McSimA+: A Manycore Simulator with Application-level+ Simulation and Detailed Microarchitecture Modeling

Jung Ho Ahn†, Sheng Li‡, Seongil O†, and Norman P. Jouppi‡
†Seoul National University, ‡Hewlett-Packard Labs
†{gajh, swdfish}@snu.ac.kr, ‡{sheng.li, norm.jouppi}@hp.com

***Abstract*—With their significant performance and energy advantages, emerging manycore processors have also brought new challenges to the architecture research community. Manycore processors are highly integrated complex system-on-chips with complicated core and uncore subsystems. The core subsystems can consist of a large number of traditional and asymmetric cores. The uncore subsystems have also become unprecedentedly powerful and complex with deeper cache hierarchies, advanced on-chip interconnects, and high-performance memory controllers. In order to conduct research for emerging manycore processor systems, a microarchitecture-level and cycle-level manycore simulation infrastructure is needed.**

This paper introduces McSimA+, a new timing simulation infrastructure, to meet these needs. McSimA+ models x86-based asymmetric manycore microarchitectures in detail for both core and uncore subsystems, including a full spectrum of asymmetric cores from single-threaded to multithreaded and from in-order to out-of-order, sophisticated cache hierarchies, coherence hardware, on-chip interconnects, memory controllers, and main memory. McSimA+ is an application-level+ simulator, offering a middle ground between a full-system simulator and an application-level simulator. Therefore, it enjoys the light weight of an application-level simulator and the full control of threads and processes as in a full-system simulator. This paper also explores an asymmetric clustered manycore architecture that can reduce the thread migration cost to achieve a noticeable performance improvement compared to a state-of-the-art asymmetric manycore architecture.

I. Introduction

Multicore processors have already become mainstream. Emerging manycore processors have brought new challenges to the architecture research community, together with significant performance and energy advantages. Manycore processors are highly integrated complex system-on-chips (SoCs) with complicated core and uncore subsystems. The core subsystems can consist of a large number of traditional and asymmetric cores. For example, the Tilera Tile64 [46] has 64 small cores. The latest Intel Xeon Phi coprocessor [9] has more than 50 medium-size cores on a single chip. Moreover, ARM recently announced the first asymmetric multicore processor known as big.LITTLE [12], which includes a combination of out-of-order (OOO) Cortex-A15 (big) cores and in-order (IO) Cortex-A7 (little) cores. While the Cortex-A15 has higher performance, the Cortex-A7 is much more energy efficient. Using them at the same time, ARM big.LITTLE targets high performance and energy efficiency at the same time. The uncore subsystems of the emerging manycore processors have also become more powerful and complex than ever, with features such as larger and deeper cache hierarchies, advanced on-chip interconnects, and high performance memory controllers. For example, the Intel Xeon E7-8870 already has a 30MB L3 cache. Scalable Network-on-Chip (NoC) and cache coherency implementation efforts have also emerged in real industry designs, such as the Intel Xeon Phi [9]. Moreover, emerging manycore designs usually require system software (such as OSes) to be heavily modified or specially patched. For example, current OSes do not support the multi-processing (MP) mode in ARM big.LITTLE, where both fat A15 cores and thin A7 cores are active. A special software switcher [12] is needed to support thread migration on the big.LITTLE processor.

Simulators have been prevalent tools in the computer architecture research community to validate innovative ideas, as prototyping requires significant investments in both time and money. Many simulators have been developed to solve different research challenges, serving their own purposes. However, new challenges brought by emerging (asymmetric) manycore processors as mentioned above demand new simulators for the research community. As discussed in the simulator taxonomy analysis in Section II, while high-level abstraction simulators are not appropriate for conducting microarchitectural research on manycore processors, full-system simulators usually are relatively slow, especially when system/OS events are not the research focus. Moreover, with unsupported features in existing OSes, such as the asymmetric ARM big.LITTLE processor, larger burdens are placed on researchers, especially when using a full-system simulator. Thus, a *lightweight*, *flexible*, and *detailed microarchitecture-level* simulator is necessary for research on emerging manycore microarchitectures. To this end, we make the following contributions in this paper:

- We introduce McSimA+. McSimA+ models x86 based (asymmetric) manycore (up to more than 1,000 cores) microarchitectures in detail for both core and uncore subsystems, including a full spectrum of asymmetric cores (from single-threaded to multithreaded and from in-order to out-of-order), cache hierarchies, coherence hardware, NoC, memory controllers, and main memory. McSimA+ is an application-level+ simulator, representing a middle ground between a full-system simulator and an application-level simulator. Therefore it enjoys the light weight of an application-level simulator and full control of threads and processes as in a full-system simulator. It is flexible in that it can support both execution-driven and trace-driven simulations. McSimA+ enables architects to perform detailed and holistic research on manycore architectures.
- We perform rigorous validations of McSimA+. The validations cover different processor configurations from the entire multicore processor to the core and uncore subsystems. The validation targets are compre-

