System-level Optimizations for Memory Access in the Execution Migration Machine (EM^2)

Keun Sup Shim Mieszko Lis Myong Hyon Cho Omer Khan Srinivas Devadas

Massachusetts Institute of Technology

Abstract. In this paper, we describe system-level optimizations for the Execution Migration Machine (EM^2), a novel shared-memory architecture to address the *memory wall* and *scalability* issues for large-scale multicores. In EM^2, data is never replicated and threads always migrate to the core where data is statically stored. This enables EM^2 not only to provide cache coherence without any complex protocols or expensive directories, but also to better utilize on-chip cache and thus experience much lower cache miss rate. However, it may incur significant execution migrations for shared data, which increases memory latency and network traffic, and thus, keeping migration rates low is a key under EM^2. We present systematic application optimization techniques to address this problem for EM^2 suitable for a compiler/OS implementation. Applying these optimizations manually to parallel benchmarks from the SPLASH-2 suite, we dramatically reduce the average migration rate for EM^2 by 53%, which directly improves parallel completion time by 34% on average. This allows EM^2 to perform competitively compared to a traditional cache-coherent architecture, on a conventional electrical network.

1 Introduction

The current trends in microprocessor design clearly indicate an era of multicores for the 2010s. As transistor density continues to grow exponentially, processor manufacturers are able to place a hundred cores (e.g., Tilera's Tile-Gx 100) on a chip with massive multicore chips on the horizon. Many industry pundits are predicting 1000 or more cores by the middle of this decade [5]. Will the current architectures (especially the memory sub-systems) scale to hundreds of cores, and will these systems be easy to program? Current memory architecture mechanisms do not scale to hundreds of cores because multicores are critically constrained by the *off-chip memory bandwidth wall* [5, 12]: the key constraint is the package pin density, which will not scale with transistor density [1]. Multicores to date have integrated larger caches on chip to reduce the number of off-chip memory accesses. Private caches, however, require cache coherence, and shared caches do not scale beyond a few cores [25].

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 hundreds of 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 and directory storage limits the efficient use of cache resources. S-NUCA [18] and its variants reduce off-chip memory access rates by unifying per-core caches into one large shared cache; accesses to memory cached in a remote core cross the interconnect and incur the associated round-trip latencies. The Execution Migration Machine (EM^2) [17], a general purpose shared memory architecture, instead migrates the computation's execution context to the core where the memory is (or is allowed to be) cached and continues execution there. Although moving execution context has a higher cost than moving data, EM^2 can outperform data migration architectures not only because memory accesses to a remote core require only one-way latencies instead of round-trip latencies, but also because successive memory accesses to the same remote cache—a frequent pattern under many modern applications with data locality—will result in one execution migration followed by a series of inexpensive local memory accesses.

The possible disadvantage of EM^2, however, is that since EM^2 restricts caching of each address to a single core, a large portion of data being shared within an application may cause significant migrations, which will increase both memory access latency and network load. In this paper, we extend the data alignment and replication techniques previously investigated in NUMA context (e.g., [27]) to the temporal dimension in order to improve migration rates and improve the overall performance under EM^2. Specifically:

1. We propose a limited-scope read-data replication optimization to reduce migration rates for an EM^2 architecture: when a shared address is read many times by several threads and seldom written, the proposed scheme allows temporary data copying to reduce the number of migrations. By taking advantage of the programmer's application-level knowledge, our replication can be applied to not only read-only pages but also read-write pages, and removes the process of page collapse (eliminating replicas on a write for read-write pages), which is a time-consuming requirement for page replication in NUMA architectures [27].
2. We show that applying the above mentioned optimizations to a baseline EM^2 architecture using a first-touch placement policy [21], lowers migration rates by 53% across the set of selected benchmarks. This improves the application performance, as measured by parallel completion time, by 34% on average. In contrast, our replication optimizations provide no benefits for cache-coherent systems because shared data are blindly replicated under cache coherence protocols.

