Design Tradeoffs for Simplicity and Efficient Verification in the Execution Migration Machine

Keun Sup Shim*, Mieszko Lis*, Myong Hyon Cho, Ilia Lebedev, Srinivas Devadas
Massachusetts Institute of Technology, Cambridge, MA, USA
{ksshim, mieszko, mhcho, ilebedev, devadas}@mit.edu

***Abstract*—As transistor technology continues to scale, the architecture community has experienced exponential growth in design complexity and significantly increasing implementation and verification costs. Moreover, Moore's law has led to a ubiquitous trend of an increasing number of cores on a single chip. Often, these large-core-count chips provide a shared memory abstraction via directories and coherence protocols, which have become notoriously error-prone and difficult to verify because of subtle data races and state space explosion. Although a very simple hardware shared memory implementation can be achieved by simply not allowing ad-hoc data replication and relying on remote accesses for remotely cached data (i.e., requiring no directories or coherence protocols), such remote-access-based directoryless architectures cannot take advantage of any data locality, and therefore suffer in both performance and energy.**

Our recently taped-out 110-core shared-memory processor, the Execution Migration Machine (EM²), establishes a new design point. On the one hand, EM² supports shared memory but does not automatically replicate data, and thus preserves the simplicity of directoryless architectures. On the other hand, it significantly improves performance and energy over remote-access-only designs by exploiting data locality at remote cores via fast hardware-level thread migration. In this paper, we describe the design choices made in the EM² chip as well as our choice of design methodology, and discuss how they combine to achieve design simplicity and verification efficiency. Even though EM² is a fairly large design—110 cores using a total of 357 million transistors—the entire chip design and implementation process (RTL, verification, physical design, tapeout) took only 18 man-months.

I. Introduction

During the past decade, continued growth in transistor densities [1] has permitted increasingly large and complex silicon designs. Because of power limitations, this increased complexity has mostly tended towards Chip Multiprocessors (CMPs): multiple cores on a single chip have become mainstream, and forward-looking pundits predict large-scale CMPs with hundreds or thousands of cores in the near future.

At the same time, chip implementation and verification costs have become more significant than ever. In addition to the increased costs of verifying the larger amount of logic in the increasingly complex cores, the sheer number of cores results in combinatorial explosion of the state space that must be examined: designers must ascertain not only the correctness of each core operating in isolation, but also of the concurrent execution of all of the cores, as well as of any protocols responsible for coordinating among them.

* Keun Sup Shim and Mieszko Lis contributed equally to this work.

One example where verification complexity has skyrocketed is hardware support for shared memory in large-scale CMPs. A full shared-memory abstraction provides programming convenience and is therefore widely accepted as a *sine qua non* for general-purpose parallel programming. In order to support shared-memory, current commodity multicore CMPs maintain coherence among private (per-core) caches using distributed directory coherence protocols. The design of even a simple coherence protocol, however, is not trivial [2]; more significantly, since the state space explodes combinatorially as the number of cores grows, many researchers have found implementing distributed directories in hardware and verifying them to be extremely difficult and not scalable [3], [4], [5], [6], [7]. Covering the entire space is impractical at even large and well-funded design houses, and verifying only small subsystems does not guarantee the correctness of the entire system [4], [6]; indeed, in modern CMPs, errors in cache coherence are one of the leading bug sources in the post-silicon debugging phase [5], and occasionally survive in chips already on the market [8].

For these reasons, various researchers have explored alternative ways to support shared memory in the realm of many-core CMPs. Several projects have proposed to transfer the burden of cache coherence from hardware to the OS and software [9], [10], or moved the coherence handling to the OS while preserving hardware support for remote cache accesses [11]. Designs like DeNovo [12] have offered simplified hardware coherence protocols at the cost of relying on more disciplined shared-memory programming models. Some general-purpose designs

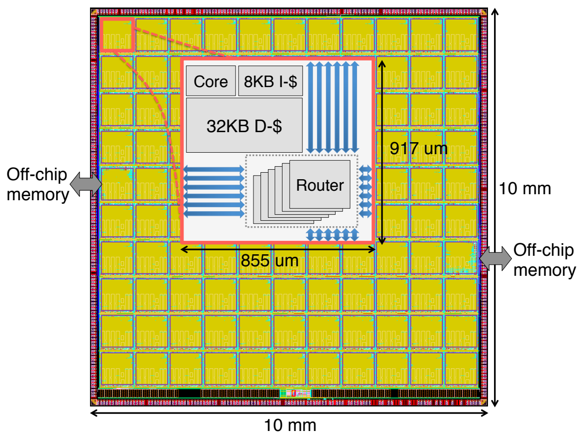


Fig. 1. The 110-core EM² chip layout. The physical chip comprises approximately 357 million transistors on a 10 mm×10 mm die in 45nm ASIC technology, using a 476-pin wirebond package. 110 tiles are placed on a 10×11 grid, connected via an on-chip network.

have done away with coherence altogether: for example, Intel's 48-core Single-Chip Cloud Computer (SCC) entirely forgoes a hardware coherence mechanism and requires the programmer to explicitly manage data coherence [13].

The 110-core Execution Migration Machine (EM²) design [14] provides a new design point on the in-hardware coherence spectrum. Built as a proof of concept for a fine-grained thread migration infrastructure [15], [16], EM² eschews traditional coherence protocols in which multiple copies of data can exist in multiple on-chip caches, and instead uses remote cache accesses to operate on remotely cached data as in [11]. Since only one copy is ever cached, coherence is trivially ensured without the need for a coherence protocol; because all decisions are local and there are no directories that must keep information about all cores in the system, the hardware is simple to implement and verification scales well with increasing core counts.

