Brief Announcement: Distributed Shared Memory based on Computation Migration

Mieszko Lis Keun Sup Shim Myong Hyon Cho Christopher W. Fletcher
Michel Kinsy Ilia Lebedev Omer Khan Srinivas Devadas

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA, USA

{mieszko,ksshim,mhcho,cwfletcher,mkinsy,ilebedev,okhan,devadas}@csail.mit.edu

1. BACKGROUND

Driven by increasingly unbalanced technology scaling and power dissipation limits, microprocessor designers have resorted to increasing the number of cores on a single chip, and pundits expect 1000-core designs to materialize in the next few years [1]. But how will memory architectures scale and how will these next-generation multicores be programmed?

One barrier to scaling current memory architectures is the *off-chip memory bandwidth wall* [1,2]: off-chip bandwidth grows with package pin density, which scales much more slowly than on-die transistor density [3]. To reduce reliance on external memories and keep data on-chip, today's multicores integrate very large shared last-level caches on chip [4]; interconnects used with such shared caches, however, do not scale beyond relatively few cores, and the power requirements and access latencies of large caches exclude their use in chips on a 1000-core scale. For massive-scale multicores, then, we are left with relatively small per-core caches.

Per-core caches on a 1000-core scale, in turn, raise the question of memory coherence. On the one hand, a shared memory abstraction is a practical necessity for general-purpose programming, and most programmers prefer a shared memory model [5]. On the other hand, ensuring coherence among private caches is an expensive proposition: bus-based and snoopy protocols don't scale beyond relatively few cores, and directory sizes needed in cache-coherence protocols must equal a significant portion of the *combined* size of the per-core caches as otherwise directory evictions will limit performance [6]. Moreover, directory-based coherence protocols are notoriously difficult to implement and verify [7].

2. EXECUTION MIGRATION MACHINE

The Execution Migration Machine (EM^2) [8,9] maintains memory coherence by allowing each address to be cached in only one core cache (the *home*), and efficiently migrating execution to the home core whenever another core wishes to access that address. A hardware-level thread migration protocol ensures that execution transfer is efficient: the architectural context (program counter, register file, and possibly other state like the TLB) is unloaded onto the interconnect network, travels to the destination core, and is loaded into the architectural state elements there [8]. Because each thread always accesses a given address from the same core, threads never disagree about the contents of memory locations so sequential consistency is trivially ensured.

The flow of a memory access under EM^2 is shown in Figure 1. Depending on the implementation, each core may be capable of multiplexing execution among several contexts at instruction granularity; when all contexts are occupied, an incoming migration causes one of them to be evicted. For deadlock-free migrations, each core has one *native* context for each of the threads that originated on that core in addition for the *guest* contexts for threads originally started on other cores: an evicted thread travels to its dedicated native context on a separate virtual network to avoid dependency loops and deadlock [10].

EM^2 can potentially outperform traditional directory-based cache coherence (CC) by avoiding the data replication and loss of effective cache capacity of CC [8, 9] and by enabling data access through a one-way migration protocol. However, migrations can negatively affect performance because of the delays involved in stopping, migrating, and restarting threads; moreover, each migration must transfer the entire execution context (1–2 KBits in a 32-bit Atom-like processor [8]) over the on-chip network, causing significant power consumption.

Optimizing performance and power, therefore, requires either (a) reducing the migration rate, or (b) reducing the amount of data transferred in each migration (and so making migrations faster and more power-efficient). Since migrations depend on the assignment of addresses to per-core caches, a good data placement method (one which keeps a thread's private data assigned to that thread's native core, and allocates shared data among the sharers) is critical. Since data placement has been investigated in the context of CC-NUMA architectures (e.g., [11]) and EM^2-specific program-level replica-

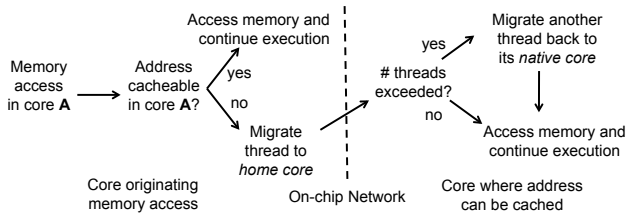

Figure 1: The life of a memory access under EM^2

SPAA'11, June 4–6, 2011, San Jose, California, USA.
ACM 978-1-4503-0743-7/11/06.

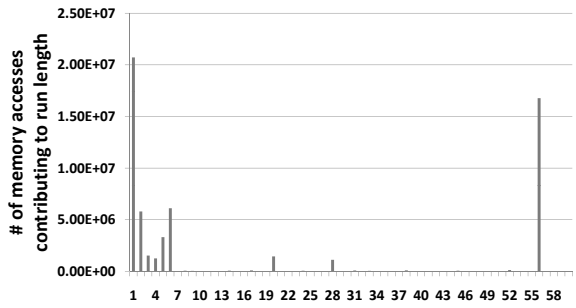

Figure 2: The number of accesses to memory cached at non-native cores for a SPLASH-2 [13] OCEAN benchmark run, binned by the number of consequent accesses to the same core (the *run length*). About half of the accesses migrate after one memory reference, while the other half keep accessing memory at the core where they have migrated. 64-core/64-thread EM^2 simulation using Graphite [14], with 16KB L1 + 64KB L2 data caches and first-touch data placement.

tion techniques have also been explored [12], the remainder of this paper focuses on part (b), and outlines two approaches to reducing the average migration cost.