2 The EM^2 architecture

Traditional hardware cache coherence multicore architectures bring *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 assigned to 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 divided 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 can, for example, be done by the OS on a first-touch basis, and is independent of the number of memory controllers). Since an address can be accessed in at most one location, ensuring properties that are difficult in traditional cache-coherent systems—such as sequential consistency and cache coherence—becomes simple. Under the same constraints of assigning each address to a unique core and not allowing local caching of remote data, moving the computation to data instead of bringing data to the computation has benefits because: (a) the execution migration is a one-way protocol whereas retrieving data requires round-trip latencies, and (b) for applications with data locality, successive memory accesses to the same remote cache will turn into *local* accesses under EM^2, whereas they would be repeated remote accesses under a remote-access design.

The cost of memory access within the EM^2 architecture is driven by the cost of memory accesses to the cache or DRAM, and the cost of migrations due to a *core miss*. A core miss is determined by computing the *home* core for a memory address. If the core that originated the memory access is the home, it is a *core hit*, otherwise, a *core miss*. The core miss cost incurred by the EM^2 architecture is dominated by transferring an execution context to the home core for any 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 to about 1.5 Kbits including the TLB [24]. Although on-chip electrical networks today are not generally designed to carry that much data in parallel, on-chip communication scales well; further, the network can be optimized because all transfers have the same size and migrations are independent.

The per-memory-access cost can be expressed in terms of core hit and miss rates as

$$cost_{access} = rate_{core\_hit} \times cost_{memory} + rate_{core\_miss} \times (cost_{migration} + cost_{memory})$$
$$\text{where} \quad cost_{memory} = rate_{cache\_hit} \times cost_{cache} + rate_{cache\_miss} \times cost_{dram}.$$

While $cost_{dram}$ is relatively constrained, we can optimize performance by improving the other variables. Assignment of addresses to the cores determines the performance of an EM^2 design by influencing: (a) off-chip memory accesses required, and (b) pauses in execution due to migrations. 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.

Prior work [17] shows that EM^2 improves $rate_{cache\_hit}$ when compared to a directory based cache-coherent configuration. However, reducing $cost_{migration}$ may require a high-bandwidth network, adding area as well as power to an already power constrained package. An alternative is to reduce $rate_{core\_miss}$, and that is the focus of this paper.

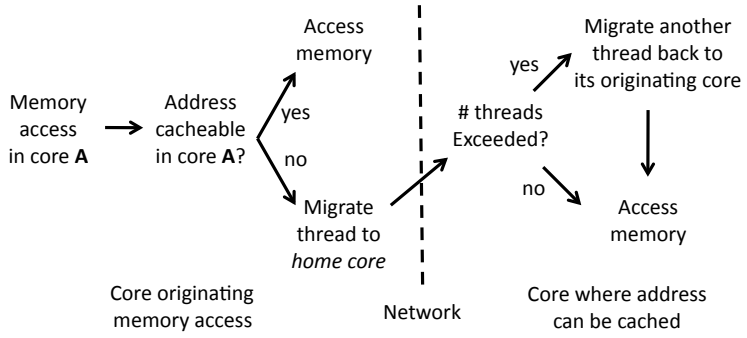


Fig. 1. Memory accesses to addresses not assigned to the local core cause the execution context to be migrated to the core.

2.1 Data placement

Because under EM^2 each physical address resides in only one core and any attempt to access it will result in a migration to that core, the mapping of virtual addresses to physical addresses directly affects migration rates and cache utilization, and, consequently, memory access performance. The OS performs the mapping using the existing virtual memory mechanism: when a virtual address is first accessed and thus should be mapped to a physical page, it chooses where the relevant page should reside by mapping the virtual page to a physical address range assigned to a specific core.

In this paper, we use the ORIGINAL scheme, a variant of first-touch [21], where pages are mapped to the accessing thread's *originating* core on the first access, and remain there for the entire duration of the execution. The ORIGINAL scheme performs well because it aims to keep each thread on its originating core for as much of its running time as possible by taking advantage of data access locality, effectively reducing the migration rate while keeping the threads spread among cores.

2.2 Migration Framework

Figure 1 shows a slight variant of the migration framework of [17]. 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, the thread running on the core B is evicted and migrated back to its originating core.

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. For this reason, we allow each core to hold multiple execution contexts, and resorting to evictions only when the number of hardware contexts running at the target core would exceed available resources. Results from [17] show that a 2-way multithreaded core microarchitecture (similar to [2]) provides sufficient performance by hiding the serialization effects of multiple threads contending for a core.