This naïve remote-access scheme, however, cannot take advantage of any data locality for remote data and therefore compromises performance and energy. To leverage this locality, EM² complements the remote cache access mechanism with fine-grained hardware-level thread migration [16]. This significantly improves both performance and energy by turning multiple remote round-trip accesses into a single migration followed by many accesses to the now local cache; because migration decisions remain local and there is no conceptually central repository holding information about all cores, the design simplicity and verification efficiency of the directoryless memory substrate are maintained.[1] Since EM² supports a shared memory model (including full sequential consistency) in *hardware*,[2] it provides more fine-grained coherence and more strict memory models than OS- and software-based approaches; at the same time, unlike conventional directory protocols, EM² has no transient states and indirections, which ensures that verification scope does not grow with the number of cores.

Although our primary goal in implementing the EM² chip was a proof-of-concept design for fast, fine-grained hardware thread migration, we secondarily strove to maintain the ease of implementation and verification that the EM² memory model provides throughout the entire design flow; our on-chip interconnect architecture, core architecture, and design/verification methodology all reflect this goal. Altogether, the entire 110-core chip design (cf. Figure 1) and implementation process (RTL, verification, physical design, tapeout) took a total of 18 man-months, proving that our implementation/verification scales efficiently despite the fairly large core count.

In the remainder of this paper, we describe the EM² memory architecture in Section II below, and the salient design choices we made to simplify design and verification in Section III. Then, in Section IV, we outline how our implementation and verification methodology helped reduce design and verification time, and offer concluding remarks in Section V.

[1]A detailed performance and energy analysis of EM², showing up to 25% improvement in performance and up to 14× reduction in network traffic over the remote-access-only design, is presented in [17]; the present paper focuses instead on how the EM² architecture enables design simplicity and verification efficiency.

[2]While software may have to be optimized to ensure optimal *performance* under EM², no program changes or restrictions are required to maintain execution *correctness*.

II. EM² Memory Architecture

EM² provides a global shared address space in hardware, where the first 86% $\left(= \frac{110}{128}\right)$ of the entire memory space of 16GB is divided into 110 non-overlapping regions as required by the EM² shared memory semantics (cf. Figure 2). Each core's data cache may only cache the address range assigned to it, and the core where the data can be cached, i.e., the *home core*, is easily computed by taking the high 7 bits of the corresponding memory address.[3] The remaining 14% of the address range is cacheable by any core but without any hardware coherence guarantees, which allows software to take advantage of patterns like read-only shared data replication. Memory is word-addressable and there is no virtual address translation; cache lines are 32 bytes. The memory subsystem consists of a single level (L1) of instruction and data caches, and a backing store implemented in external DRAM. Each tile contains an 8KB read-only instruction cache and a 32KB data cache, for a total of 4.4MB on-chip cache capacity; the caches are capable of single-cycle read hits and two-cycle write hits.

EM² implements a hybrid memory access framework which combines a remote cache access protocol (as in [11]) with a hardware-level thread migration protocol [16] for non-local memory accesses (i.e., accesses to addresses out of the requesting core's cacheable memory region). The two protocols are described below.

A. Remote Cache Access

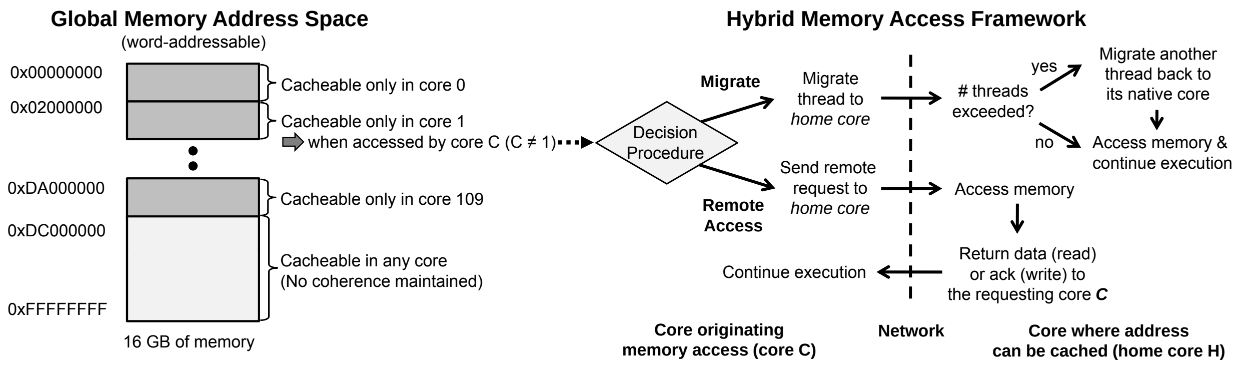
Under the remote-access protocol [18], [11], each load or store access to an address cached in a different core causes a word-granularity round-trip message to the tile allowed to cache the address: a request is transmitted over the interconnect network, the access is performed in the remote core, and the data (for loads) or acknowledgement (for writes) is sent back to the requesting core. Note that the retrieved data (word) for loads is never cached locally; the combination of word-level access and no local caching ensures correct memory semantics.

Unlike a private cache organization where a coherence protocol (e.g., directory-based protocol) takes advantage of spatial and temporal locality by making a copy of the block containing the data in the local cache, this protocol incurs round-trip costs *for every remote access*. Design complexity, on the other hand, is extremely low and scalable because every cache line can only reside in a single cache at its home core.

