement. To gain some intuition about the memory access latencies under cache coherence and EM^2, we broke down memory accesses into fundamental components (such as DRAM access itself or coherence traffic) and created a model to estimate average memory access latency (Figure 2). We then measured parameters like cache and core miss rates (shown in Table III) by running the LU benchmark in our simulator under both an MSI coherence protocol and EM^2 (we chose LU because it had the highest core miss rate out of all SPLASH-2 benchmarks, which avoids pro-EM^2 bias). Applying these to the measured parameters (see Figure 3) shows that, on the average, EM^2 memory accesses in the LU benchmark take **1.7**$\times$ less time.

Although the average memory latency is decidedly the main component of wall-clock application performance, data sharing in EM^2 might cause multiple threads to execute on one core and can affect performance; indeed, our simulations, which accurately

Parameter	CC	EM^2
$cost_{L1}$	2 cycles	2 cycles
$cost_{L2}$ *(in addition to L1)*	5 cycles	5 cycles
$cost_{\$invalidate}$, $cost_{\$flush}$	7 cycles	—
cache line size	64 bytes	64 bytes
average network distance	12 hops	12 hops
$cost_{per\text{-}hop}$	1 cycle	1 cycle
$cost_{congestion}$	10 cycles	12 cycles
$cost_{DRAM\,latency}$	235 cycles	235 cycles
$cost_{DRAM\,serialization}$ (1 cache line)	50 cycles	50 cycles
$cost_{dir\,lookup}$	10 cycles	—
flit size	256 bits	256 bits
execution context (32-bit x86)	—	1.5 Kbit [12]
$rate_{L1\,miss}$	11.6%	2%
$rate_{\$miss}$ (both L1 and L2 miss)	9.2%	0.1%
$rate_{core\,miss,EM^2}$	—	66%
$rate_{rdI}$, $rate_{rdS}$, $rate_{rdM}$, $rate_{wrI}$, $rate_{wrS}$, $rate_{wrM}$	16.84%, 20.69%, 25.03%, 12.35%, 24.97%, 0.12%	—

TABLE I
VARIOUS PARAMETER SETTINGS FOR THE ANALYTICAL COST MODEL FOR THE LU BENCHMARK

modeled this effect, showed a 1.5× improvement in *parallel completion time* of EM^2 vs. CC for LU (see Section VI-C for details). Although these effects heavily depend on the application access patterns, we point out two factors which mitigate this serialization effect. Firstly, our cores can simultaneously hold (but not execute) two execution contexts (see Section V-A for details): this ensures that while one thread is stalled on a memory access (even a cache hit) or in mid-migration, another thread can make progress. Secondly, our address-to-core data placement heuristic (see Sections II-A and V-D) attempts to keep each thread on a separate core as much as possible precisely to limit the serialization effect. Empirical observations (Section VI-C) suggest that the combination of these two factors alleviates in-core thread serialization enough that average memory latency is a good proxy for parallel application performance.

The memory latency model also gives a flavor for how EM^2 might scale as the number of cores grows. Centralized effects like off-chip memory contention and network congestion around directories, which limit performance in CC, will only increase as the ratio of core count to off-chip memory bandwidth increases. Performance-limiting costs under EM^2, on the other hand, are either decentralized (core migrations are distributed across the chip) or much smaller (contention for off-chip memory is much lower because cache miss rates are small compared to CC), and will scale more gracefully. EM^2 also benefits much more from interconnect improvements, such as a futuristic high-bandwidth, low-latency on-chip optical interconnect. Under EM^2, migration costs account for most of the amortized memory access latency, and reducing network latency significantly improves performance, while in cache coherence DRAM access costs dominate, and an improved interconnect will affect performance much less: replacing the network cost in our model with an optical network model, we arrive at 33.6 cycles for CC and 9 cycles for EM^2, giving EM^2 a **3.7**× advantage.

$$AML_{CC} = cost_{\$access,CC} + rate_{\$miss,CC} \times cost_{\$miss,CC} \quad (1)$$

$$cost_{\$access} = cost_{L1} + rate_{L1\,miss} \times cost_{L2} \quad (2)$$

$$cost_{rdI,wrI,rdS} = cost_{core\to dir} + cost_{dir\,lookup} + cost_{DRAM} + cost_{dir\to core} + cost_{\$\,insert} \quad (3)$$

$$cost_{wrS} = cost_{core\to dir} + cost_{dir\,lookup} + cost_{dir\to core} + cost_{\$\,invalidate} + cost_{core\to dir} + cost_{DRAM} + cost_{dir\to core} + cost_{\$\,insert} \quad (4)$$

$$cost_{rdM} = cost_{core\to dir} + cost_{dir\,lookup} + cost_{dir\to core} + cost_{\$\,flush} + cost_{core\to dir} + cost_{DRAM} + cost_{dir\to core} + cost_{\$\,insert} \quad (5)$$

$$cost_{wrM} = cost_{core\to dir} + cost_{dir\,lookup} + cost_{dir\to core} + cost_{\$\,flush} + cost_{core\to dir} + cost_{dir\to core} + cost_{\$\,insert} \quad (6)$$

$$cost_{\$miss,CC} = rate_{rdI,wrI,rdS} \times cost_{rdI,wrI,rdS} + rate_{wrS} \times cost_{wrS} + rate_{rdM} \times cost_{rdM} + rate_{wrM} \times cost_{wrM} \quad (7)$$

$$cost_{DRAM,CC} = cost_{DRAM\,latency} + cost_{DRAM\,serialization} + cost_{DRAM\,contention} \quad (8)$$

$$cost_{message\,xfer} = cost_{\to,CC} + \left\lceil \frac{pkt\,size}{flit\,size} \right\rceil \quad (9)$$

$$cost_{\to,CC} = \#hops \times cost_{per\text{-}hop} + cost_{congestion,CC} \quad (10)$$

(a) MSI cache coherence protocol

$$AML_{EM^2} = cost_{\$access,EM^2} + rate_{\$miss,EM^2} \times cost_{\$miss,EM^2} + rate_{core\,miss,EM^2} \times cost_{context\,xfer} \quad (11)$$

$$cost_{\$access\,EM^2} = cost_{L1} + rate_{L1\,miss,EM^2} \times cost_{L2} \quad (12)$$

$$cost_{\$miss,EM^2} = cost_{core\to mem} + cost_{DRAM} + cost_{mem\to core} \quad (13)$$

$$cost_{DRAM,EM^2} = cost_{DRAM\,latency} + cost_{DRAM\,serialization} + cost_{DRAM\,contention} \quad (14)$$

$$cost_{message\,xfer} = cost_{\to,EM^2} + \left\lceil \frac{pkt\,size}{flit\,size} \right\rceil \quad (15)$$

$$cost_{\to,EM^2} = \#hops \times cost_{per\text{-}hop} + cost_{congestion,EM^2} \quad (16)$$

(b) EM^2

Fig. 2. Average memory latency (AML) costs for our MSI cache coherence protocol and for EM^2. The significantly less complicated description for EM^2 suggests that EM^2 is easier to reason about and implement. The description for a protocol such as MOSI would be significantly bigger than for MSI.