3 System-level Optimizations for EM^2

For non-trivially parallel applications, application optimization for a specific memory architecture is paramount in achieving the fastest possible performance, since it results in dramatic improvements in memory access latencies, a critical determinant of overall application performance. Although any shared-memory application can run on EM^2 without any modifications, applications resulting in significant migrations may suffer in performance since they will both increase the memory access latency and the network traffic.

In this section, we present OS- and application-level optimization techniques that significantly improve application performance by dramatically reducing migration rates for EM^2. We then show how these techniques apply in real application code by analyzing example benchmarks from the SPLASH-2 suite.

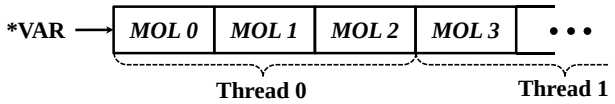
3.1 Optimization techniques

Per-thread heap memory allocation In most implementations, `malloc()` uses a shared heap to allocate memory to any requesting threads without regard to page boundaries: consecutive segments are assigned to different threads in the order in which the `malloc()` calls were invoked. Under EM^2 this can result in a kind of false sharing: private data used by separate threads are likely to end up on the same physical page and the threads will unnecessarily contend for that core.

When optimizing applications for EM^2, our goal is then to ensure that all thread-local data allocated using `malloc()` can be mapped to the thread that allocated them. With the ORIGINAL data placement scheme (described in Section 2.1), the address-to-core mapping occurs at a page granularity, and we can guarantee correct thread mapping by ensuring that `malloc()` calls in separate threads allocate memory from separate pages.

Operating system and library support for this optimization has two components: (a) ensure that `malloc()` and friends allocate data for separate threads in different pages, and (b) optionally allow the programmer to specify the thread to which the memory should belong. The first part is entirely transparent to the programmer, and consists of replacing the central dynamic memory management structure (say a free list) by a set of equivalent per-thread structures, and allocating data for each thread from its own pool. The second component exposes additional system details to the programmer, but works well in the common case where memory is first allocated in one thread and later different, disjoint regions are used in other threads (possibly spawned after memory has been allocated and initialized). This requires modifying the memory allocation (e.g., `malloc()`) and thread spawning (e.g., `pthread_create()`) library functions to take an additional parameter to identify the core where the memory should be mapped: for `malloc()`, this applies to the allocated memory, while in `pthread_create()` it applies to the newly created thread stack.

Restructuring data for private sharing In addition to overlapping sections of heap-allocated memory, data structures allocated contiguously by the programmer contain swathes of data private to different threads; for example, the WATER benchmark allocates an array of molecules processed separately by different threads:



—in this case, unless the molecule boundaries coincide with EM^2 page boundaries, false sharing will occur.

To improve EM^2 performance, the relevant data structure must be restructured (indeed, this is the same technique used to eradicate cache-line-level false sharing in the LU_CONTIGUOUS version of the LU benchmark). In most cases, this kind of transformation can only be done by the programmer, as the typical compiler would not, in general, be able to determine that different sections of the data structure are accessed by separate threads.

Read sharing and limited replication Some shared application data are written only once (or very few times) and read many times in multiple threads. In a cache-coherent architecture, this data will be replicated automatically in all user caches by the coherence protocol; under EM^2, however, each data element will stay in the core it was mapped to, and threads not running on that core will have to migrate there for access.

For example, several matrix transformation algorithms contain at their heart the pattern reflected by the following pseudocode:

```
barrier();
for (...) {
  ...     D1 = D2 + D3;     ...
}
barrier();
```

where `D1` "belongs" to the running thread but `D2` and `D3` are owned by other threads and stored on other cores; this induces a pattern where the thread must migrate to load `D2`, then migrate to load `D3`, and then again to write the sum to `D1`.

