Shared Memory via Execution Migration

Mieszko Lis Keun Sup Shim Omer Khan Srinivas Devadas
Massachusetts Institute of Technology, Cambridge, MA, USA

I. Background

The large-scale multiprocessor era started with supercomputers: vast numbers of single-chip microprocessors linked by an off-chip network and clusters of servers connected by even slower system-level networks. Their substantial cost and special-purpose application justified the complexity and expense of carefully designing applications using message-passing among processors with private memories: whoever could afford a supercomputer could certainly pay for the programmers needed to optimize the few applications the system was bought to run.

Today's single-die multicores, on the other hand, are within anyone's reach: GPUs already have hundreds of cores, general-purpose multicores with 64+ cores are available, and power constraints portend a massive-multicore future. Shared memory is key to programming of these systems: without this abstraction, GPGPUs, although successful in niche applications, have yet to find wide and general-purpose applicability.

Traditionally, the shared memory illusion has been preserved either by ensuring that the individual private caches hold the same values for any shared addresses via directory-based cache-coherence (CC) protocols or by dividing the address space among caches so that data are never shared and remote accesses (RA) are required to access addresses assigned to non-local caches. Neither approach, however, is perfect, and as the number of cores on a die grows into the hundreds, their problems will only worsen.

Most CC protocols blindly make a local copy of a cache line whenever it is read. Although this makes accessing shared read-only data efficient, it also reduces the effective on-chip cache capacity, since data cached in multiple caches leave less space for other data. This, in turn, leads to significantly higher cache miss rates (Figure 1); these must be handled by off-chip memory accesses or cache-to-cache transfers, which incur delays of hundreds of cycles. As core and thread counts grow, this will only worsen: not only does the impact on on-chip cache capacity increase with the number of sharers, but the much more frequent off-chip accesses will contend for the same limited off-chip bandwidth, which grows much more slowly than on-chip gate counts [2], a phenomenon known as the *off-chip memory bandwidth wall* [3], [4].

RA, a distributed shared cache design, allows each address to be cached in only one place, and therefore enjoys much lower cache miss rates (Figure 1). Rather, its Achilles' Heel is spatio-temporal data locality: in many applications, a remote access to an address cached on some core is frequently followed by many more accesses to other addresses on that core (Figure 2), each of which must incur a separate round-trip message to the remote cache to keep the memory space sequentially consistent. Although clever data placement techniques (e.g., [4], [5]) can mitigate remote access costs by placing data close to the threads that access it, this offers little improvement for data shared among many threads: indeed, the results in Figure 2 correspond to carefully hand-optimized data placement and reflect true sharing levels.

To address these shortcomings, we propose the Execution Migration Machine (EM^2), a distributed shared cache architecture which implements the single coherent memory abstraction by efficiently migrating the execution thread to a remote core whenever it needs to access data cached in that core. Unlike CC and RA, which bring *data* to the computation that needs it, EM^2 takes advantage of spatio-temporal locality by bringing *computation* to data, and constitutes a novel approach to ensuring shared memory coherence.

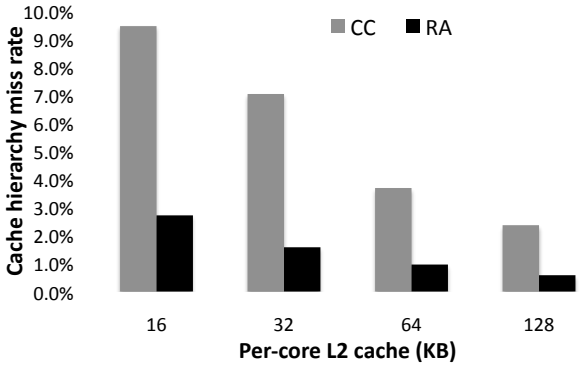

Fig. 1. Average SPLASH-2 [1] cache miss rates for various cache sizes show that duplication of shared data under CC significantly increases cache miss rates vs. RA, which does not replicate data in multiple caches. (256 x86 cores with a 16KB L1, average over the SPLASH-2 applications).

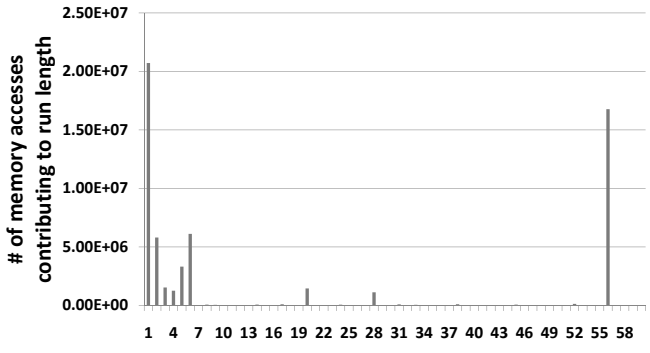

Fig. 2. The number of *non-local* memory accesses in the OCEAN benchmark binned by the number of consecutive accesses to the same remote cache (run length) highlights the weakness of RA, which pays a round-trip penalty for each remote access. (64 x86 cores with a 16KB L1 and 64KB L2 per core, hand-optimized data-to-cache assignment).