B. Coherence Protocol Variations

It is natural to ask whether MSI is the correct protocol for large-scale chip multiprocessors. On the one hand, data replication can be completely abandoned by only allowing modified or invalid states (MI); on the other hand, in the presence of data replication, off-chip access rates can be lowered via protocol extensions such as an *owned* state (MOSI) combined with cache-to-cache transfers whenever possible.

To evaluate the impact of these variations, we compared the average memory latencies for various SPLASH-2 benchmarks under MSI, MI, MOSI, and EM^2 (using parameters from Table III). As shown in Figure 4, MI exhibits by far the worst memory latency: although it may at first blush seem that MI removes sharing and should thus improve cache utilization much like EM^2, in actuality eschewing the *S* state only spreads the sharing—and the cache pollution which leads to capacity misses—over time when compared with MSI. At the same time, MI gives up the benefits of read-only sharing and suffers many more cache

Interconnect traversal costs. Both protocols incur the cost of on-chip interconnect transmissions to retrieve data from memory, migrate thread contexts (in EM^2), and communicate among the caches (in CC). In the interconnect network model we assume a 16×16 mesh with one-cycle-per-hop 256-bit flit pipelined routers, an average distance of 12 hops with network congestion overheads consistent with what we observed for LU), making the network transit cost

$$\begin{aligned} cost_{\rightarrow,CC} &= 12+10 = 22, \text{ and} \\ cost_{\rightarrow,EM^2} &= 12+12 = 24. \end{aligned} \tag{17}$$

Delivering a message adds a load/unload latency dependent on the packet size: for example, transmitting the 1.5 Kbit EM^2 context requires

$$cost_{context\,xfer} = 24 + \frac{1536 \text{ bits}}{256 \text{ bits}} = 30. \tag{18}$$

By the same token, in EM^2, transmitting a single-flit request takes 25 cycles and transferring a 64-byte cache line needs 26; in CC, these become 23 and 24 cycles.

Off-chip DRAM access costs. In addition to the DRAM latency itself, off-chip accesses may experience a queueing delay due to contention for the DRAM itself; moreover, retrieving a 64-byte cache line must be serialized over many cycles (Equations 8 and 14). Because there were dramatically fewer cache misses under EM^2, we observed relatively little DRAM queue contention (1 cycle), whereas the higher off-chip access rate of CC resulted in significantly more contention on average (25 cycles):

$$\begin{aligned} cost_{DRAM,EM^2} &= 235+50+1 = 286 \\ cost_{DRAM,CC} &= 235+50+25 = 310. \end{aligned} \tag{19}$$

EM^2 memory access latency. Given the network and DRAM costs, it's straightforward to compute the average memory latency (AML) for EM^2 which depends on the cache access cost and, for every cache miss, the cost of accessing off-chip RAM; under EM^2, we must also add the cost of migrations caused by core misses (Equation 11).

The cache access cost is incurred for every memory request and depends on how many accesses hit the faster L1 cache: for EM^2,

$$cost_{\$\,access\,EM^2} = 2+2\%\times 5 = 2.1. \tag{20}$$

Each cache miss under EM^2 contacts the memory controller, retrieves a cache line from DRAM, and sends it to the requesting core:

$$cost_{\$\,miss,EM^2} = 25+286+26 = 337. \tag{21}$$

Finally, then, we arrive at the average memory latency:

$$AML_{EM^2} = 2.1+0.1\%\times 337+66\%\times 30 = 22.2. \tag{22}$$

Cache coherence memory access latency. Since CC does not need to migrate execution contexts, memory latency depends on the cache access and miss costs (Equation 1). Because fewer accesses hit the L1 cache, even the cache access cost itself is higher than under EM^2:

$$cost_{\$\,access} = 2+11.6\%\times 5 = 2.56. \tag{23}$$

The cost of a cache miss is, however, significantly more complex, as it depends on the kind of access (read or write, respectively *rd* and *wr* below) and whether the line is cached nowhere (*I* below) or cached at some other node in shared (*S*) or modified (*M*) state:

- Non-invalidating requests (49.9% of L2 misses for LU)—loads and stores with no other sharers, as well as loads when there are other read-only sharers—contact the directory and retrieve the cache line from DRAM:

$$cost_{rdI,wrI,rdS} = 23+10+310+26+7 = 376. \tag{24}$$

- Stores to data cached in read-only state elsewhere (25%) must invalidate the remote copy before retrieving the data from DRAM: in the best case of only one remote sharer,

$$cost_{wrS} = 23+10+23+7+23+310+24+7 = 427. \tag{25}$$

- Loads of data cached in modified state elsewhere (25.1%) must flush the modified remote cache line and write it back to DRAM before sending the data to the requesting core via a cache-to-cache transfer:

$$cost_{rdM} = 23+10+23+7+24+310+24+7 = 428. \tag{26}$$

- Stores to data cached in modified state elsewhere (0.1%) must also flush the cache line but avoids a write-back to DRAM by sending the data to the requesting core via a cache-to-cache transfer:

$$cost_{wrM} = 23+10+23+7+24+24+7 = 118. \tag{27}$$

Combining the cases with their respective observed rates for each (cf. Table I), we arrive at the average cost of a cache miss under CC:

$$cost_{\$\,miss,CC} = (16.84\%+12.35\%+20.69\%)\times 376+25.03\%\times 427+20.69\%\times 428+0.12\%\times 118 = 383.12, \tag{28}$$

and, finally, the average memory cost for CC:

$$AML_{CC} = 2.56+9.2\%\times 383.12 = 37.8, \tag{29}$$

over **1.7**× greater than under EM^2.

Fig. 3. Average memory latency (AML) comparison for EM^2 and a cache-coherent MSI protocol, based on the model from Figure 2.