This observation suggests an optimization strategy for EM^2: during time periods when shared data is *read* many times by several threads and *not written*, make temporary local copies of the data and compute using the local copies:

```
barrier();
// copy D2 and D3 to local L2, L3
for (...) {
  ...     D1 = L2 + L3;    ...
}
barrier();
```

While a cache coherence protocol will do this blindly to all data regardless of how often it is read or written (and thus suffers high write-driven invalidation rates in benchmarks like RADIX), in EM^2 the programmer applies this technique judiciously using our profiling tool. The PIN-based profiler keeps track of the number of execution migrations for each code line, which tells the programmer which data are causing most migrations, and thus, better to be replicated. Since these local copies are guaranteed to be only read

within the barriers by the programmer, there is no need to invalidate replicated data under our replication optimization.

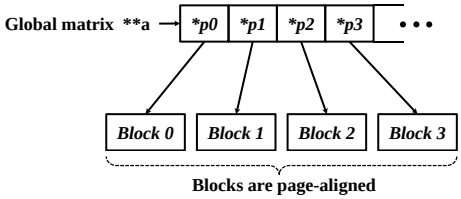
In our proof-of-concept SPLASH-2 benchmark refinements we applied this optimization by hand, and the copy process incurred many back-and-forth migrations. The number of these migrations can be significantly reduced by adding an architecture-level memory copy operation. Unlike string instructions present in some architectures (e.g., `movsb` and friends on x86) which are executed by the CPU, however, this operation would occur at the memory controller and would not involve any network traffic beyond the request itself and completion acknowledgement.

Architecturally, such an instruction would result in a message to the relevant DRAM controller requesting the transfer, and an acknowledgement-wait stall state if an instruction attempted to access the fresh copy of the data. If both memory ranges resided in the same memory controller, the copy would be internal to the controller and involve no traffic; if, on the other hand, the copied address ranges were mapped to two separate memory controllers, an efficient network-level block transfer would be used directly between the controllers. Finally, the memory controller would signal the requesting CPU core that the transfer has completed and accesses to the target memory range may proceed. In either case, the resulting network traffic would be significantly less than the many migrations required by "vanilla" EM^2 to complete the copy.

Because this architectural extension is not required for EM^2 functionality, however, the results we present here do *not* assume such an operation, and any data copy operations incur migrations as the copying threads bounce between the two relevant cores.

Specific benchmarks With these optimizations, we modified a set of SPLASH-2 benchmarks (FFT, LU, OCEAN, RADIX, RAYTRACE, and WATER) in order to reduce migration rate under the EM^2 architecture. Although we only describe our modifications for LU and WATER here, we have applied the same techniques for the rest of the benchmarks.

LU : In the original version optimized for cache coherence (LU_CONTIGUOUS), which we used as a starting point for optimization, the matrix to be operated on is divided into multiple blocks in such a way that all data points in a given block—which are operated on by the same thread—are allocated contiguously. Each block is also already page-aligned, as shown below:



Therefore no data restructuring is required to reduce false sharing.

During each computation phase, however, each thread repeatedly reads blocks owned by other threads, but writes only its own thread; e.g., in the LU source code snippet

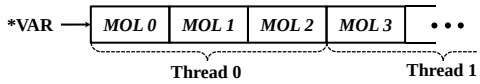
```
for (k=0; k<dimk; k++) {
  for (j=0; j<dimj; j++) {
    alpha = -b[k+j*strideb];
    for (i=0; i<dimi; i++)
```

```
        c[i+j*stridec] += alpha*a[i+k*stridea];
    }
  }
```

since the other threads' blocks (`a` and `b`) are mapped to different cores than the current thread's own block (`c`), nearly every access triggers a migration.

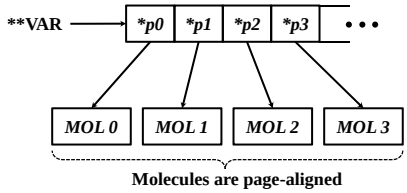
Since blocks `a` and `b` are read-only data within this function and the contents are not updated by other threads in the scope, we can apply the method of limited local replication as described in Section 3.1. In the modified version, a thread copies the necessary blocks—`a` and `b` in the example above—to local variables (which are also page-aligned to avoid false-sharing); the computation then only accesses local copies, eliminating migrations once the replication is done. We similarly replicate global read-only data such as the number of threads, matrix size, and the number of blocks per thread.