II. The EM^2 architecture

Whenever a thread wishes to read or write an address that is not assigned to the local cache (a *core miss*), the processor pipeline stops its execution and sends the entire execution context (register file, TLB, etc., a total of ca. 1.5 KBits on x86 [6]) over the interconnect network to the core where the address can be cached. In an in-order core likely to dominate massive-scale multicores, this is a simple and efficient procedure (similar to a precise exception); similarly, loading the context into the destination core requires only atomically filling the register file and TLB with the received context data.

What if the destination core is already running another thread? That thread must then be *evicted* to another core. Because the naïve solution of "swapping" the two threads is susceptible to deadlock, the deadlock-free EM^2 architecture requires that each core have space for at least two contexts: a *native* context for the thread that originated there, and a *guest* context for the threads that migrate there to access its cache; this means that there are two register files, two TLBs, etc., and the core multiplexes execution between the two on a per-instruction granularity (Figure 3).

Thus, when a core C running thread T executes a memory access for address A, it must

1) compute the *home* core H for A (e.g., by masking the appropriate bits);
2) if $H = C$ (a *core hit*),
   a) forward the request for A to the cache hierarchy (possibly resulting in an off-chip DRAM access);
3) if $H \neq C$ (a *core miss*),
   a) interrupt the execution of the thread on C,
   b) migrate the architectural state to H via the on-chip interconnect network:
      i) if H is the native core for T, place it in in the native context slot;
      ii) otherwise:
         A) if the guest slot on H contains another thread T', evict T' and migrate it to its native core N'
         B) move T into the guest slot for H;
   c) resume execution of T on H, requesting A from its cache hierarchy (and potentially accessing off-chip DRAM).

Because cached data are never duplicated and always accessed from the same core, all accesses to the same address are ordered and sequential consistency is trivially ensured.

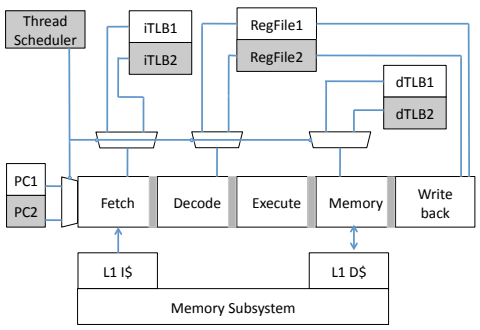

Fig. 3. An EM^2 core multiplexes execution among at least two contexts, each with its own architectural state. Additional state compared to a standard single-issue in-order core is highlighted in gray.

III. Scalability

To understand the scaling potential of the three architectures, we compare the average memory latency (AML) for a memory access under each scheme. Under CC, this is just

$$rate_{\$\_hit} \times cost_{\$\_hit} + rate_{\$\_miss} \times (cost_{protocol} + cost_{DRAM\_access})$$

As discussed in Section I, both $rate_{\$\_miss}$ and $cost_{DRAM\_access}$ will increase as the number of cores grows, and the only remedy is to bring down $rate_{\$\_miss}$ with larger per-core caches.

For RA, the AML can be expressed as

$$rate_{core\_hit} \times (rate_{\$\_hit} \times cost_{\$\_hit} + rate_{\$\_miss} \times cost_{DRAM\_access}) + rate_{core\_miss} \times (cost_{remote\_access} + rate_{\$\_hit} \times cost_{\$\_hit} + rate_{\$\_miss} \times cost_{DRAM\_access})$$

Because RA does not replicate shared data, $rate_{\$\_miss}$ under RA is lower than in CC and will not grow dramatically with core counts, which significantly scales down the expensive off-chip DRAM access component compared to CC. The disadvantage of RA is that, for highly shared addresses, consecutive remote accesses to the same core will drive up $rate_{core\_miss}$. As noted in Section I, this is unavoidable for data with high degree of sharing, and the only avenue for improvement—reducing the round-trip time of $cost_{remote\_access}$—is limited by the network topologies implementable with high core counts.

Finally, for EM^2, the AML becomes

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

Since there is no replication, $rate_{\$\_miss}$ is the same as under RA. But EM^2 scales better: $rate_{core\_miss}$ can be lowered because, after a migration, subsequent memory accesses to the same core become fast *local* accesses, and can be further optimized by reordering memory accesses. Moreover, although $cost_{migration}$ is higher than $cost_{remote\_access}$ on a low-bandwidth network because migrations must transfer many more bits, each context constitutes one packet, and the network can be trivially scaled by widening the datapaths; since migrations are one-way, $cost_{migration}$ can be made as low as $\frac{1}{2} cost_{remote\_access}$.

We argue, therefore, that execution migration scales to high core counts much better than CC and RA, and, as the core counts per die grow into the hundreds, offers a straightforward and scalable way to implement coherent shared memory.

References

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

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

[3] S. Borkar, "1000-core chips: a technology perspective," in *DAC*, 2007.

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

[5] B. Verghese, S. Devine *et al.*, "Operating system support for improving data locality on CC-NUMA compute servers," *SIGPLAN Not.*, vol. 31, pp. 279–289, 1996.

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