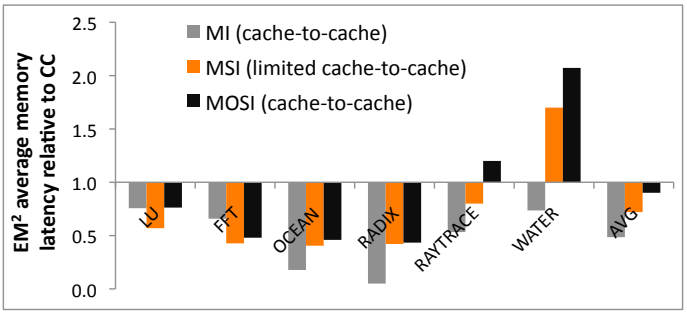


Fig. 4. Average memory latency for EM^2 relative to CC protocol variations using parameters from Table III. Instead of reducing cache pollution due to sharing, MI only spreads it over time while giving up the benefits of sharing, in the end performing much worse than MSI. MOSI in addition to cache-to-cache transfers for accesses to shared data allow modified cache lines to be exchaged between caches without writeback. MOSI reduces DRAM accesses at the expense of an increase in coherence traffic and protocol implementation and validation complexity.

evictions: its cache miss rates were 2.3× greater than under MSI.

While our MSI implementation supports all cache-to-cache transfers permitted by the protocol, we reasoned that the more complex MOSI protocol stands to benefit from using cache-to-cache transfers more extensively to avoid writing back modified data to off-chip RAM: instead of accessing DRAM for the data, a designated *owner* cache becomes responsible for handling the data requests. Our analysis shows that, while cache-to-cache transfers result in many fewer DRAM accesses, they instead induce significantly more coherence traffic (even shared reads now take 4 messages) and, in the end, MOSI memory latency is still higher than EM^2 while paying the cost of significantly increased protocol, implementation and validation complexity. Since our simulations indicate (Figure 4) that MSI offers a good trade-off between performance and implementation complexity for large-scale systems, we use MSI as a baseline for comparison in the remainder of this paper and relegate investigation of other protocols to future research (see Section VIII).

C. The Problem of Data Sharing

As our analysis shows, the main benefit of EM^2 comes from improving on-chip cache utilization by not replicating writable shared data and minimizing cache capacity/conflict misses. To estimate the impact of data sharing in future massively parallel application workloads, we created synthetic benchmarks that randomly access addresses with varying degrees of read-only sharing and read-write sharing (see Table II). The benchmarks vary along two axes: the fraction of instructions that access read-only data, and the degree of sharing of the shared data: for example, for read-write shared data, degree *d* denotes that

% of non-memory instructions	70%
% of memory instructions accessing shared data	10%
% of memory instructions accessing private data	20%
% of read-only data in shared data	{25%, 75%, 100%}
Load:store ratio	2:1
Private data per thread	16 KB
Total shared data	1 MB
Degree of sharing	{1, 2, 4, 8, 32, 64, 128, 256}
Number of instructions per thread	100,000

TABLE II
SYNTHETIC BENCHMARK SETTINGS

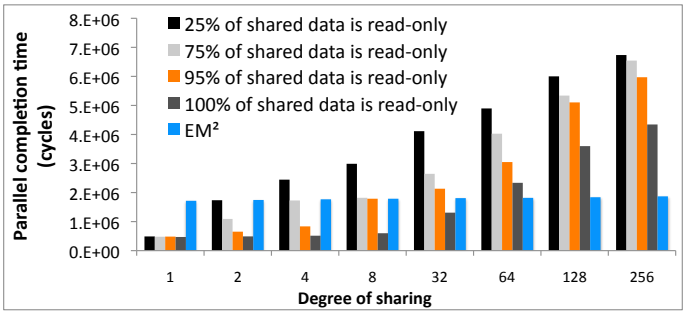


Fig. 5. The performance of CC degrades as the degree of sharing increases: for read-write sharing this is due to cache evictions, and for read-only sharing due to reduced core-local cache effectiveness when multiple copies of the same data are cached.

this data can be read/written by up to d sharers or threads. We then simulated the benchmarks using our cache-coherent (MSI) model (Table III), and measured parallel application performance (which, unlike our memory latency model, includes effects not directly attributable to memory accesses like serialization or cache/directory eviction costs).

Figure 5 shows that cache coherence performance worsens as the degree of sharing increases. This is for two reasons: one is that a write to shared data requires that all other copies be invalidated (this explains the near-linear growth in parallel completion time when most accesses are writes), and the other is that even read-only sharing causes one address to be stored in many core-local caches, reducing the amount of cache left for other data (this is responsible for the slower performance decay of the 100% read-only benchmarks). EM^2 performance, on the other hand, reduced very slowly with degree of sharing: this is because (a) each address, even shared by multiple threads, was still assigned to only one cache, leaving more total cache capacity for other data, and (b) the uniform random pattern of accesses, combined with the large 1 MB data region, made precisely simultaneous accesses by many threads—and thus thread serialization effects—unlikely. Thus, in this benchmark, EM^2 is limited only by the very high execution context migration rate (ca. 67% of memory accesses), which is also due to the random memory access pattern and independent of sharing degree. (For EM^2 performance on real application loads with non-random access patterns, see Section VI).

This result highlights the problem of data sharing in traditional cache coherent systems. Even in the unrealistic case where all shared data is read-only, the higher cache miss rates of cache coherence cause substantial performance degradation for degrees of sharing greater than 32; when more and more of the shared data is read-write, performance starts dropping at lower and lower degrees of sharing. This effect is bound to increase as core counts grow, and motivates further evaluation of EM^2.

IV. Methods

We use Pin [13] and Graphite [14] to model the proposed EM^2 architecture as well as the cache-coherent baseline. Pin enables runtime binary instrumentation of parallel programs, including the SPLASH-2 [15] benchmarks we use here; Graphite implements a tile-based multicore, memory subsystem, and network, modeling performance and ensuring functional correctness.

The default settings used for the various system configuration parameters are summarized in Table III; any deviations are noted when results are reported. In experiments comparing EM^2 against CC, the parameters for both were identical, except for (a) the memory directories which are not needed for EM^2 and were set to sizes recommended by Graphite on basis of the

Parameter	Settings
Number of cores	256
Multithreading	2 threads, 1 issue-slot
L1 instruction and data cache per core	32 KB, 16 KB
L2 cache size per core	64 KB
Electrical network	Mesh, XY routing, 1 cycle per hop
Network flit size	256 bits
Migration scheme	ORIGINAL, 4 KB page size
Coherence protocol	Direcory-based MSI
Memory	30 GB/s bandwidth, 75 ns latency

TABLE III
SYSTEM CONFIGURATIONS USED

Component	#	Total area (mm^2)	Read energy (nJ/instance)	Write energy (nJ/instance)	Details
EM^2 regFile	256	2.48	0.005	0.002	4-Rd, 4-Wr ports; 64x24 bits
EM^2 router	256	15.67	0.022	0.007	5-Rd, 5-Wr ports; 256x20 bits
CC router	256	3.78	0.006	0.003	5-Rd, 5-Wr ports; 64x20 bits
Directory cache	8	2.45	0.192	0.186	256 KB cache (16-way assoc)
L2 Cache	256	26.65	0.086	0.074	64 KB (4-way assoc)
L1 Data Cache	256	6.44	0.034	0.017	16 KB cache (2-way assoc)
Off-chip DRAM	8	N/A	6.333	6.322	1 GB RAM

TABLE IV
AREA AND ENERGY ESTIMATES

total cache capacity in the simulated system, (b) the 2-way multithreaded cores which are not needed for CC system, and (c) the interconnection network.