WATER : In the original code, the main data structure (`VAR`) is a 1D array of molecules to be simulated, and each thread is assigned a portion of this array to work on:



The problem with this data structure is that, as all molecules are allocated contiguously, molecules processed by different threads can share the same page and this false sharing can induce unnecessary migrations.

To address this, we modify the `VAR` data structure as follows:



By recasting `VAR` as an array of pointers, we can page-align all of the molecules, entirely eliminating false-sharing among them; this guarantees that, under EM^2, a thread never needs to migrate to access a molecule assigned to it.

In addition, WATER can also be optimized by locally replicating read-only data. For each molecule, the thread computes some intermolecular distances to other molecules, which requires read accesses to the molecules owned by other threads:

```
CSHIFT() {
  XL[0] = XMA-XMB;     XL[1] = XMA-XB[0];   XL[2] = XMA-XB[2];
  XL[3] = XA[0]-XMB;   XL[4] = XA[2]-XMB;   XL[5] = XA[0]-XB[0];
  XL[6] = XA[0]-XB[2]; XL[7] = XA[2]-XB[0];      ...
}
```

Here, `XMB` and `XB` are parts of molecules owned by other threads, while `XMA`, `XA`, and `XL` belong to the thread that calls this function. Since all threads are synchronized before and after this step, and the other threads' molecules are not updated, we can safely make

a read-only copy in the local memory of the caller thread. Thus, after initially copying `XMB` and `XB` to thread-local data, the remainder of the computation induces no further migrations.

4 Methods

We use Pin [4] and Graphite [23] to model the EM^2 architecture. Pin enables runtime binary instrumentation of parallel programs, including the SPLASH-2 [28] 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 1. 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 EM^2 and were set to sizes recommended by Graphite on basis of the total cache capacity in the simulated system, and (b) the 2-way multithreaded cores which are not needed for cache coherent system.

Parameter	Settings
Number of cores	256, each with 2 threads, 1 issue-slot
L1/L2 data cache per core	16 KB/64 KB
Network	Mesh, 1 cycle per hop, 128 bit flits, XY Routing
Data placement scheme	ORIGINAL, VM page size 4 KB
Coherence protocol	Directory-based full-map MSI
Memory	30 GB/s bandwidth, 75 ns latency

Table 1. System configurations used

4.1 On-chip interconnect

Experiments were performed using Graphite's model of an electrical mesh network with XY routing. Each packet on the network is partitioned into fixed size flits, and we use the flit size of 128-bits for the electrical network. Since modern network-on-chip routers are pipelined [10], we argue that modeling a 1-cycle per hop router latency [20] is reasonable for the on-chip network; we account for the appropriate pipeline latencies associated with delivering a packet. In addition to the fixed per-hop latency, contention delays are modeled; the queuing delays at the router are estimated using a probabilistic model similar to the one proposed in [19].

4.2 Measurements

Our experiments used a set of SPLASH-2 benchmarks: FFT, LU, OCEAN, RADIX, RAYTRACE, and WATER. 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 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.

5 Evaluation

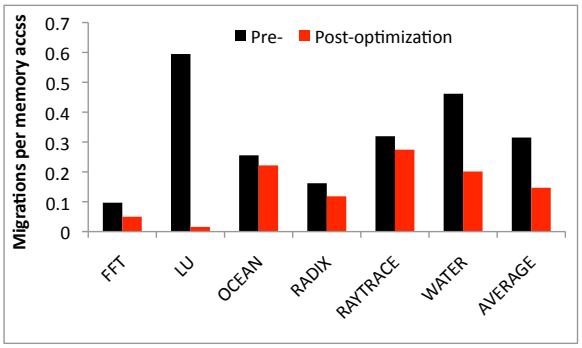


Fig. 2. EM^2 migration rates before and after the proposed optimizations. Because of better data distribution and judicious replication, migration rates drop significantly.

Figure 2 shows the effects of applying the optimizations described in Section 3. Distributing data on page boundaries to avoid false sharing, combined with judicious local replication of frequently used read-only data, combine to improve the average migration rate from 32% to 15% for the benchmarks we optimized for EM^2—a ca. 2 $\times$ improvement.

Although migration rate is not the only determinant of overall performance under EM^2, reducing the number of memory accesses that trigger migrations lowers the overall memory access time and significantly improves parallel completion times (Figure 3).