978-1-4673-5779-1/13/$31.00 ©2013 IEEE

TABLE I. SUMMARY OF EXISTING SIMULATORS CATEGORIZED BY FEATURES. ABBREVIATIONS (DETAILS IN MAIN TEXT): (FS/A)-FULL-SYSTEM (FS) VS. APPLICATION-LEVEL (A), (DC)-DECOUPLED FUNCTIONAL AND PERFORMANCE SIMULATIONS, (μAR)-MICROARCHITECTURE DETAILS, (X86)-X86 ISA SUPPORT, (MC)-MANYCORE SUPPORT, (SS)-SIMULATION SPEED; (A+)-A MIDDLE GROUND BETWEEN FULL-SYSTEM AND APPLICATION-LEVEL SIMULATION, (Y)-YES, (N)-NO, (N/A)-NOT APPLICABLE, (P)-PARTIALLY SUPPORTED. †X86 IS NOT FULLY SUPPORTED FOR MANYCORE. *MANYCORE (E.G. 1,000 CORES AND BEYOND) IS NOT FULLY SUPPORTED DUE TO EMULATORS/HOST OSES. A PREFERRED MANYCORE SIMULATOR SHOULD BE LIGHTWEIGHT (A+ AND DC) AND REASONABLY FAST, WITH SUPPORT OF MC, μAR, AND X86. †UNLIKE OTHER SIMULATORS, THE CMP$IM FAMILY IS NOT PUBLICLY AVAILABLE.

Simulators	FS/A	DC	μAr	x86	Mc	SS	**Simulators**	FS/A	DC	μAr	x86	Mc	SS
gem5 [35]	FS	N	Y	Y	P*	+	**SimpleScalar [3]**	A	N	Y	N†	N	++
GEMS [30]	FS	Y	Y	N†	P*	+	**Booksim [22]**	N/A	N/A	Y	N/A	N	++
MARSSx86 [11]	FS	Y	Y	Y	P*	+	**Garnet [2]**	N/A	N/A	Y	N/A	N	++
SimFlex [45]	FS	Y	Y	N†	P*	+	**GPGPUsim [5]**	A	Y	Y	N/A	N	++
PTLsim [48]	FS	Y	Y	Y	P*	+	**DRAMsim [39]**	N/A	N/A	Y	N/A	N	++
Graphite [31]	A	Y	N	Y	Y	+++	**Dinero IV [19]**	A	N	Y	N/A	N	++
SESC [38]	A	N	Y	N†	N	++	**Zesto [26]**	A	N	Y	Y	N	+
Sniper [8]	A	Y	N	Y	Y	+++	**CMP$im [21], [33]**‡	A	Y	Y	Y	N	++
Preferred	A+	Y	Y	Y	Y	$\geq$ ++							

hensive ranging from a real machine to published results. In all validation experiments, McSimA+ demonstrates good performance accuracy.

- We propose an Asymmetry Within a cluster and Symmetry Between clusters (AWSB) design to reduce thread migration overhead in asymmetric manycore architectures. Using McSimA+, our study shows that the AWSB design performs noticeably better than the state-of-the-art clustered asymmetric architecture as adopted in ARM big.LITTLE.

II. WHY YET ANOTHER SIMULATOR?

Numerous processor and system simulators are already available as shown in Table I. All of these simulators have their own merits and serve their different purposes well. McSimA+ was developed to enable detailed asymmetric manycore microarchitecture research, and we have no intention to position our simulator as "better" than existing ones. For a better understanding of why we need another simulator for the above-mentioned purpose, we first navigate through the space of the existing simulators and explain why those do not cover the study we want to conduct. Table I shows the taxonomy of the existing simulators with the following six dimensions: 1) full-system vs. application-level simulation (FS/A), 2) decoupled vs. integrated functional and performance simulation (DC), 3) microarchitecture-level (i.e., cycle-level) vs. high-level abstract simulation (μAr), 4) supporting x86 or not (x86), 5) whole manycore system support or not (Mc), and 6) the simulation speed (SS).

a) Full-system (FS) vs. application-level simulation (A): Full-system simulators, such as gem5 [35] (full-system mode), GEMS [30], MARSSx86 [11], and SimFlex [45] run both applications and system software (mostly OSes). A full-system simulator is particularly beneficial when the simulation involves heavy I/O activities or extensive OS kernel function support. However, these simulators are relatively slow and make it difficult to isolate the impact of architectural changes from the interaction between hardware and software stacks. Moreover, because they rely on existing OSes, they usually do not support manycore simulations well. They also typically require research on both the simulator and the system software at the same time, even if the research targets only architectural aspects. For example, current OSes (especially Linux) do not support manycore processors with different core types; thus, OSes must be changed to support this feature. In contrast, these aspects are the specialties of application-level simulators, such as SimpleScalar [3], gem5 [35] (system-call emulation mode), SESC [38], and Graphite [31] along with its derivative Sniper [8]. However, a pure application-level simulation is insufficient, even if I/O activity and time/space sharing are not the main areas of focus. For example, thread scheduling in a manycore processor is important for both performance accuracy and research interests. Thus, it is desirable for application-level simulators to manage threads independently from the host OS and the real hardware on which the simulators run.