B. Thread Migration

EM² complements the remote access mechanism with fine-grained, hardware-level thread migration to better exploit data locality [16]. This mechanism brings the execution to the locus of the data instead of the other way around: when a thread needs access to an address cached on another core, the hardware efficiently migrates the thread's execution context to the core where the data is (or is allowed to be) cached. This can automatically turn contiguous sequences of remote cache accesses into a migration to the core where the data is cached followed by a sequence of local accesses, potentially improving performance over a remote-access-only design.

[3]Instruction caches are read-only and can cache a memory line with any address.



Global Memory Address Space
(word-addressable)
0x00000000
0x02000000
0xDA000000
0xDC000000
0xFFFFFFFF
Cacheable only in core 0
Cacheable only in core 1
when accessed by core C (C ≠ 1)
Cacheable only in core 109
Cacheable in any core
(No coherence maintained)
16 GB of memory
Hybrid Memory Access Framework
Migrate
Decision Procedure
Remote Access
Migrate thread to home core
Send remote request to home core
# threads exceeded?
yes
no
Migrate another thread back to its native core
Access memory & continue execution
Access memory
Return data (read) or ack (write) to the requesting core C
Continue execution
Core originating memory access (core C)
Network
Core where address can be cached (home core H)

Fig. 2. The global memory address space of EM² and its access mechanism. In EM², shared memory leverages a combination of remote access and thread migration; memory accesses to addresses not assigned to the local core can either result in a remote cache access or cause the execution context to be migrated to the relevant core.

If a thread is already executing at the destination core, it must be evicted and moved to a core where it can continue running. To reduce the need for evictions and amortize migration latency, cores duplicate the architectural context (register file, etc.) and allow a core to multiplex execution among two concurrent threads. To prevent deadlock, one context is marked as the *native context* and the other as the *guest context*: a core's native context may only hold the thread that started execution there (called the thread's *native core*), and evicted threads must return to their native cores to ensure deadlock freedom [15].

C. Hybrid Framework

The EM² memory access framework is the combination of remote access and thread migration: for each access to memory cached on a remote core, an instruction-granularity decision algorithm (see Section III-E) determines whether the thread should migrate or execute a remote access, as illustrated in Figure 2. The overall memory access protocol of EM² for accessing address A by thread T executing on core C is as follows:

1) compute the *home* core H for A (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 a DRAM access);
3) if $H \neq C$ (a *core miss*), and remote access is selected,
   a) send a remote access request for address A to core H,
   b) when the request arrives at H, forward it to H's cache hierarchy (possibly resulting in a DRAM access),
   c) when the cache access completes, send a response back to C,
   d) once the response arrives at C, continue execution.
4) if $H \neq C$ (a *core miss*), and thread migration is selected,
   a) interrupt the execution of the thread on C (as for a precise exception),
   b) migrate the microarchitectural state to H via the on-chip interconnect:
      i) if H is the native core for T, place it in the native context slot;
      ii) otherwise:
         A) if the guest slot on H contains another thread T', evict T' 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 DRAM).

Although the migration framework requires hardware changes to the remote-access-based design, the combined architecture of EM² is still significantly less complex than a directory-based coherence protocol. Since the absence of a conceptually central directory means that correctness arguments do not depend on the number of cores, and the combined remote-access/migration protocol does not incur the many transient states endemic in coherence protocols, EM² is far easier to reason about. In the next section, we explore how this shared-memory mechanism results in a simple implementation, and in Section IV-B we show how it enables a scalable verification flow.

III. EM² Design and its complexity

Our goals in building the EM² chip focused on (a) achieving the most efficient implementation of thread migration possible, and (b) evaluating how fine-grained thread migration can reduce on-chip traffic (and therefore dynamic power dissipation). Below, we describe how these criteria allowed us to make design choices that lead to a simpler, more easily verifiable design.

A. Homogeneous tiled architecture

As an effective evaluation of the potential of our migration architecture directed us towards as large a core count as feasible in our 10mm×10mm of silicon, we chose a tiled architecture with simple, homogeneous tiles, laid out in a 2D grid. Out of 110 tiles in the EM² ASIC, 108 are identical, while the remaining two include interfaces to off-chip memory (Figure 1).

The alternative here would have been to save area by separately optimizing tiles at each of the four edges and each of the four corners; this would have permitted us to elide on-chip network buffers as well as parts of the crossbar that faced the unused direction(s). Optimizing the edge and corner tiles, however, would have required us to verify *eleven* different kinds of tiles instead of three (one for the regular tiles, eight for the edge and corner tiles, and two for the tiles with off-chip interfaces), and significantly complicated our scalable verification strategy (described in Section IV-B below) while still not extracting sufficient area to add an extra row or column to the 2D mesh layout.

B. Memory architecture

Because our main targets were fast migration and reduction of on-chip traffic, we chose to maximize *on-chip* performance and forgo a high-bandwidth *off-chip* memory interface. Instead, EM² connects off-chip memory via simple rate-matching interfaces (described below) that extend the on-chip interconnect to the outside at two points on the sides of the chip (see Figure 1); these in turn connect to a "northbridge" FPGA that implements DDR3 memory controllers.

This choice allowed us to make several simplifications in the development process. Firstly, we avoided implementing or purchasing—and, more importantly, verifying—a DDR3 interface. Instead, verifying correct off-chip memory operation was as simple as verifying that the relevant on-chip networks extended to the outside, the packets carried correct request data, and injected response packets were properly transmitted to the on-chip interconnect. Secondly, reducing off-chip bandwidth allowed us to tape out with the simpler wirebond I/O rather than using the higher-bandwidth but more complex flip-chip option; this also meant that we could use a 476-pin package and fit in a standard socket, thus making our bring-up infrastructure simpler and more modular.