3. EM² WITH REMOTE CACHE ACCESS

One scenario where EM^2 performance and power efficiency can be improved is evident from Figure 2: in about half of the non-local cache accesses (which cause the accessing thread to migrate), the thread migrates to another core after just one memory access (and possibly other non-memory instructions), usually back to the core from which the first migration originated (data not shown). In each case, a full execution context (including the entire register file) traverses the on-chip interconnect only to bring back one word of data (or, for writes, no data), clearly a suboptimal scenario.

To address this, we propose to extend EM^2 with a remote cache access capability: for some memory accesses, a thread will contact the remote cache and retrieve (or write) the necessary word instead of migrating back and forth. Although memory coherence approaches based entirely on remote cache access have been proposed [15], they must make a separate access for each word to ensure memory coherence; the combination with EM^2 is therefore uniquely poised to address both the one-off remote cache accesses and the runs of consequent accesses shown in Figure 2.

Figure 3 illustrates the memory access process under this hybrid architecture (EM^2-RA). To avoid interconnect deadlock, the remote-access virtual subnetwork must be separate from the subnetworks used for migrations and (cf. [10]), requiring six virtual channels in total. Clearly, the migration-vs.-remote-access decision is crucial to EM^2-RA performance; we therefore outline a simplified analytical model that establishes an upper bound on performance of decision schemes and thus allows us to quickly evaluate how close to optimal a given hardware-implementable scheme is.

The simplified model considers one thread at a time (and so ignores evictions caused by migrations to a core with no free guest contexts), ignores local memory access delays (since the migration-vs.-RA decision mainly affects network delays), and assumes knowledge of the full memory trace of the application as well as the address-to-core data placement. Under these assumptions, the solution can be obtained efficiently via the following dynamic program:

Given a thread memory trace $m_1, \cdots, m_N$, the data placement implies a corresponding sequence of cores $d(m_1), \cdots, d(m_N)$. Suppose we have the optimal solution to the sub-trace $m_1, \cdots, m_k$ when the thread starts at core c_0 and ends at core c_i; call this $OPT(m_1, m_k, c_i)$. The solution for the longer memory sub-trace $m_1, \cdots, m_k, m_{k+1}$, with the thread ending at a particular core c_j, can be broken into two cases:

- Core miss for m_{k+1}: $c_j \neq d(m_{k+1})$. The thread stays at c_j and performs a remote access, so we return:
  $OPT(m_1, m_k, c_j) + cost_{remote\_access}\big(c_j, d(m_{k+1})\big)$.
- Core hit for m_{k+1}: $c_j = d(m_{k+1})$. The thread either stays at c_j and accesses the *local* cache (for free) or migrates from another core and then accesses the local cache, so we return:
  $\min\big(OPT(m_1, m_k, c_j),$
  $\min_{c_i : c_i \neq c_j} OPT(m_1, m_k, c_i) + cost_{migration}(c_i, c_j)\big)$.

This optimal solution can be computed in time $O(NP^2)$, where N is the length of the trace and P is the number of processor cores. Computing the equivalent cost of a specific decision requires applying the decision procedure to each memory access in the trace, and so is $O(N)$.

4. STACK-BASED EM² ARCHITECTURE

While the EM^2-RA hybrid effectively reduces the migrated context size on average by replacing some migrations with round-trip remote cache accesses, optimal performance requires potentially complex logic in each core to make the migration vs. remote access decision.

But how can we reduce the migrated context size for *all* migrations? The minimum migration context comprises the program counter (necessary to retrieve the next instructions) and the entire register file (necessary to execute the instructions). Although one could imagine sending only a portion of the register file (based, for example, on the operands of the next few instructions), the register file is a random-access data structure and the next few instructions could refer to any subset of registers, necessitating complex muxing in and out of the register file block. Clearly, to drastically reduce migration context size we must dramatically reduce or entirely give up the register file.

Stack architectures, which do not have a random-access register file, offer a natural solution. In a stack-based ISA, most instructions do not specify their operands but instead access the top of the stack: for example, an ADD instruction would replace the top two entries on the stack with one entry containing their sum. Most often, there are two stacks (the expression stack, used for evaluation, and the return stack, used for procedure return addresses and loop counters); the top few entries of each stack are typically cached in registers and backed by a region of main memory with overflows and underflows of the stack cache automatically and transparently handled in hardware.