Figure 3 shows the overall parallel completion times for all benchmarks before and after our optimization. Before specifically optimizing the applications for EM^2, the EM^2 architecture was on the average outperformed by the cache coherence system; this is not very surprising, as most of these benchmarks have been specifically written with cache coherence systems in mind, and our choice of a network with 128 bit flit size. (Increasing network bandwidth beyond 128 bits benefits EM^2 much more than cache coherence [17].) After applying our optimizations, however, EM^2 on average performs competitively compared to the cache coherence system due to the significant drops in migration rates.

6 Related Work

Implicitly moving data to computation has been explored in great depth with many years of research on cache coherence protocols, and has become textbook material [13].

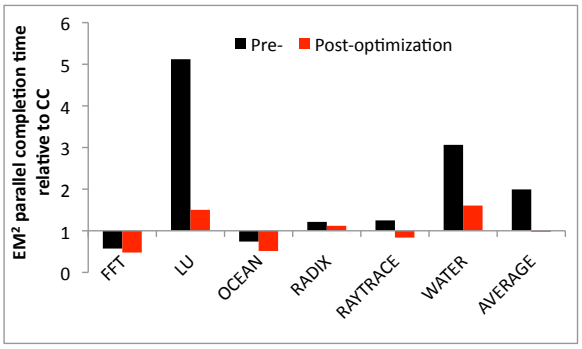


Fig. 3. In comparison to the directory-based, MSI cache-coherent system (CC), EM^2 performed $2\times$ worse on average before optimizations. After optimizations, EM^2 performs competitively on average due to the significant drops in migration rates. The CC runs and the pre-optimization EM^2 runs used the original, cache-coherence-optimized SPLASH-2 benchmarks, while the post-optimization EM^2 runs used EM^2-optimized benchmark versions. EM^2 optimizations may worsen performance under CC and hence we used the original benchmarks for all CC simulations.

Meanwhile, page replication and migration have been extensively evaluated in the context of multiprocessor NUMA architectures. Verghese *et al* [27] propose OS supported dynamic page migration and replication to alleviate the problem of large remote access latencies in CC-NUMA architectures. In these NUMA systems, both interconnect and memory latencies were high and an OS-level approach provided sufficient performance; with today's fast on-chip interconnects, however, operating system interrupts are relatively much slower, and quick, low-overhead mechanisms are needed for good performance. Moreover, our replication optimization differs from prior NUMA research in that, using our profiling tool, we choose data to replicate by the access pattern that causes significant *migrations* and not by the number of *sharers*, because our focus is to reduce migration rates under EM^2. In addition, while the page replication in CC-NUMA requires the page collapse process to eliminate replicas on a write, the optimizations we present do not require this invalidation process since the replicated data are guaranteed to be only read in a limited scope by the programmer.

More recent research has explored data distribution and migration among on-chip NUCA caches [18] with traditional and hybrid cache coherence schemes. OS-level and OS-assisted software approaches [9, 12, 3, 6] leverage the operating system to map data to caches near where threads using it are scheduled (on the same core for private addresses and geographically close for shared data) and optionally replicate read-only pages. Other schemes add hardware support for page migration support [8, 26] or replication of recently used cache lines [29]. In general, only read-only pages are shared; in contrast, the optimizations we present here take advantage of the programmer's application-level knowledge to allow replication of read-write shared data during periods when it is not being written.

The idea of computation migration was originally considered in the context of distributed multiprocessor architectures [11], and has recently re-emerged in single-chip multicores for threads [22, 16] as well as thread segments [7]; compiler transformations

for migration support have also been considered [15]. EM^2 [17] differs from these in that data sharing is completely abandoned (and therefore cache coherence protocols are not needed), and migration is required to provide memory coherence rather than employed to speed up access to cached data.

Finally, because of the complexity of coherence protocols and unscalable directory memory requirement, many recent many-core architectures (e.g., Intel's Single-chip Cloud Computer (SCC) [14]) rely on the message passing programming model instead of the shared memory model and give up on providing coherence support beyond software cache coherence. Message passing models, however, present the programmer with very low-level abstractions and, as they are relatively difficult to program, have historically been limited to specialized niche applications like scientific computing and telecommunications. In this paper, therefore, we have focused our optimization efforts on EM^2, a simple and scalable shared-memory architecture.