b) Decoupled vs. integrated functional and performance simulation (DC): Simulators need to maintain both functional correctness and performance accuracy. Simulators such as gem5 [35] choose a complex "execute-in-execute" approach that integrates functional and performance simulations to model microarchitecture details with very high levels of accuracy. However, to simplify the development of the simulator, some simulators trade modeling details and accuracy for reduced complexity and decouple functional simulation from performance simulation by offloading the functional simulation to third party software, such as emulators or dynamic instrumentation tools, while focusing on evaluating the performance of new architectures with benchmarks. This is acceptable for most manycore architecture studies, where reasonably detailed microarchitecture modeling is sufficient. For example, GEMS [30] and SimFlex [45] offload functional simulations to Simics [29], PTLSim [48] and its derivative MARSSx86 [11] offload functional simulations to QEMU [6], and Graphite [31] and its derivative Sniper [8] offload functional simulations to Pin [28].

c) Details (μAr) vs. simulation speed (SS): A manycore processor is a highly integrated complex system with a large number of cores and complicated core and uncore subsystems, leading to a tradeoff between simulation accuracy and speed. In general, the more detailed an architecture the simulator can handle, the slower the simulator simulation speed. For example, Graphite [31] uses less detailed models, such as the one-IPC model, to achieve better simulation speed. Sniper [8] uses better abstraction methods such as interval-

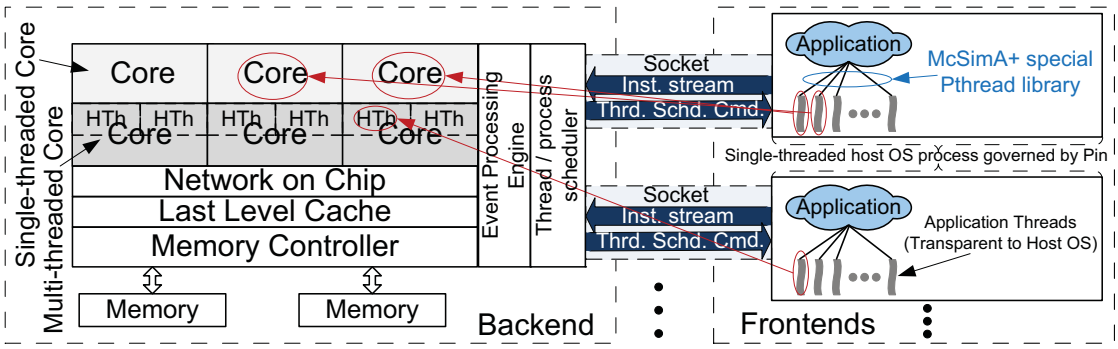


Fig. 1. McSimA+ infrastructure. Abbreviations: "Inst. stream"– instruction stream, "Thrd. Schd. Cmd"– thread scheduling commands.

based simulation to gain more accuracy with less performance overhead. While these simulators are good for early stage design space explorations, they are not sufficiently accurate for detailed microarchitecture-level studies of manycore architectures. Graphite [31] and Sniper [8] are considered faster simulators because they use parallel simulation to improve the simulation speed. Trace-driven simulations can also be used to trade simulation accuracy for speed. However, these are not suitable for multithreaded applications because the real-time synchronization information is usually lost when using traces. Thus, execution-driven simulations (i.e., simulation through actual application execution) are preferred. On the other hand, full-system simulators model both microarchitecture-level details and OSes. Thus, they sacrifice simulation speed for accuracy. Zesto [26] focuses on very detailed core-level microarchitecture simulations, which results in even lower simulation speeds. Instead, it is desirable to have a simulator to model manycore microarchitecture details while remaining faster than full-system simulators, which have both hardware and software overhead.

d) Support of manycore architecture (Mc): As this paper is about simulators for emerging (asymmetric) manycore architectures, it is important to assess existing simulators on their support of (asymmetric) manycore architectures. Many simulators were designed with an emphasis on one subsystem of a manycore system. For example, Booksim [22] and Garnet [2] focus on NoC; Dinero IV [19] and CMP$im [21], [33] family focus on the cache; DRAMsim [39] focuses on the DRAM main memory system, Zesto [26] focuses on cores with limited multicore support, and GPGPUSim [5] focuses on GPUs. Full-system simulators support multicore simulations but require non-trivial changes (especially to the OS) to support manycore systems stably with a large number (e.g., more than 1,000) of asymmetric cores. Graphite [31] and Sniper [8] support manycore systems but lack microarchitecture-level details, as mentioned earlier.

e) Support of x86 (x86): While it is arguable as to whether an ISA is a key feature for simulators given that many researches do not need support for a specific ISA, supporting the x86 ISA has advantages in reality because most studies are done on x86 machines. For example, complicated cross-platform tool chains are not needed in a simulator with x86 ISA support.