Since the EM² memory architecture eschews a coherence protocol by allowing data to be cached only in a single L1 cache based on its memory address, the L1 cache is already effectively shared and partitioned among the 110 tiles (see Figure 2). This first-level cache organization maximizes effective on-chip cache capacity given a fixed amount of on-chip SRAM, and therefore leaves no strong reason to add another higher-level of shared cache. In addition, because the L1 is already shared, leaving out a second level of cache does not significantly affect cache access patterns, since the L2 would only back up the L1 cache segment located in the same tile (in contrast to a design with a private L1 and shared L2, where each L2 slice handles requests from any private L1). Our implementation, therefore, includes only L1 instruction and data caches, and we focus our evaluation on how thread migration can effectively complement a remote cache access protocol. This choice simplified both design and verification: there were only two cache controllers to design and verify (I$ and D$), and no need to test interactions between different cache levels. Moreover, this allowed us to fit more tiles on the chip: the inflexible nature of SRAM blocks significantly complicates tile layout around them, and, since the tiles were already dominated by their large cache SRAMs, adding additional SRAM blocks would have resulted in bigger (and fewer) tiles.

The off-chip I/O interface itself (see Figure 3) uses a simple and reliable credit-based protocol with dedicated wires (rather than control messages) for control flow; the interface is content-oblivious (i.e., it directly transmits on-chip network flits). In this way, nearly all of the complexity is relegated to the off-chip "northbridge" FPGA. The ASIC generates an I/O clock

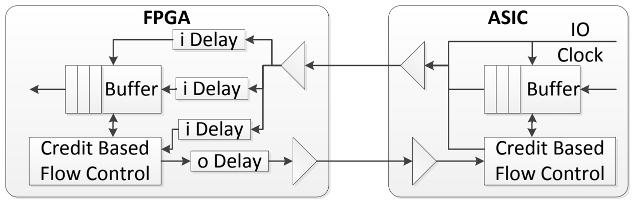


Fig. 3. Block diagram of the chip↔northbridge interface. Advanced FPGA I/O capabilities allow the ASIC side of the link to be implemented in a very simple and easy-to-verify manner.

(at a rate configurable to an integral fraction of the chip's clock speed), and all data transfer is governed by this clock. Although this means that the FPGA we interface to has to individually phase-align each pin using delay lines, this choice made it unnecessary to add clock matching and arbitrary clock crossing logic on the ASIC side, and greatly simplified I/O verification and reduced tapeout risk.

Critically, we were able to make these simplifying choices without jeopardizing our evaluation goals. This is because we specifically aimed to examine how migration can accelerate a remote-cache-access design; because programs running under both regimes access exactly the same cache lines (either via remote cache access or by thread migration followed by a local access), they induce the same cache miss patterns and therefore make the same off-chip memory requests. This observation allows us to factor out off-chip memory, implement a single-level cache, and focus on a steady-state evaluation of on-chip memory performance and interconnect traffic.

C. Stack-based core architecture

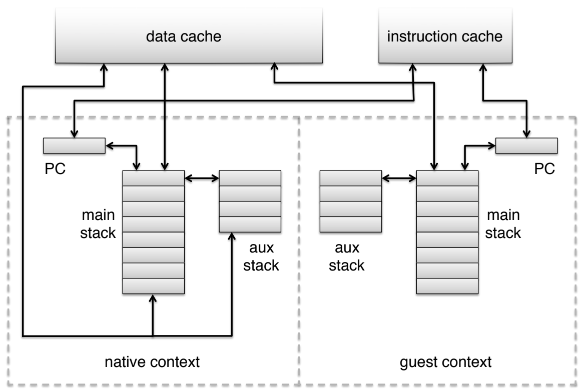


Fig. 4. The processor core consists of two contexts in an SMT configuration, each of which comprises two stacks and a program counter, while the cache ports and migration network ports (not shown) are shared between the contexts. Stacks of the native context are backed by the data cache in the event of overflow or underflow.

EM² supports hardware-level thread migration which directly moves the thread's architectural context via the on-chip interconnect; the migration cost, therefore, depends on the amount of migrated context. Since always moving the entire thread context can be wasteful if only a portion of it will be used at the destination core, our quest for the most efficient thread migration dictated that we implement *partial context*

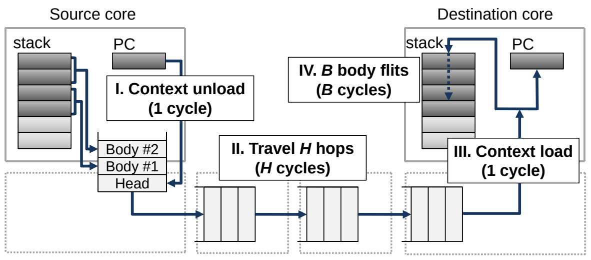

Source core
Destination core
stack
PC
I. Context unload (1 cycle)
IV. *B* body flits (*B* cycles)
stack
PC
Body #2
Body #1
Head
II. Travel *H* hops (*H* cycles)
III. Context load (1 cycle)

Fig. 5. Hardware-level thread migration via the on-chip interconnect. Only the main stack is shown for simplicity.

migration (i.e., allow the transfer of only the necessary part of the thread context).

Transferring a partial context, however, requires the hardware to predict/know which part of the context is likely to be used remotely. While implementing this in a register-to-register RISC architecture is not impossible, the random-access nature of a register file makes for complex prediction logic. Instead, we chose to implement a custom 32-bit stack architecture as shown in Figure 4; two stacks (main and auxiliary) are provided for programmer convenience. This choice greatly simplifies predicting which part of the context to migrate, since the entries near the top of the stack are far more likely to be used than entries closer to the bottom: instead of tracking precisely which registers should be migrated for each migration entry point, the prediction logic need only track how many stack entries should be taken along.