A. On-chip interconnect

Experiments were performed using Graphite's model of an electrical mesh network with XY routing as well as with an optical network model [7]. Each packet on the network is partitioned into fixed size flits. As EM^2 may require high bandwidth due to larger size of execution context (architecture registers and TLB entries account for 1.5 Kbits in an x86 [12]), we vary the flit size between 64-bits to 256-bits for the electrical network; for the optical network, the flit size is fixed at 256-bits to model the additional available bandwidth of an optical network. The choice of a flit size is a function of packet injection rate and the area overhead of additional buffers and crossbar in the router. Since modern network-on-chip routers are pipelined [16], we argue that modeling a 1-cycle per hop router latency [17] is reasonable for both electrical and optical networks; we account for the appropriate pipeline latencies associated with delivering a packet. In addition to the fixed per-hop latency, contention delays are modeled for both the electrical and optical networks; the queuing delays at the router are estimated using a probabilistic model similar to the one proposed in [18].

B. Area and energy estimation

We assume 32 nm process technology and use CACTI [19] to estimate the area requirements of the on-chip caches and interconnect routers. To estimate the area overhead of extra hardware context in the 2-way multithreaded core for EM^2, we

used Synopsys Design Compiler [20] to synthesize the extra logic and register-based storage. We also use CACTI to estimate the dynamic energy consumption of the caches, routers, register files, and DRAM. The area and dynamic energy numbers used in this paper are summarized in Table IV. We implemented several energy counters (for example the number of DRAM reads and writes) in our simulation framework to estimate the total energy consumption of running SPLASH-2 benchmarks for both CC and EM^2. Note that DRAM only models the energy consumption of the RAM and the I/O pads and pins will only add to the energy cost of going off-chip.

C. Measurements

Our experiments used a set of SPLASH-2 benchmarks: FFT, LU, OCEAN, RADIX, RAYTRACE, and WATER. Each application was run to completion using the recommended input set for the number of cores used, except as otherwise noted. For each simulation run, we tracked the total application completion time, the parallel work completion time, the percentage of memory accesses causing cache hierarchy misses, and the percentage of memory accesses causing migrations. While the total application completion time (wall clock time from application start to finish) and parallel work completion time (wall clock time from the time the second thread is spawned until the time all threads re-join into one) show the same general trends, we focused on the parallel work completion time as a more accurate metric of average performance in a realistic multicore system with many applications.

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.

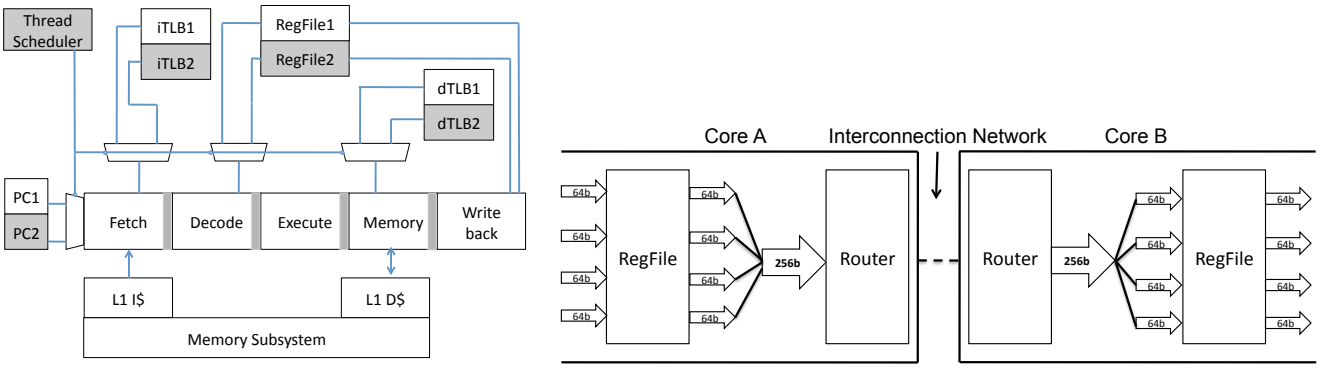
V. CHARACTERIZATION

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

A. Multithreaded Core: Sharing & serialization

EM^2 employs a multithreading architecture [21], primarily to enable efficient pipelining of execution migrations. A multi-threaded core, shown in Figure 6a, 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 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, necessitating a trade-off study.

EM^2 permits multiple threads to run in a single pipeline in a round-robin fashion; 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.



(a) The architecture of a 2-way multithreaded core for EM^2

(b) Microarchitecture for a single context transfer in EM^2

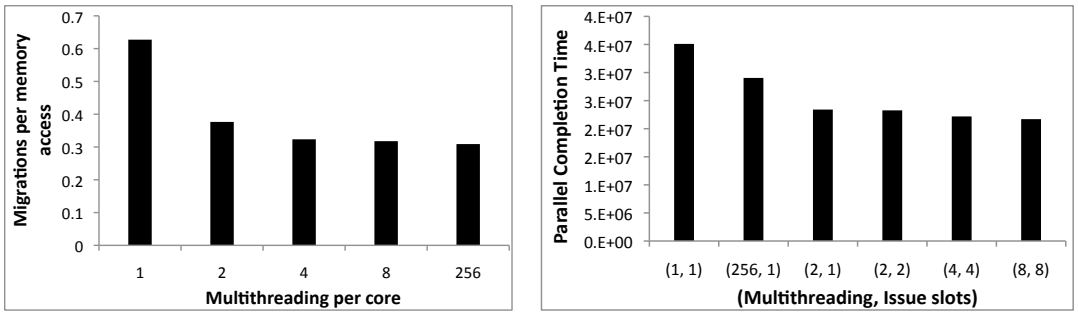
Fig. 6. Proposed EM^2 core's differences from a single-threaded pipeline are highlighed. For a context transfer, the register file of the originating core is unloaded onto the router, transmitted across the network and finally loaded onto the home core's register file via the router.