As shown in Table I, while existing simulators serve their purposes well, research on emerging (asymmetric) manycore processors prefers a new simulator that can *accurately* model the *microarchitecture* details of *manycore* systems. The new simulator is better at avoiding the weight of modeling both hardware and OSes so as to be *lightweight* yet still capable of controlling *thread management* for manycore processors. McSimA+ was developed specifically to fill this gap.

III. McSimA+: Overview and Operation

McSimA+ is a cycle-level detailed microarchitecture simulator for multicore and emerging manycore processors. Moreover, McSimA+ offers full control over thread/process management for manycore architectures, so it represents a middle ground between a full-system simulator and an application-level simulator. We refer to this as an *application-level+* simulator henceforth. It enjoys the light weight of an application-level simulator and better control of a full-system simulator. Moreover, its thread management layer makes implementing new functional features in emerging manycore processors much easier than changing the OSes with full-system simulators. McSimA+ supports detailed microarchitecture-level modeling not only of the cores, such as OOO, in-order, multi-threaded, and single-threaded cores, but also of all uncore components, including caches, NoCs, cache-coherence hardware, memory controllers, and main memory. Moreover, innovative architecture designs such as asymmetric manycore architectures and 3D stacked main-memory systems are also supported. By supporting the microarchitectural details and rich features of the core and uncore components, McSimA+ facilitates holistic architecture research on multicore and emerging manycore processors. McSimA+ is a simulator capable of decoupled functional simulations and timing simulations. As shown in Figure 1, there are two main areas in the infrastructure of McSimA+: 1) the Pin [28] based frontend simulator (*frontend*) for functional simulations and 2) the event-driven backend simulator (*backend*) for timing simulations.

Each frontend performs a functional simulation of a multi-threaded workload using dynamic binary instrumentation using Pin and generates the instruction stream for the backend timing simulation. Pin is a dynamic instrumentation framework that can instrument an application in the granularity of an instruction, a basic block, or a function. Applications being executed are instrumented by Pin and the information of each instruction, function call, and system call is delivered to the McSimA+ frontend. After being processed by the frontend, the information is delivered to the McSimA+ backend, where the detailed target system including cores, caches, directories, on-chip networks, memory controllers, and main-memory subsystems are modeled. Once the proper actions are performed by

the components affected by the instruction, the next instruction of the benchmark is instrumented by Pin and sent to the backend via the frontend. The frontend functional simulator also supports *fast forward*, an important feature necessary to skip instructions until the execution reaches the simulation region of interest.

The backend is an event-driven component that improves the performance of the simulation. Every architecture operation (such as a TLB/cache access, an instruction scheduling, and an NoC packet traversal) triggered by instruction processing generates a unique event with a component-type attribute (such as a core, a cache, and an NoC) and a time stamp. These events are queued and processed in a global event-processing engine. When processing the events, a series of architecture events may be induced in a chain reaction manner; the global processing engine shown in Figure 1 processes all of the events in a strict timing order. If events occur in a single cycle, the simulation is performed in a manner similar to that of a cycle-by-cycle simulation. However, if no event occurs in a cycle, the simulator can skip the cycle without losing any information. Thus, McSimA+ substantially improves the simulation speed without a loss of cycle-level accuracy compared to cycle-driven simulators.

A. Thread Management for Application-level+ Simulation

Although McSimA+ is not a full-system simulator, it is not a pure application-level simulator either. Given that a manycore processor includes a large number of cores, hardware threads, and complicated uncore subsystems, a sophisticated thread/process management scheme is needed. OSes and system software usually lag behind the new features in emerging manycore processors; thus, modifying OSes for full-system simulators is a heavy burden. Therefore, it is important to gain full control of thread management for manycore microarchitecture-level studies without the considerable overhead of a full-system simulation. By using thread management layer and by taking full control over thread management from the host OS, McSimA+ is an application-level+ simulator that represents a middle ground between a full-system simulator and an application-level simulator.

The fact that it is an application-level+ simulator is also important in how it reduces simulation overhead and improves performance accuracy. As a decoupled simulator, McSimA+ leverages Pin by executing applications on native hardware to achieve a fast simulation speed. One way to support a multithreaded application in this framework is to let the host OS (we borrow the terms used on virtual machines) orchestrate the control flow of the application. However, this approach has two drawbacks. First, it is difficult to micro-manage the execution order of each thread governed by the host OS. The timing simulator can make progress only if all the simulated threads held by all cores receive instructions to be executed or are explicitly blocked by synchronization primitives, whereas the host OS schedules the threads based on its own policy without considering the status of the timing simulator. This mismatch requires huge buffers to hold pending instructions, which is especially problematic for manycore simulations [32]. Second, if an application is not race free, we must halt the progress of a certain thread if it may change the flow of other threads that are pending in the host OS but may also be executed at an earlier time on the target architecture simulated in the timing simulator, which is a very challenging task.