D. Deadlock avoidance in thread migration

Figure 5 illustrates how the processor cores and the on-chip network efficiently support fast instruction-granularity thread migration. When the core fetches an instruction that triggers a migration (for example, because of a memory access to data cached in a remote tile), the migration destination is computed, the thread context is serialized onto the network, and execution resumes at the destination core; the serialization and instruction fetch are pipelined, resulting in a very low migration latency that ranges from a minimum of 4 cycles to a maximum of 33 cycles (assuming no network congestion).

In a naïve implementation, a difficulty can arise if the destination core is already executing a thread. This thread can be evicted, but it is unclear where it should be sent; for example, sending it to the core that originated the migration might seem like a good idea, but this can lead to network deadlock, and at any rate another thread might already have started running there. Although a number of strategies can be employed to mitigate this kind of deadlock (for example, additional virtual channels and buffers), many require complex coordination and have special deadlock-recovery states that would complicate verification. Instead, we chose to guarantee that deadlock never arises by implementing two SMT-like thread contexts in each core: one for the thread that originated there (the *native context*) and one for any threads that originated in other cores (the *guest context*). Restricting the native context to only the thread that originated on a given core, together with requiring an evicted thread to return to its native context, ensures that deadlock never arises [15].

When multiple threads contend for the same guest core due to high data sharing within a relatively small time window, threads will evict each other according to our deadlock-free protocol. This *core contention* may result in ping-pong effect where a thread repeatedly migrates to access data only to be evicted and return to its native core, potentially degrading performance. In [17], however, we observed that performance penalty due to such evictions is not significant for our set of benchmarks, and can be kept low by allowing a guest thread to execute a set of *N* instructions before being evicted.

Although the extra context incurs an additional area cost, it significantly eases verification. In addition to the protocol itself being provably livelock- and deadlock-free [15], implementing it requires no exceptional deadlock recovery state, as both the native and guest contexts are used in normal operation and are therefore already verified.

E. Migration decision scheme

For a load/store instruction that accesses an address that cannot be cached locally, EM² has a choice either to perform remote access or to migrate a thread (see Figure 2 and Section II-C). Although our ISA allows the programmer to directly specify whether the instruction should migrate or execute via remote cache access, in general this decision can be dynamic and dependent on the phase of the program; therefore, EM² relies on an automatic hardware *migration predictor* [19] in each tile.

The main goals for the migration predictor were efficiency and simplicity—a prediction might be required in every cycle, and the state space of the hardware must be simple enough to verify exhaustively. Therefore, we implemented the predictors as a simple direct-mapped data structure indexed by the program counter (PC): a memory instruction will result in a migration only if a given PC is in the predictor table, i.e., if it is a *migratory instruction*. The predictor is based on the observation that, much like branches, sequences of consecutive memory accesses to the same home core are highly correlated with the program (instruction) flow, and the predictors learn in a fashion similar to a simple branch predictor: detecting a contiguous sequence of accesses to the same core causes the start PC to be added to the table, and migrating without executing enough local memory instructions to make the trip "worth it" causes the entry to be deleted. The simplicity of this design and its state space allowed us to verify the predictor in isolation and integrate it with the CPU cores without the need for extensive end-to-end verification.

The detection of *migratory instructions* is done by tracking how many consecutive accesses to the same remote core have been made, and if this exceeds a threshold, inserting the PC into the predictor to trigger migration. If it does not exceed the threshold, the instruction is classified as a remote-access instruction, which is the default state. Each thread tracks (1) *Home*, which maintains the home core ID for the current requested memory address, (2) *Depth*, which indicates how many times so far a thread has contiguously accessed the current home location (i.e., the *Home* field), and (3) *Start PC*, which tracks the PC of the very first instruction among memory sequences that accessed the home location in the *Home* field. We separately define the depth threshold θ, which indicates the depth at which we determine the instruction as migratory.

The detection mechanism is as follows: when a thread T executes a memory instruction for address A whose PC = P, it must first find the home core H for A; then,

1) if $Home = H$ (i.e., memory access to the same home core as that of the previous memory access),
    a) if $Depth < \theta$,
        i) increment $Depth$ by one; then if $Depth = \theta$, $StartPC$ is regarded as a migratory instruction and thus, is inserted into the migration predictor;.
2) if $Home \neq H$ (i.e., a new sequence starts with a new home core),
    a) if $Depth < \theta$,
        i) $StartPC$ is regarded as a remote-access instruction;
    b) reset the entry (i.e., $Home = H$, $PC = P$, $Depth = 1$).

While the predictor described in [19] only supports full-context migration, the migration predictor of EM² further supports stack-based partial context migration. Each predictor entry consists of a tag for the PC and the transfer sizes for the main and auxiliary stacks upon migrating a thread. When an instruction is first inserted into the predictor, the stack transfer sizes for the main and auxiliary stack are set to the default values of 8 and 0, respectively; when the thread migrates back to its native core due to stack overflow (or underflow) at the guest core, the stack transfer size of the corresponding PC is decremented (or incremented), accordingly.

Guided by the goal of simplicity and verification efficiency, we chose to implement one *per-core* migration predictor, shared between the two contexts in each core (native and guest), rather than dual per-core predictors (one for the native context and one for the guest), or *per-thread* predictors whose state is transferred as a part of the migration context. The per-thread predictor scheme was easy to reject because it would have significantly increased the migrated context size and therefore violated our goal of the most efficient thread migration mechanism. The dual predictor solution, on the other hand, could in theory improve predictions because the two threads running on a core would not "pollute" each other's predictor tables—at the cost of additional area and verification time. Instead, we chose to preserve simplicity and implement a single per-core predictor shared between the native and guest contexts, sizing the predictor tables so that our tests showed no noticeable performance degradation (32 entries).