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 7a 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 7b) 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 further evaluation of EM^2. The context transfer flow through an EM^2 architecture is shown in Figure 6b.

B. Instruction cache

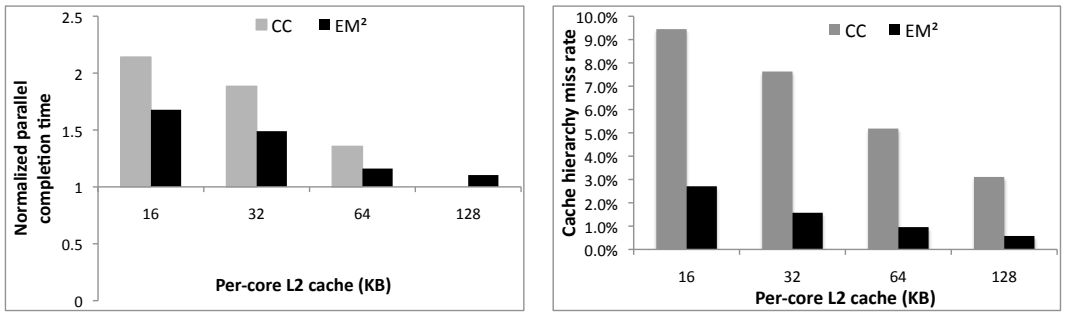
Since the migrated execution context does not contain instruction cache entries, the target core might not contain the relevant instruction cache lines and a thread might incur an instruction cache miss immediately upon migration. To evaluate the potential impact of this phenomenon, we compared L1 instruction cache miss rates for EM^2 and CC; simulations on our SPLASH-2



(a) # migrations per memory access

(b) application completion time (cycles)

Fig. 7. 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.



(a) application parallel completion time, normalized to CC with 128 KB L2 cache per core

(b) cache hierarchy miss rate (%)

Fig. 8. Effect of cache sizes on performance using parameters from Table III.

multithreaded benchmarks showed an average instruction cache miss rate of 0.19% under EM^2 as compared to 0.27% for the CC architecture. 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.

C. 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 CC baseline and our EM^2 design, cache miss rates are much lower for EM^2 and equivalent or better performance can be achieved with smaller on-chip caches (Figure 8). When caches are large, on the other hand, they tend to fit most of the working set of our SPLASH-2 benchmarks and both EM^2 and CC almost never miss the cache: in that regime CC tends to outperform EM^2. This is, however, not a realistic scenario in a system concurrently running many applications: we empirically observed that as the input data set size increases, larger and larger caches are required for CC to keep up with EM^2. To avoid pro-EM^2 or pro-CC bias, we chose 64 KB data caches as our default configuration because it offers a reasonable performance tradeoff and, at the same time, results in a massive 25 Mbytes of on-chip total cache.

D. 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. 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.

The STRIPED scheme distributes the virtual address space of an application statically and evenly among the available cores on a single-page (for us, 4 K) granularity: page 0 is mapped to core 0, page 255 to core 255, page 256 to core 0 again, and so on. It is the simplest to implement as the target core can be obtained from the least significant bits of the page number.

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. Using the existing virtual memory subsystem, page-to-core mapping is delayed until some thread actually reads or writes

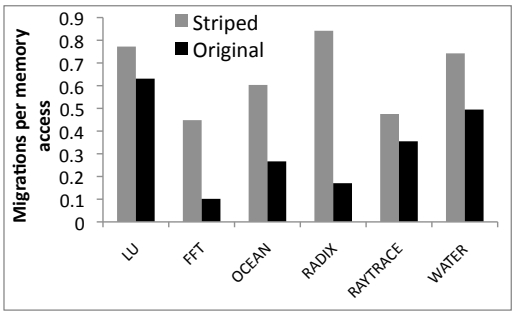
an address on the given page; on the first access, the page is mapped to the accessing thread's home core and remains there for the duration of the execution. (While superficially similar to the first-touch heuristic for reducing cc-NUMA memory latency [22], the ORIGINAL scheme instead aims to keep each thread on its home core for as much of its runtime as possible).

Figure 9a 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. We chose the ORIGINAL scheme for our evaluation because it significantly lowered the execution migrations from 65% for STRIPED to 33%.

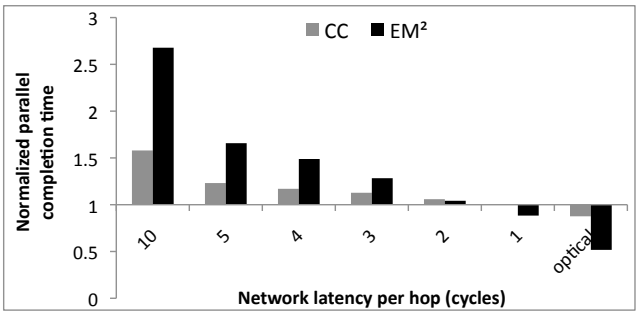
Although a detailed analysis of possible data placement algorithms is beyond the scope of this paper, the present discussion warrants a few observations. First, an optimal algorithm will treat addresses differently based on whether they are private to a thread (in which case the OS might try to keep 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). 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. 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.

E. 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. 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 (classical network-on-chip routers employ pipelines with three to four stages [16] and recently a 1-cycle electrical



(a) migration rate comparisons under STRIPED and ORIGINAL data placement schemes



(b) application parallel completion time, normalized to CC with 1-cycle per hop electrical network

Fig. 9. Page-to-core mapping strategy can significantly affect the core miss rate. Fewer core misses mean less pressure on the interconnection network for EM^2 and therefore higher performance. Additionally the network itself can be improved: for example, an optical on-chip network enables low-latency communication at higher bandwidth compared to a single cycle per hop electrical network, giving EM^2 a $\sim 2\times$ boost in performance.

router has been implemented [17]), as well as an optical network where each transfer takes 1-cycle (in addition to load/unload between electrical and optical components as well as contention delays). 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 9b). EM^2 outperforms or stays competitive for an equivalent cache-coherent design on an optical network and an electrical network with 1-, and 2-cycle per-hop latency, the trend reverses for a 3-cycle per-hop latency, and EM^2 does significantly worse when each hop takes 10 cycles.

F. Area tradeoffs

In terms of silicon area, EM^2 compares fairly with a directory-based cache-coherent design: while EM^2 saves area by eschewing the directories, it needs extra silicon for the additional execution context in the 2-way multithreaded core, and, at the end of the day, the areas are almost precisely the same (see Table IV).