B. Implementing the Thread Management Layer in McSimA+

When implementing the thread management layer in McSimA+ for an application-level+ simulation, we leveraged the solution proposed by Pan et al. [40] and designed a special Pthread [7] library[1] implemented as part of the McSimA+ frontend. This Pthread library enables McSimA+ to manage threads completely independently of the host OS and the real system according to the architecture status and characteristics of the simulated target manycore processors. There are two major components in the special Pthread library: the Pthread controller and the Pthread scheduler. The Pthread controller handles all Pthread functionalities, such as pthread_create, pthread_destroy, pthread_mutex, pthread local storage and stack management, and thread-safe memory allocation. The thread scheduler in our special Pthread library is responsible for blocking and resuming threads during thread join, mutex/lock competition, and conditional wait operations. Existing Pthread applications can run on McSimA+ without any change of the code. An architect only needs to link to the special Pthread library rather than to the native one. During execution, all Pthread calls are intercepted by the McSimA+ frontend and replaced with the special Pthread calls. In order to separate thread execution from the OS, a multithreaded application appears to be a single threaded process from the perspective of the host OS/Pin. Thus, OS/Pin is not aware of the threads in the host OS process and surrenders the full control of thread management and scheduling to McSimA+.

In order to simulate unmodified multi-programmed workloads (each workload can be a multithreaded application), multiple frontends are used together with a single backend timing simulator. All frontends are connected to the backend via inter-process communication (sockets). All threads from the frontend processes are mapped to the hardware threads in the backend and are managed by the process/thread scheduler in the backend, as shown in Figure 1. The thread scheduler in the Pthread library in the frontend maintains a queue of threads and schedules a particular thread to run when the backend needs the instruction stream from it. We implemented a global process/thread scheduler in the backend that controls the execution of all hardware threads on the target manycore processor. While the frontend thread scheduler manages threads according to the program information (i.e., the thread function calls), the backend process/thread scheduler has the global information (e.g. cache misses, resource conflicts, branch mispredictions, and other architecture events) of all of the threads in all processes and manages all of the threads accordingly. The backend scheduler sends the controlling information to the individual frontends to guide the thread scheduling process in each multithreaded application, with the help of the thread scheduler in the special Pthread libraries in the frontends. Different thread scheduling policies (the default is round-robin) can be implemented to study the effects of scheduling policies

[1] Building a full fledged special Pthread library requires a significant amount of work, even if our implementation is based on the preliminary implementation from Pan et al. [40]. First, we built important Pthread APIs, such as pthead_barrier, that were previously unsupported. Second, we re-implemented the library since the previous implementation was incompatible with the latest Pin. Third, we added 64-bit support for the library.

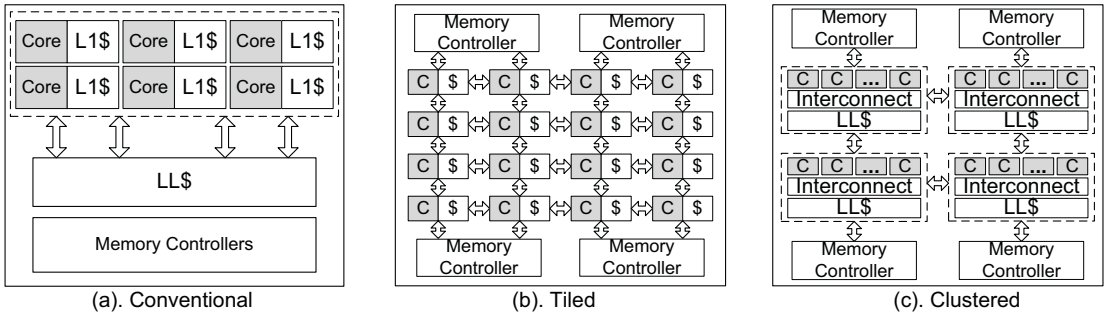


Fig. 2. Example manycore architectures modeled in McSimA+. (a) shows a fully connected (with a bus/crossbar) multicore processor such as the Intel Nehalem [24] and Sun Niagara [23] processors, where all cores directly share all last-level caches through the on-chip fully connected fabric. (b) shows a tiled architecture, such as the Tilera Tile64 [46] and Intel Knights Corner [9], where cores and local caches are organized as tiles and connected through a ring or a 2D-mesh NoC. (c) shows a clustered manycore architecture as proposed in [12], [25], [27], where on-chip core tiles first use local interconnects to form clusters that are then connected via ring or 2D-mesh NoC.

on the simulated system. Thus, as an application-level+ simulator, McSimA+ can be used to study advanced thread/process management schemes in manycore architectures.