With our partial-context migration predictor, EM² never performs worse than the remote-access only baseline for our set of benchmarks; a detailed evaluation is presented in [17]. EM² offers up to 25% reduction in completion time, and throughout our benchmark suite, EM² also offers significant reductions in on-chip network traffic, up to 14× less traffic (66% less on average).

F. On-chip interconnect

EM² has three types of on-chip traffic: migration, remote access, and off-chip memory access. Although it is possible for this traffic to share on-chip interconnect channels, this would require suitable arbiters (and possibly deadlock recovery logic), and would significantly expand the state space to be verified. To avoid this, we chose to trade off area for simplicity, and route traffic via six separate channels, which is sufficient to ensure deadlock-free operation [15].

Further, the six channels are implemented as six physically separate on-chip networks, each with its own router in every tile. While using a single network with six virtual channels would have utilized available link bandwidth more efficiently and made inter-tile routing simpler, it would have exponentially increased the crossbar size and significantly complicated the allocation logic (the number of inputs grows proportionally to the number of virtual channels and the number of outputs to the total bisection bandwidth between adjacent routers). More significantly, using six identical networks allowed us to verify in isolation the operation of a *single* network, and then safely replicate it six times to form the interconnect, significantly reducing the total verification effort.

G. Module complexity comparison

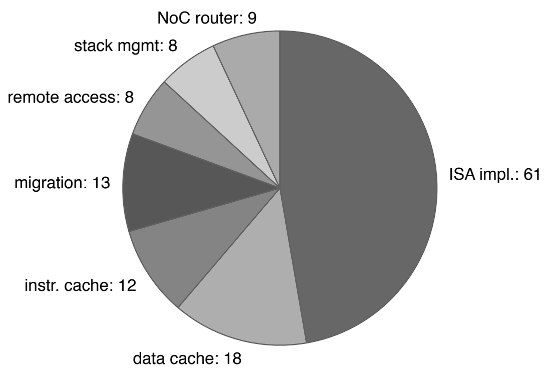


Fig. 6. Complexity comparison of the main modules contributing to the EM² tile, expressed in terms of Bluespec rule counts.

Figure 6 shows the breakdown of implementation complexity in the core modules as the number of Bluespec [20] rules used to implement each of them. A single rule roughly corresponds to a semantic operation that the hardware must perform (e.g., executing an instruction), and a rule count therefore makes a good proxy for overall complexity (see Section IV-A below). The migration infrastructure adds relatively little to the overall complexity (about 10% of the logic), and is comparable in complexity to the remote cache access mechanism or a simple cache controller. Although our chip implementation did not include a directory-based coherence protocol, the complexity would far exceed the combined complexity of the migration and remote access mechanisms in EM²: the number of equivalent rules in a basic MSI protocol would be 33 (22 state transitions for caches and 11 for directories), vs. 13 for migration and 8 for remote access.

IV. EM² Design and Verification Methodology

A. Hardware design language

Bluespec [20] is a high-level hardware design language based on synthesizable guarded atomic actions [21]. In Bluespec, a design is described using *rules*, each of which specifies the condition under which it is enabled (the guard) and the consequent state change (the action). Unlike standard Verilog `always` blocks, rules are *atomic*—in each clock cycle, a given rule will either be applied in its entirety or not at all, and the final state can be reconstructed as a sequence of rule applications. Actions in different rules may attempt to alter the same state element; when those changes conflict (e.g., two rules attempting to enqueue something in a single-port FIFO), the compiler automatically generates control logic (called a schedule) to ensure that only one of the conflicting rules may fire in a given clock cycle. When rule actions do not overlap or can be executed in parallel (e.g., one rule enqueuing into a FIFO and the other dequeuing from it), the compiler will allow the rules to fire in the same clock cycle. Source written in Bluespec is compiled to synthesizable Verilog, which is then combined with any custom Verilog/VHDL modules (such as SRAMs) and synthesized as part of the standard ASIC flow. The quality of results in terms of timing and area has been shown to be on par with hand-optimized Verilog RTL [22].

The atomic rule semantics of Bluespec encourage a coding style where each semantically distinct operation is described separately, instead of a style that focuses on describing each hardware element (as in Verilog). For example, in our stack-ISA CPU, the operations of automatically refilling and spilling the in-CPU stack into backing data memory (`fill_stack` and `spill_stack`) both access the stack registers and the data cache interface, but are described in two separate rules; the Bluespec compiler automatically creates the necessary data path muxes and control logic. Crucially, rule atomicity means that adding or removing a rule does not require any changes in existing rules: for example, the rule that completes an ALU operation (`alu_op`) also accesses the stack registers, but adding this rule requires no changes to `fill_stack` and `spill_stack`—the compiler will just infer slightly different muxes and control. This independence is handy in the design phase, and encourages a progressive-refinement approach to design "one rule at a time." Far more significantly, however, it means that implementation errors are localized to specific rules; thus, a bug-fix that changes `fill_stack` will not require changing or verifying `alu_op` even if the changes affect which stack registers are accessed and how, and the fix will work regardless of why the stack is being refilled (stack-to-stack computation, outbound migration, etc).