Although our performance results assumed equal-bandwidth networks for both EM^2 and CC to ensure fairness, we reasoned that EM^2 requires a higher-bandwidth network for context migrations and a cache-coherent design might save area by integrating a lower-bandwidth network. In this case, an EM^2 design based on a 256-bit-flit network is ca. 12 mm^2 larger than a CC design built on 64-bit-flit—a mere 3% of a 400 mm^2 chip in a 32 nm process technology, and not a critical design factor.

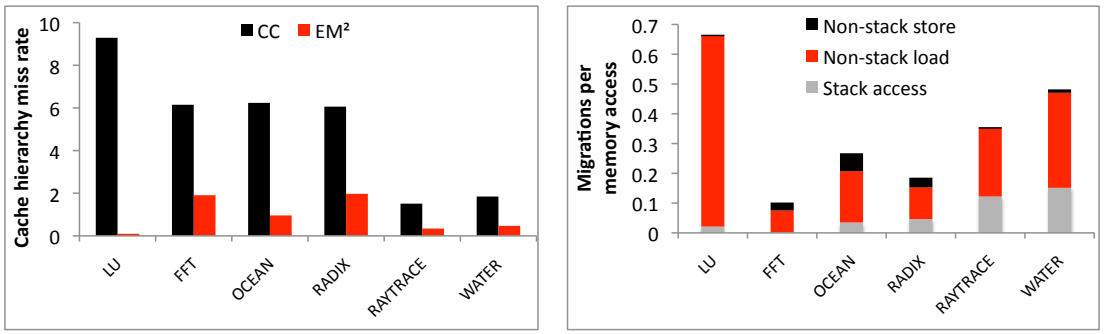
VI. Evaluation

A. Cache miss rates

The main premise of the EM^2 architecture is to improve on-chip cache utilization by not replicating writable data, which in turn, limits the number of cycles spent waiting for off-chip memory access. Because the improvement arises from not storing addresses in many locations, cache miss rates naturally depend on the memory access pattern of specific applications (Figure 10a), and the size of the L2 cache (the directories, the size of which is another significant factor in CC, are absent in EM^2). For example, the FFT benchmark does not exhibit much sharing and the high cache miss rate of 6% for CC is due mainly to significant directory evictions. EM^2 lowers the cache miss rate to under 2% as the caches are only subject to capacity and compulsory misses; indeed, to match EM^2 performance, FFT requires 4× the size of L2 cache under CC.

The LU benchmark does not incur any directory evictions but exhibits significant read-write sharing, which, in CC, causes mass evictions of cache lines actively used by the application; at the same time, replication of the same data in many core caches adds to the cache capacity requirements. A combination of capacity and coherence misses results in a 9% miss rate for CC, while EM^2 eliminates all such misses and only incurs compulsory misses at a rate of 0.1%.

The remaining benchmarks see a combination of directory evictions and read-write sharing patterns. In RADIX and OCEAN, the cache miss rate for CC is significantly higher than RAYTRACE and WATER because a higher proportion of L2 misses results in invalidation or flush requests. Overall, CC requires in excess of 4× the L2 cache to match the cache miss rate of the EM^2 architecture.



(a) Cache miss rates under EM^2 and CC

(b) Core miss rates for EM^2

Fig. 10. On average, the cache miss rate under EM^2 is 5× lower because storing each cache line in only one location eliminates many capacity and coherence-related evictions and effectively increases the availability of cache lines. Performance of EM^2 is dependent on lowering cache misses as well as core misses.

B. Core miss rates

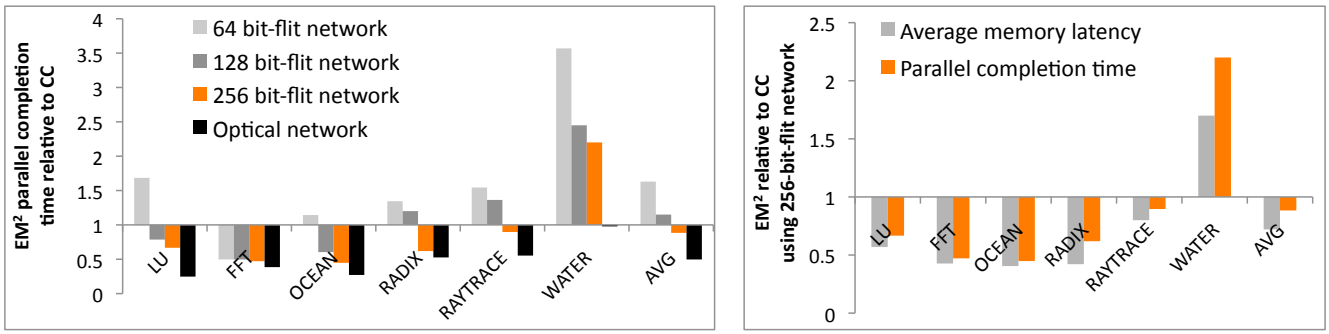
Unlike traditional hardware cache coherence mechanisms, EM^2 must frequently move the execution context of threads from core to core, which could potentially limit performance. Figure 10b shows migration frequencies and the access types causing migrations across a range of benchmarks. In RAYTRACE and WATER, nearly 70% of memory accesses are to the stack, which means that any non-stack access is most likely followed by a migration back to the core that holds the stack, and the overall migration rate is 30%. While our models use an x86 architecture, a more register-rich instruction-set architecture designed to reduce stack memory accesses can improve EM^2 core miss rates.

Our simulations indicated that more than 60% of migrations are due to shared memory reads. This suggests that much of the shared data is read-only for significant portions of the execution, and that a more optimized data placement scheme that, for example, detects application phases where some address ranges are read-only and replicates the relevant data explicitly in memory would lower core miss rates (cf. Section VIII for future extensions to our EM^2 data placement scheme).

Depending on the application memory access patterns, migrations in an EM^2 architecture may occur frequently, and so the interconnect network is a much more important design point for EM^2 than for CC. In particular, applications that have a lot of read-only sharing (e.g., WATER) and fit in per-core caches do not cause much coherence traffic under CC, but may be subject to many migrations under EM^2.