IV. Microarchitecture Modeling of Asymmetric Manycore Architectures

The key focus of McSimA+ is to provide fast and detailed microarchitecture simulations for manycore processors. McSimA+ also supports flexible manycore designs. Figure 2 shows a few examples of the flexibility of McSimA+ in modeling different manycore architectures from a fully connected multicore processor (Figure 2(a)), such as the Intel Nehalem [24] and Sun Niagara [23], to tiled architectures (Figure 2(b)), such as the Tilera Tile64 [46] and Intel Knights Corner [9], and to clustered manycore architectures (Figure 2(c)) as in ARM big.LITTLE [12]. Moreover, McSimA+ supports a wide spectrum of innovative and/or emerging technologies, such as asymmetric cores [12] and 3D main memory [41]. By supporting detailed and flexible manycore architecture modeling, McSimA+ facilitates comprehensive and holistic research on multicore and manycore processors.

A. Modeling of Core Subsystem

McSimA+ supports detailed and realistic models of the scheduling units based on existing processor core designs, including in-order, OOO, and multithreaded core architectures. Figure 3 demonstrates the overall core models in McSimA+ for OOO and in-order cores. We depict the cores as a series of units and avoid calling them "pipeline stages," as they are high-level abstractions of the actual models in McSimA+ and because many detailed models of hardware structures (e.g., L1 caches and reservation stations) are implemented within these generic units.

1) Modeling of Out-of-Order Cores: The OOO core architecture in McSimA+ has multiple units, including the fetch, decode, issue, execution (exec), write-back, and commit stages, as shown in Figure 3(a). The fetch unit reads a cache line containing multiple instructions and stores the instructions in an instruction stream buffer. By modeling the instruction stream buffer, McSimA+ ensures that the fetch unit only accesses the TLB and instruction cache once for each cache line (with multiple instructions) rather than for each instruction. As pointed out in earlier work [26], most other academic simulators fail to model the instruction stream buffer and generate a separate L1-I$ request and TLB request for each instruction, which leads to overinflated accesses to the L1-I$ and TLB and subsequent incorrect simulation results. Next, instructions are taken from the instruction stream buffer and decoded. Because McSimA+ obtains its instruction stream from the Pin-based frontend, it can easily assign different latency levels based on the different instruction types and opcodes.

The issue unit assigns hardware resources to the individual instructions. By default, McSimA+ models the reservation-station (RS)-based (data-capture scheduler) OOO core following the Intel Nehalem [24]/P6 [18] microarchitectures. McSimA+ allocates a reorder buffer (ROB) entry and an RS entry to each instruction. If either resource is full, the instruction issue stalls until both the ROB and RS have available entries. Once instructions are issued to the RS, the operands available in either the registers or the ROB are sent to the RS entry. The designators of the unavailable source registers are also copied into the RS entry and are used for matching the results from functional units and waking up proper instructions; thus, only true read-after-write data dependencies may exist among the instructions in the RS.

The execution unit handles the dynamic scheduling of instructions, their movement between the reservation stations and the execution units, the actual execution, and memory instruction scheduling. While staying in the RS, instructions wait for their source operands to become available so that they can be dispatched to execution units. If the execution units are not available, McSimA+ does not dispatch the instructions to execute, even if the source operands of the instructions are ready. It is possible for multiple instructions to become ready in the same cycle. McSimA+ models the bandwidth of each execution unit, including both integer ALUs, floating point units, and load/store units. Instructions with operands ready bid on these dispatch resources, and McSimA+ arbitrates and selects instructions based on their time stamps to execute on the proper units. Instructions that fail in the competition have to stall and try again at the next cycle. For load and store units, McSimA+ assumes separate address generation units (AGU) are available for computing addresses as in the Intel Nehalem [24] processor.

The write-back unit deals with writing back the results of both non-memory and memory instructions. Once the result is

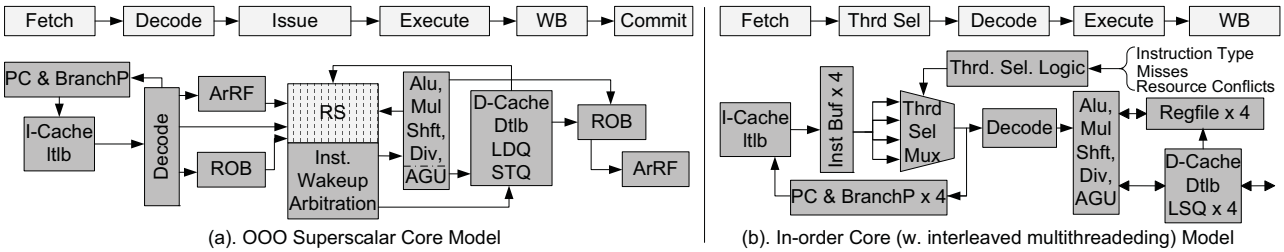


Fig. 3. Core modeling in McSimA+.

available, McSimA+ will update both the destination entry in the ROB and all entries with pending results in the RS. The RS entry will be released and marked as available for the next instruction. The commit unit completes the instructions, makes the results globally visible to the architecture state, and releases hardware resources. McSimA+ allows the user to specify the commit width.