The stack-machine approach has been used to ensure fast procedure calls in early computers (e.g., the Burroughs B5000), simplify embedded and resilient controller architectures (see [16] for a review), and to reduce code footprint in virtual machines (e.g., the JVM). For EM^2, a stack machine dramatically reduces the required context size: because instructions can only access the top of the stack, only the top few entries must be sent over to a remote core when a memory access causes a migration. Since stack overflows and underflows are handled by loads and stores to memory,

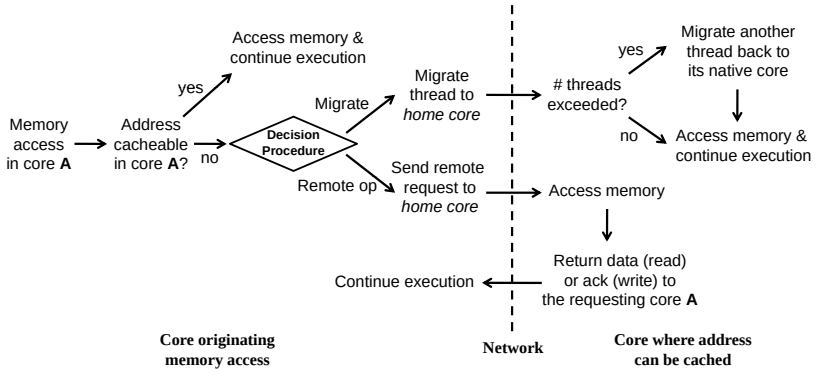

Figure 3: The life of a memory access under EM^2-RA.

the offending thread will automatically migrate back to its native core (where its stack memory is assigned) when the migrated stack overflows or underflows.

A stack-based EM^2 architecture can choose to migrate only a portion of the stack cache—with enough data to continue execution on the remote core while data accesses are being made there, and enough space to carry back any results without overflows—and flush the rest to the stack memory prior to migration. Since the migrated depth can be different for every access, determining the best per-migration depth requires a decision algorithm. Indeed, to evaluate such schemes, we can use the same analytical model described for the EM^2-RA case and a similar optimization formulation to compute the optimal stack depths (instead of the binary migrate-vs.-RA decision, the algorithm considers the various stack depths) and compares them against a given depth-decision scheme.

5. CONCLUSION

The Execution Migration Machine (EM^2) is a memory architecture that provides distributed shared memory using fast, hardware-level thread migrations. In this paper, we focus on reducing the size of execution context that is sent over the network in every migration, which improves both latency (especially on low-bandwidth interconnects) and power dissipation. To this end, we introduce two variant architectures which reduce average context size: (a) EM^2 with remote cache access that replaces some migrations with smaller round-trip remote accesses, and (b) a stack-machine EM^2 architecture where the migrated context size can vary from a few top-of-stack registers to a larger portion of the stack.

Both architectures require a fast core-local decision for every memory access: for EM^2-RA, whether to migrate or do a remote cache access, and for stack-EM^2, how much of the stack to migrate. We therefore introduce a simplified analytical model of EM^2 performance together with a dynamic-programming algorithm to compute the optimal decision sequence from an application's memory access trace, which will allow us to evaluate hardware-implementable decision schemes that will be a focus of our future research.

6. REFERENCES

[1] S. Borkar, "Thousand core chips: a technology perspective," in *DAC*, 2007.

[2] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *ISCA*, 2009.

[3] I. T. R. for Semiconductors, "Assembly and packaging," 2007.

[4] 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*, 2009.

[5] A. C. Sodan, "Message-Passing and Shared-Data Programming Models—Wish vs. Reality," in *HPCS*, 2005.

[6] A. Gupta, W. Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *International Conference on Parallel Processing*, 1990.

[7] D. Abts, S. Scott, and D. J. Lilja, "So Many States, So Little Time: Verifying Memory Coherence in the Cray X1," in *IPDPS*, 2003.

[8] O. Khan, M. Lis, and S. Devadas, "EM^2: A Scalable Shared-Memory Multicore Architecture," *MIT-CSAIL-TR-2010-030*, 2010.

[9] M. Lis, K. S. Shim, O. Khan, and S. Devadas, "Shared Memory via Execution Migration," in *ASPLOS I&P*, 2011.

[10] M. H. Cho, K. S. Shim, M. Lis, O. Khan, and S. Devadas, "Deadlock-Free Fine-Grained Thread Migration," in *NOCS*, 2011.

[11] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating system support for improving data locality on CC-NUMA compute servers," *SIGPLAN Not.*, vol. 31, no. 9, pp. 279–289, 1996.

[12] K. S. Shim, M. Lis, M. H. Cho, O. Khan, and S. Devadas, "System-level Optimizations for Memory Access in the Execution Migration Machine (EM^2)," in *CAOS*, 2011.

[13] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *ISCA*, 1995.

[14] 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*, 2010.

[15] C. Fensch and M. Cintra, "An OS-Based Alternative to Full Hardware Coherence on Tiled CMPs," in *HPCA*, 2008.

[16] P. Koopman, *Stack Computers: The New Wave*. Ellis Horwood, 1989.