C. Overall performance

Figure 11a shows the parallel completion time speedup of EM^2 over CC for all benchmarks. Four interconnect design points are shown: an electrical network with 64, 128, and 256-bit flit size, and an optical network with link bandwidth equivalent to a 256-bit-per-flit electrical network. Overall, CC outperforms EM^2 for the 64-bit and 128-bit flit electrical network because EM^2 must load the large 1.5K-bits execution context packets onto the network; an electrical network with 2×–4× more bandwidth, however, allows EM^2 to match or outperform CC by up to 2.5×. The WATER benchmark is the only exception—CC performs better for all electrical network configurations—because it combines a low cache miss rate under CC and a high core miss rate under EM^2. We reasoned that migration rates can be lowered by limited in-memory replication of read-only data; indeed, our



(a) Parallel completion time comparisons with equivalent network

(b) Performance comparisons for high-bandwidth electrical network

Fig. 11. EM^2 performance relative to CC scales with interconnection network. On average, a high-bandwidth network with 256-bit flits improves EM^2 average memory latency by 1.4× and parallel completion time by 1.13×. The drop in performance for parallel completion time relative to average memory latency is attributed primarily to the serialization of threads in EM^2 (see Section V-A).

initial experiments with the WATER benchmark suggest a core miss rate reduction by more than 50%, bringing the performance of EM^2 up to par with CC. (cf. Section VIII).

To a first approximation, parallel completion time is determined by the average memory latency: as shown in Figure 11b, they follow the same trends. Advantages in parallel completion time, however, can be somewhat lower for EM^2 because locations concurrently accessed by many threads (e.g., barriers) can cause a serialization effect where several threads compete for execution on one single-issue core (see Section III and V-A for more details): because of this, the 1.4× improvement in overall memory latency for EM^2 translated to a 1.13× improvement in parallel completion time. The magnitude of this effect depends on the exact data sharing patterns in each application: for example, in RADIX and WATER, which show the highest levels of data sharing, the parallel completion time advantage is respectively 47% and 29% smaller; on the other hand, FFT exhibits little sharing and EM^2 suffers a smaller 10% drop in performance due to serialization. In applications with relatively unstructured access to data, expected to typify future multicore workloads [23], this serialization would have comparatively little effect: indeed, this is exactly what happened with our synthetic random-access benchmark (Section III-C).

While cache coherence performance depends directly on the on-chip cache performance, EM^2 relies primarily on the interconnection network. Optical interconnect technology is not a far-fetched reality, and enables low-latency, high-bandwidth, and low-power communication on-chip [7], [11]. Our results using an on-chip optical interconnect show an average 2× advantage of EM^2 over CC—an optical network allows EM^2 to migrate contexts efficiently and reduce off-chip memory access rates by utilizing the on-chip resources more efficiently.

D. Dynamic energy vs. performance

On the one hand, EM^2 incurs significant increase in dynamic energy consumption due to increased traffic on the high bandwidth on-chip network, and the additional register file per core to pipeline migrations. On the other hand, dramatic reductions in off-chip accesses equates to very significant reduction in energy to access DRAM. Table V shows the breakdown of dynamic energy for our SPLASH-2 benchmarks using a 256-bit-flit router for EM^2 and a 64-bit-flit router for CC. Results show that many benchmarks show dramatic reduction (up to 2.5×) in energy consumption for EM^2 relative to CC. For example, the LU benchmark with the highest migration rate of 65% spends 88% (4.79μJ) of its dynamic energy on orchestrating the

Component	LU		FFT		OCEAN		RADIX		RAYTRACE		WATER	
	CC	EM^2	CC	EM^2	CC	EM^2	CC	EM^2	CC	EM^2	CC	EM^2
Extra register file	—	2.23	—	0.44	—	0.94	—	1.01	—	0.62	—	1.86
L1 cache	0.57	0.55	0.95	0.93	0.78	0.76	1.21	1.19	0.38	0.38	0.84	0.83
L2 cache	0.27	0.03	0.27	0.10	0.25	0.06	0.35	0.14	0.05	0.02	0.08	0.03
Router	0.11	2.56	0.12	0.62	0.11	1.12	0.17	1.33	0.01	0.72	0.03	2.15
Directory cache	0.43	—	0.31	—	0.34	—	0.41	—	0.04	—	0.11	—
DRAM	9.00	0.10	11.99	3.54	9.25	1.51	18.43	5.94	1.02	0.26	3.14	0.91
Total Energy (μJ)	10.38	5.46	13.64	5.64	10.73	4.39	20.57	9.62	1.49	2.00	4.20	5.78
EM^2 energy vs. CC	**0.50**		**0.4**		**0.4**		**0.5**		**1.34**		**1.38**	
EM^2 EDP vs. CC	**0.35**		**0.2**		**0.18**		**0.29**		**1.2**		**3**	

TABLE V
ENERGY CONSUMPTION BREAKDOWN FOR EM^2 AND CC

execution contexts between cores. CC incurs very little on-chip communication, and spends 87% (9μJ) of its energy on accessing off-chip DRAM. Since the cost of DRAM access is much more expensive than on-chip high-bandwidth routers, the LU benchmark has a $2\times$ energy reduction for EM^2. At the other extreme, the WATER benchmark combines low cache miss rate under CC and a high core miss under EM^2, resulting in a net loss of $1.32\times$ in energy consumption for EM^2. Lowering the core miss rates for EM^2 will not only result in better performance, but also energy consumption (cf. Section VIII). Overall, our results indicate that EM^2 improves dynamic energy consumption by $1.3\times$ on average when compared to an MSI based cache coherent design. In addition to an average $1.13\times$ performance gain, EM^2 can deliver up to $5.4\times$ energy-delay product (EDP) gain.

The above results are for a network with 1 cycle per hop. While EM^2 performance is slightly worse than CC under a network with 3 cycles per hop (cf. Figure 9b), energy consumption grows much more slowly with the number of pipeline stages because pipelining the router requires only adding pipeline registers: higher network traffic under EM^2 makes its energy advantage decrease over CC as the number of pipeline stages grows, but EM^2 still uses 24% less energy in a 3-cycle design and 9.3% less in a 10-cycle design.

VII. 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 [24]. 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 [25]. Computation spreading [26] 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 [27]. This work attempts to find an optimal data placement for cache lines. A compile-time program transformation based migration scheme is proposed in [28] 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. Dataflow machines (e.g., [29])—and, to some extent, out-of-order execution—are superficially similar as they allow an activated instruction to be claimed by any available execution unit, but cannot serve as a shared-memory abstraction. The J-machine [30] ties processors to on-chip memories, but relies on user-level messaging and does not address the challenge of off-chip memory bandwidth. Our proposed execution migration machine is unique among the previous 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. Bringing data to the locus of computation has been explored in great depth by 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 [31] 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], [31]. The CoG [32] page coloring scheme moves pages to the "center of gravity" to improve data placement. The O^2 scheduler [33], 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 [34], [35]. 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 [36] 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 [37] propose migrating the execution of critical sections to a powerful core for performance improvement. Core Salvaging [38] exploits inter-core redundancy to provide fault tolerance via execution migration. Thread motion [12] exchanges running threads to provide fine-grain power management.