2) Modeling of In-Order Cores: Figure 3(b) shows an in-order core with fine-grained interleaved multithreading as modeled in McSimA+. The core has six units, including the fetch, decode, select, execution (exec), memory, and write-back units. For an in-order core, the models of the fetch and decode units are similar to those of OOO cores, while the models of execution and writeback units are much simpler than those for OOO cores. For example, the model of the instruction scheduling structure for in-order cores in McSimA+ degenerates to a simple instruction queue. Figure 3(b) also shows the modeling of interleaved multithreading in McSimA+. This core model closely resembles the Sun Niagara [23] processor. McSimA+ models the thread selection unit after the fetch unit. McSimA+ maintains the detailed status of each hardware thread and selects one to execute on the core pipeline every cycle in a round-robin fashion from all active threads. A thread may be removed from the active list for various reasons. Threads can be blocked and marked as inactive by the McSimA+ backend due to operations with a long latency, such as cache misses and branch mispredictions or by the McSimA+ frontend thread scheduler owing to the locks and barriers within a multithreaded application. When selecting the thread to run in the next cycle, McSimA+ also considers resource conflicts such as competitions pertaining to execution units. McSimA+ arbitrates the competing active threads in a round-robin fashion, and a thread that fails will wait until the next cycle.

B. Modeling of Cache and Coherence Hardware

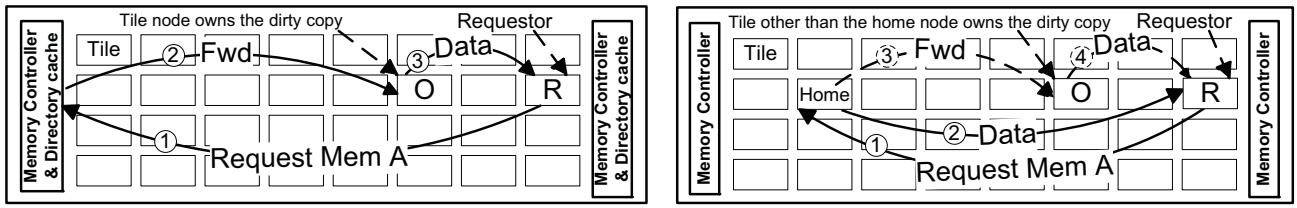
McSimA+ supports highly detailed models of cache hierarchies (such as private, coherent, shared, and non-blocking caches) to provide detailed microarchitecture-level modeling for both core and uncore subsystems in manycore processors. Faithfully modeling coherence protocol options for manycore processors is critical to model all types of cache hierarchies correctly. Because McSimA+ supports flexible compositions of cache hierarchies, the last-level cache (LLC) can be either private or shared. The address-interleaved shared LLC has a unique location for each address, eliminating the need for a coherence mechanism. However, even when the LLC is shared, coherence between the upper-level private (e.g., L1 or L2) caches must be explicitly maintained. Figure 4 shows the tiled architecture with a private LLC to demonstrate the coherence models in McSimA+. We assume directory-based coherence because McSimA+ targets future manycore processors that can have 64 or more cores, where frequent broadcasts are slow, difficult to scale, and power-hungry.

McSimA+ supports three mainstream directory-based cache coherence implementations (to enable important trade-off studies of the performance, energy, scalability, and complexity of different architectures): the DRAM directory with a directory cache (DRAM-dir, as shown in Figure 4(a)) as in the Alpha 21364 [20], the distributed duplicate tag (duplicate-tag, as shown in Figure 4(b)) as in the Niagara processors [23], [36], and the distributed sparse directory (sparse-dir, as shown in Figure 4(b)) [13].

DRAM-dir is the most straightforward implementation; it stores directory information in main memory with an additional bit-vector for every memory block to indicate the sharers. While the directory information is logically stored in DRAM, performance requirements may dictate it to be cached in the on-chip directory caches that are usually co-located at the on-chip memory controllers, as the directory cache has frequent interactions with main memory. Figure 4(a) demonstrates how the DRAM-dir is modeled in McSimA+. Each core is a potential sharer of a cache block. A cache miss triggers a request and sends it through the NoC to the appropriate memory controller based on address interleaving to where the target directory cache resides. The directory information is then retrieved. If the data is on chip, the directory information manages the data forwarding between the owner and the sharers. If a directory cache miss/eviction occurs, McSimA+ generates memory accesses at a memory controller and fetches the directory information (and the data if needed) from the main memory.

McSimA+ supports both the duplicate-tag and the sparse-dir features to provide smaller storage overheads than DRAM-dir and to make the directory scalable for processors with a large number of cores. The duplicate-tag maintains a copy of the tags of every possible cache that can hold the block, and no explicit bit vector is needed for sharers. During a directory lookup operation, tag matches indicate finding by the sharers. The duplicate-tag eliminates the need to store and access the directory information in DRAM. A block not found in a duplicate tag is known to be uncached.