7 Conclusions and future work

In this manuscript, we have introduced a set of system-level optimizations to improve memory access latency for an EM^2 architecture: we chose a page-to-core mapping strategy, outlined several optimization techniques, and evaluated their effects on benchmarks from the SPLASH-2 suite. Our results show that these optimizations significantly reduce migration rates, which effectively improves the performance, enabling an EM^2 architecture to perform competitively compared to a traditional cache coherent system.

Our future research directions include automating the techniques presented here via a combination of low-overhead compiler, operating system, and/or architecture implementations; we believe a combination of the three will be critical in overcoming the limitations of previously explored migration/replication techniques. Furthermore, we will also consider how these optimizations can be applied to other application domains with different memory access patterns (e.g., streaming applications).

References

1. Assembly and packaging. *International Technology Roadmap for Semiconductors*, 2007.
2. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. smith. The Tera Computer System, 1990.
3. 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*, pages 250–261, 2009.
4. M. M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C. Luk, G. Lyons, H. Patil, and A. Tal. Analyzing parallel programs with pin. *Computer*, 43:34–41, 2010.
5. S. Borkar. Thousand core chips: a technology perspective. In *DAC*, pages 746–749, 2007.
6. S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek. Reinventing scheduling for multicore systems. In *HotOS*, 2009.
7. K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation spreading: employing hardware migration to specialize CMP cores on-the-fly. In *ASPLOS*, pages 283–292, 2006.
8. M. Chaudhuri. PageNUCA: selected policies for page-grain locality management in large shared chip-multiprocessor caches. In *HPCA*, pages 227–238, 2009.

9. S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-Level page allocation. In *MICRO*, pages 455–468, 2006.
10. W. J. Dally and B. Towles. *Principles and practices of interconnection networks*. Morgan Kaufmann, 2003.
11. H. Garcia-Molina, R. Lipton, and J. Valdes. A massive memory machine. *IEEE Trans. Comput.*, C-33:391–399, 1984.
12. N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *ISCA*, pages 184–195, 2009.
13. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, September 2006.
14. J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *ISSCC*, pages 108 –109, Feb. 2010.
15. W. C. Hsieh, P. Wang, and W. E. Weihl. Computation migration: enhancing locality for distributed-memory parallel systems. In *PPOPP*, pages 239–248, 1993.
16. M. Kandemir, F. Li, M. Irwin, and S. W. Son. A novel migration-based NUCA design for chip multiprocessors. In *SC*, pages 1–12, 2008.
17. O. Khan, M. Lis, and S. Devadas. EM^2: A Scalable Shared-Memory Multicore Architecture. *MIT-CSAIL-TR-2010-030*, 2010.
18. C. Kim, D. Burger, and S. W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *ASPLOS*, 2002.
19. 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.
20. A. Kumar, P. Kundu, A. P. Singh, L.-S. Peh, and N. K. Jha. A 4.6tbits/s 3.6ghz single-cycle noc router with a novel switch allocator. In *in 65nm CMOS, ICCD*, 2007.
21. 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.
22. P. Michaud. Exploiting the cache capacity of a single-chip multi-core processor with execution migration. In *HPCA*, pages 186–195, 2004.
23. J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA*, pages 1–12, 2010.
24. K. K. Rangan, G. Wei, and D. Brooks. Thread motion: fine-grained power management for multi-core systems. In *ISCA*, pages 302–313, 2009.
25. S. Rusu, S. Tam, H. Muljono, D. Ayers, J. Chang, R. Varada, M. Ratta, and S. Vora. A 45nm 8-core enterprise Xeon® processor. In *A-SSCC*, pages 9–12, 2009.
26. K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis. Micro-pages: increasing DRAM efficiency with locality-aware data placement. *SIGARCH Comput. Archit. News*, 38(1):219–230, 2010.
27. B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on cc-numa compute servers. *SIGPLAN Not.*, 31(9):279–289, 1996.
28. S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.
29. M. Zhang and K. Asanović. Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *ISCA*, pages 336–345, 2005.