VIII. Optimizations and Future Work

Memory access latency can be optimized in various ways in both cache coherence and EM^2. On the one hand, more and more protocol states offer the potential for complex cache coherence designs (e.g., MOSI, MESI, MOESI, etc.) to keep data on-chip and lower off-chip memory access rates (cf. Section III-B). On the other hand, EM^2 can be extended with limited data replication and coherence support. Because of the complex interactions among the many states, we have focused on the MSI version of cache coherence and a pure EM^2 design, relegating the comparison against other variants of these protocols to future research.

At the same time, applications themselves can be optimized for either a specific cache-coherent design or EM^2. For example, rewriting the SPLASH-2 LU benchmark to take specific advantage of cache coherence lowers the cache miss rates over ten-fold [39], while rewriting LU specifically for EM^2 dropped the core miss rates from 66% to 1.7%: these address the critical areas of cache coherence and EM^2, respectively, and significantly improve application performance. A thorough investigation of application-level optimization remains an area of future research for us.

REFERENCES

[1] S. Rusu, S. Tam, H. Muljono, D. Ayers *et al.*, "A 45nm 8-core enterprise Xeon® processor," in *A-SSCC*, 2009, pp. 9–12.
[2] S. Bell, B. Edwards, J. Amann, R. Conlin *et al.*, "TILE64 - processor: A 64-Core SoC with mesh interconnect," in *ISSCC*, 2008, pp. 88–598.
[3] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe *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] S. Borkar, "Thousand core chips: a technology perspective," in *DAC*, 2007, pp. 746–749.
[5] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *ISCA*, 2009, pp. 184–195.
[6] "Assembly and packaging," *International Technology Roadmap for Semiconductors*, 2007.
[7] G. Kurian, J. E. Miller, J. Psota, J. Eastep *et al.*, "Atac: A 1000-core cache-coherent processor with on-chip optical network," in *PACT*, 2010.
[8] N. Kirman and J. Martinez, "A power-efficient all-optical on-chip interconnect using wavelength-based oblivious routing," in *ASPLOS*, 2010.
[9] M. Cianchetti, J. Kerekes, and D. Albonesi, "Phastlane: A rapid transit optical routing network," in *ISCA*, 2009.
[10] B. Welch, "Silicon photonics: Optical connectivity at 25 gbps and beyond," in *Hot Chips*, 2010.
[11] J. Xue, A. Garg, B. Ciftcioglu, S. Wang *et al.*, "An intra-chip free-space optical interconnect," in *ISCA*, 2010.
[12] K. K. Rangan, G. Wei, and D. Brooks, "Thread motion: fine-grained power management for multi-core systems," in *ISCA*, 2009, pp. 302–313.
[13] M. M. Bach, M. Charney, R. Cohn, E. Demikhovsky *et al.*, "Analyzing parallel programs with pin," *Computer*, vol. 43, pp. 34–41, 2010.
[14] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald *et al.*, "Graphite: A distributed parallel simulator for multicores," in *HPCA*, 2010, pp. 1–12.
[15] S. Woo, M. Ohara, E. Torrie, J. Singh *et al.*, "The SPLASH-2 programs: characterization and methodological considerations," in *ISCA*, 1995, pp. 24–36.
[16] W. J. Dally and B. Towles, *Principles and practices of interconnection networks*. Morgan Kaufmann, 2003.
[17] A. Kumar, P. Kundu, A. P. Singh, L. shiuan Peh *et al.*, "A 4.6tbits/s 3.6ghz single-cycle noc router with a novel switch allocator," in *in 65nm CMOS, ICCD*, 2007.
[18] T. Konstantakopulos, J. Eastep, J. Psota, and A. Agarwal, "Energy scalability of on-chip interconnection networks in multicore architectures," *MIT-CSAIL-TR-2008-066*, 2008.
[19] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman *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.
[20] www.synopsys.com, "Synopsys design compiler."
[21] R. Alverson, D. Callahan, D. Cummings, B. Koblenz *et al.*, "The tera computer system," 1990.
[22] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. Scott, "Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems," in *IPPS*, 1995.
[23] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *ISPASS*, 2009.
[24] H. Garcia-Molina, R. Lipton, and J. Valdes, "A massive memory machine," *IEEE Trans. Comput.*, vol. C-33, pp. 391–399, 1984.
[25] P. Michaud, "Exploiting the cache capacity of a single-chip multi-core processor with execution migration," in *HPCA*, 2004, pp. 186–195.
[26] K. Chakraborty, P. M. Wells, and G. S. Sohi, "Computation spreading: employing hardware migration to specialize CMP cores on-the-fly," in *ASPLOS*, 2006, pp. 283–292.
[27] M. Kandemir, F. Li, M. Irwin, and S. W. Son, "A novel migration-based NUCA design for chip multiprocessors," in *SC*, 2008, pp. 1–12.
[28] W. C. Hsieh, P. Wang, and W. E. Weihl, "Computation migration: enhancing locality for distributed-memory parallel systems," in *PPOPP*, 1993, pp. 239–248.
[29] G. M. Papadopoulos and D. E. Culler, "Monsoon: an explicit token-store architecture," in *ISCA*, 1990.
[30] M. D. Noakes, D. A. Wallach, and W. J. Dally, "The j-machine multicomputer: An architectural evaluation," in *ISCA*, 1993.
[31] S. Cho and L. Jin, "Managing distributed, shared L2 caches through OS-Level page allocation," in *MICRO*, 2006, pp. 455–468.
[32] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter, "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," in *HPCA*, 2009, pp. 250–261.
[33] S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek, "Reinventing scheduling for multicore systems," in *HotOS*, 2009.
[34] M. Chaudhuri, "PageNUCA: selected policies for page-grain locality management in large shared chip-multiprocessor caches," in *HPCA*, 2009, pp. 227–238.
[35] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi *et al.*, "Micro-pages: increasing DRAM efficiency with locality-aware data placement," *SIGARCH Comput. Archit. News*, vol. 38, no. 1, pp. 219–230, 2010.
[36] M. Zhang and K. Asanović, "Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *ISCA*, 2005, pp. 336–345.
[37] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in *ASPLOS*, 2009, pp. 253–264.
[38] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee, "Architectural core salvaging in a multi-core processor for hard-error tolerance," in *ISCA*, 2009, pp. 93–104.
[39] S. C. Woo, J. P. Singh, and J. L. Hennessy, "The performance advantages of integrating block data transfer in cache-coherent multiprocessors," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 11, pp. 219–229, 1994.