Less obvious but equally important, reasoning about our design in an operation-centric manner and expressing it using atomic rules allowed us to separate *functionality* and *performance*, and verify (and correct) the two aspects independently. By far most of our verification effort focused on functionality (i.e., the module being tested producing the correct output for any given input). Relying on the Bluespec compiler to automatically generate muxes and interlocks for conflicting rules, we did not have to think about the precise cycle-to-cycle operations or worry that concurrent execution of separate operations might combine to cause unexpected bugs. Once functionality was verified, we tuned the cycle-to-cycle operation to meet our performance goals, mostly by guiding the compiler with respect to rule priority and ordering and without changing any of the rules themselves. This separation of functionality and cycle-to-cycle performance also allowed us to optimize performance without fear of breaking functionality. For example, at a late stage, we discovered an unnecessary bubble in our pipeline between some pairs of instructions. In our flow, the fix was simple and localized to a faulty bypass rule; because we knew that the compiler would preserve the operational correctness as described in the rules, we needed to re-verify only the performance aspect. Had we used a design methodology where correctness and performance cannot be easily separated, we might well have judged that the risk of breaking existing functionality was not worth the benefit of improved performance, and taped out without fixing the bug.

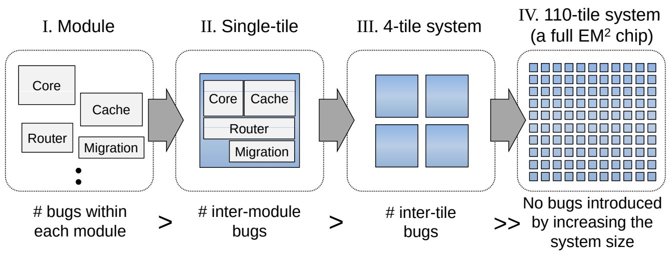

I. Module
II. Single-tile
III. 4-tile system
IV. 110-tile system (a full EM² chip)
Core
Cache
Router
Migration
# bugs within each module
>
# inter-module bugs
>
# inter-tile bugs
>>
No bugs introduced by increasing the system size

Fig. 7. Bottom-up verification methodology of EM². The high modularity and design simplicity of EM² enabled verification to scale as we integrated modules and increase the system size: the number of bugs found decreased at each step, and no new bugs were discovered by moving from a 4-tile model to the full 110-tile system.

B. System verification

With evolving VLSI technology and increasing design complexity, verification costs have become more critical than ever. Increasing core counts are only making the problem worse because any pairwise interactions among cores result in a combinatorial explosion of the state space as the number of cores grows. Distributed cache coherence protocols in particular are well known to be notoriously complex and difficult to design and verify. The response to a given request is determined by the state of *all* actors in the system (for example, when one cache requests write access to a cache line, any cache containing that line must be sent an invalidate message); moreover, the indirections involved and the nondeterminism inherent in the relative timing of events requires a coherence protocol implementation to introduce many transient states that are not explicit in the higher-level protocol. This causes the number of actual states in even relatively simple protocols (e.g., MSI, MESI) to explode combinatorially [4], and results in complex cooperating state machines driving each cache and directory [2]. In fact, one of the main sources of bugs in such protocols is reachable transient states that are missing from the protocol definition, and fixing them often requires non-trivial modifications to the high-level specification. To make things worse, many transient states make it difficult to write well-defined testbench suites: with multiple threads running in parallel on multicores, writing high-level applications that exercise all the reachable low-level transient states—or even enumerating those states—is not an easy task. Indeed, descriptions of more optimized protocols can be so complex that they take experts months to understand, and bugs can result from specification ambiguities as well as implementation errors [3]. Significant modeling simplifications must be made to make exploring the state space tractable [23], and even formally verifying a given protocol on a few cores gives no confidence that it will work on 100.

While design and verification complexity is difficult to quantify and compare, both the remote-access-only baseline and the full EM² system we implemented have a significant advantage over directory cache coherence: a given memory address may only be cached in a single place. This means that any request—remote or local—will depend only on the validity of a given line in a single cache, and no indirections or transient states are required. The VALID and DIRTY flags that together determine the state of a given cache line are local to the tile and cannot be affected by state changes in other cores. The thread migration framework does not introduce additional complications, since the data cache does not care whether a local memory request comes from a native thread or a migrated thread: the same local data cache access interface is used. The overall correctness can therefore be cleanly separated into (a) the remote access framework, (b) the thread migration framework, (c) the cache that serves the memory request, and (d) the underlying on-chip interconnect, all of which can be reasoned about separately. This modularity makes the EM² protocols easy to understand and reason about, and enabled us to safely implement and verify modules in isolation and integrate them afterwards without triggering bugs at the module or protocol levels (see verification steps I and II in Figure 7).

The homogeneous tiled architecture we chose for EM² allowed us to significantly reduce verification time by first integrating the individual tiles in a 4-tile system. This resulted in far shorter simulation times than would have been possible with the 110 cores, and allowed us to run many more test programs. At the same time, the 4-tile arrangement exercised all of the inter-tile interfaces, and we found no additional bugs when we switched to verifying the full 110-core system. Unlike directory entries in directory-based coherence designs, EM² cores never store information about more than the local core, and all of the logic required for the migration framework—the decision whether to migrate or execute a remote cache access, the calculation of the destination core, serialization and deserialization of network packets from/to the execution context, evicting a running thread if necessary, etc.—is local to the tile. As a result, it was possible to exercise the entire state space in the 4-tile system; perhaps more significantly, however, this also means that the system could be scaled to an arbitrary number of cores without incurring an additional verification burden.

C. System configuration and bootstrap

To initialize the EM² chip to a known state at during power-up, we chose to use a scan-chain mechanism. Unlike the commonly employed bootloader strategy, in which one of the cores is hard-coded with a location of a program that configures the rest of the system, successful configuration via the scan-chain approach does not rely on any cores to be operating correctly: the only points that must be verified are (a) that bits correctly advance through the scan chain, and (b) that the contents of the scan chain are correctly picked up by the relevant core configuration settings. In fact, other than a small state machine to ensure that caches are invalidated at reset, the EM² chip does not have any reset-specific logic that would have to be separately verified.

The main disadvantages here are (a) that the EM² chip is not self-initializing, i.e., that system configuration must be managed external to the chip, and (b) that configuration at the slow rate

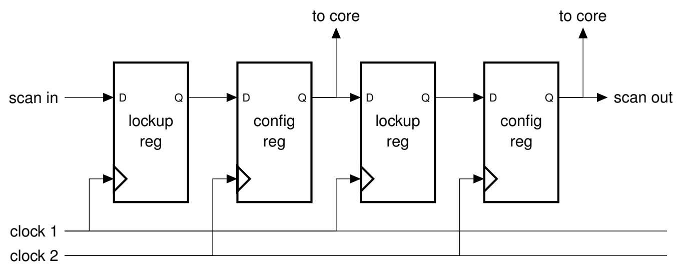


Fig. 8. The two-stage scan chain used to configure the EM² chip. The two separate scan clocks and two sets of registers prevent hold time violations due to short paths between the scan chain registers.

permitted by the scan chain will take a number of minutes. For an academic chip destined to be used exclusively in a lab environment, however, those disadvantages are relatively minor and worth offloading complexity from the chip itself onto test equipment.

The scan chain itself was designed specifically to avoid hold-time violations in the physical design phase. To this end, the chain uses two sets of registers and is driven by two clocks: the first clock copies the current value of the scan input (i.e., the previous link in the chain) into a "lockup" register, while the second moves the lockup register value to a "config" register, which can be read by the core logic (see Figure 8). By suitably interleaving the two scan clocks, we ensure that the source of any signal is the output of a flip-flop that is not being written at the same clock edge, thus avoiding hold-time issues. While this approach sacrificed some area (since the scan registers are duplicated), it removed a significant source of hold-time violations during the full-chip assembly phase of physical layout, likely saving us time and frustration.

V. Conclusion

The Execution Migration Machine (EM²) is a recently taped-out 110-core shared-memory processor, built as a proof of concept for a fine-grained thread migration infrastructure. Within the wide design spectrum of hardware shared memory support, EM² offers a new design point: it maintains the design simplicity of a directoryless shared memory while improving performance of a pure remote-cache-access design using fast hardware-level migration support.

With many-core processors quickly becoming the norm, design complexity and verification costs are receiving more attention than ever. From design to tape-out, the architectural and methodology choices we made in developing the 110-core EM² processor have been guided by the desire to keep the design easy to reason about and the implementation simple. More importantly, those choices have allowed us to make the time and effort required for verification independent of the number of cores in the system, significantly reducing total design and verification time.

References

[1] International Technology Roadmap for Semiconductors, "Assembly and Packaging," 2007.

[2] D. E. Lenoski and W.-D. Weber, *Scalable Shared-memory Multiprocessing*. Morgan Kaufmann, 1995.

[3] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. Tuttle, and Y. Yu, "Checking Cache-Coherence Protocols with TLA+," *Formal Methods in System Design*, vol. 22, pp. 125–131, 2003.

[4] Arvind, N. Dave, and M. Katelman, "Getting formal verification into design flow," in *FM2008*, 2008.

[5] A. DeOrio, A. Bauserman, and V. Bertacco, "Post-silicon verification for cache coherence," in *ICCD*, 2008.

[6] M. Zhang, A. R. Lebeck, and D. J. Sorin, "Fractal coherence: Scalably verifiable cache coherence," in *MICRO*, 2010.

[7] J. G. Beu, M. C. Rosier, and T. M. Conte, "Manager-client pairing: a framework for implementing coherence hierarchies," in *MICRO*, 2011.

[8] *Intel Core 2 Duo and Intel Core 2 Solo Processor for Intel Centrino Duo Processor Technology Specification Update*, December 2010.

[9] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, J. W. Meira, S. Dwarkadas, and M. Scott, "VM-based shared memory on low-latency, remote-memory-access networks," in *ISCA*, 1997.

[10] H. Zeffer, Z. Radović., M. Karlsson, and E. Hagersten, "TMA: A Trap-Based Memory Architecture," in *ICS*, 2006.

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

[12] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *PACT*, 2011.

[13] T. Mattson, R. Van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core SCC Processor: the Programmer's View," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, 2010.

[14] M. Lis, K. S. Shim, B. Cho, I. Lebedev, and S. Devadas, "Hardware-level thread migration in a 110-core shared-memory processor," in *HotChips*, 2013.

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

[16] M. Lis, K. S. Shim, M. H. Cho, O. Khan, and S. Devadas, "Directoryless Shared Memory Coherence using Execution Migration," in *PDCS*, 2011.

[17] M. Lis, K. S. Shim, B. Cho, I. Lebedev, and S. Devadas, "The Execution Migration Machine," in *MIT CSAIL CSG Technical Memo 511*, August 2013. [Online]. Available: http://csg.csail.mit.edu/pubs/memos/Memo-511/memo511.pdf

[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] K. S. Shim, M. Lis, O. Khan, and S. Devadas, "Thread migration prediction for distributed shared caches," *Computer Architecture Letters*, Sep 2012.

[20] *Bluespec SystemVerilog™ Reference Guide*, Bluespec, Inc, 2011.

[21] J. C. Hoe and Arvind, "Scheduling and Synthesis of Operation-Centric Hardware Descriptions," in *ICCAD*, 2000.

[22] Arvind, N. Dave, R. Nikhil, and D. Rosenband, "High-level synthesis: An Essential Ingredient for Designing Complex ASICs," in *ICCAD*, 2004.

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