Despite its good coverage for all of the cached memory blocks, a duplicate-tag directory can be challenging as the number of cores increases because its associativity must equal the product of the cache associativity and the number of caches [4]. McSimA+ supports sparse-dir [37] as a low-cost alternative to the duplicate-tag directory. Sparse-dir reduces the degree of directory associativity but increases the number



(a) DRAM-dir

(b) Duplicate-tag and Sparse-dir, both with home nodes

Fig. 4. Cache coherence microarchitecture modeling in McSimA+. In DRAM-dir (a) model, each tile contains core(s), private cache(s), local interconnect (if necessary) within a tile, and global interconnect for inter-tile communications. Directory caches are co-located with memory controllers. In Duplicate-tag and Sparse-dir (b), McSimA+ assumes that directory is distributed across the tiles using the *home node* concept [9], [23], [36]. Thus, the tiles in (b) have the extra directory information although not shown in the figure.

of directory sets. Because this operation loses the one-to-one correspondence of directory entries to cache frames, each directory entry is extended with the bit vector for storing explicit sharer information. Unfortunately, the non-uniform distribution of entries across directory sets in this organization incurs set conflicts, forcing the invalidation of cached blocks tracked by the conflicting directory entries and thus reducing the performance of the system. McSimA+ provides all these different designs to facilitate in-depth research of manycore processors.

As shown in Figure 4(a), a coherent miss in DRAM-dir generates NoC traffic, and the request needs to travel through the NoC to reach the directory even if the data is located nearby. In order to model scalable duplicate-tag directories and sparse-dirs, we model the home node-based distributed implementation as in the Intel Xeon Phi [9] and Niagara processors [23], [36], where the directory is distributed among all nodes by mapping a block address to the *home node*, as shown in Figure 4(b). We assume that home nodes are selected by address interleaving on low-order blocks or page addresses. A coherent miss first looks up the directory in the home node. If the home node has the directory and data, the data will be sent to the request directly via steps (1)-(2) shown in Figure 4(b). The home node may only have the directory information without the latest data, in which case the request will be forwarded to the owner of the copy and the data will be sent from there via steps (1), (3), and (4), as shown in Figure 4(b). If a request reaches the home node but fails to find a matching directory entry, it allocates a new entry and obtains the data from memory. The retrieved data is placed in the home tile's cache and a copy is returned to the requesting core. Before victimizing a cache block with an active directory state, the protocol must first invalidate sharers and write back dirty copies to memory.

C. Modeling of Network-on-Chips (NoCs)

McSimA+ supports different on-chip interconnects, including buses, crossbars, and multi-hop NoCs with various topologies, including ring and 2D mesh topologies. A multi-hop NoC has links and routers, where the per-hop latency is a tunable parameter. As shown in Figure 2, McSimA+ supports a wide range of hierarchical NoC designs, where cores are grouped into local clusters and the clusters are connected by global networks. The global interconnects can be composed of buses, crossbars, or multi-hop NoCs. McSimA+ models different message types (e.g., data blocks, addresses, and acknowledgements) that route in the NoC of a manycore processor. Multiple protocol-level virtual channels in the NoC are used to avoid deadlocks in the on-chip transaction protocols. A protocol-level virtual channel is also modeled to have multiple virtual channels inside to avoid a deadlock within the NoC hardware and improve the performance of the network.

McSimA+'s detailed message and virtual channel models not only guarantee simulation correctness and performance accuracy but also facilitate important microarchitecture research on NoCs. For example, when designing a manycore processor with a NoC, it is often desirable to have multiple independent logical networks for deadlock avoidance, privilege isolation, independent flow control, and traffic prioritization purposes. However, it is an interesting design choice as to whether the different networks should be implemented as logical or virtual channels over one large network, as in the Alpha21364 [20], or as independent physical networks as in Intel Xeon Phi [9]. An architect can conduct in-depth studies of these alternatives using McSimA+.

D. Modeling of the Memory Controller and Main Memory

McSimA+ supports detailed modeling of memory controllers and main-memory systems. First, the placement of memory controllers, an important design choice [1], can be freely determined by the architects. As shown in Figure 2, the memory controllers can be connected by crossbars/buses and placed at edges. They can also be distributed throughout the chip and connected to the routers in the NoC. McSimA+ supports numerous memory scheduling policies, including FCFRFS [43] and PAR-BS [34]. For each memory scheduling policy, an architect can further choose to use either open-page or close-page scheduling policies on top of the base scheduling policy. For example, if the PAR-BS policy is assumed to be the base memory scheduling policy, a close-page policy on top of it will close the DRAM page when there is no pending access in the scheduling queue to the current open DRAM page. Moreover, the modeled memory controller also supports a DRAM power-down mode during which DRAM chips consume only a fraction of their normal static power but require extra cycles to enter and exit the state. When this option is chosen, the controller will schedule the main memory to enter a power-down mode after the scheduling queue is empty and thus the attached memory system has been idle for a predefined interval. This facilitates research on trade-offs between power-saving benefits and performance penalties.

In order to model the main-memory system accurately, the main-memory timing is also rigorously modeled in McSimA+. For the current and near-future standard DDRx memory

TABLE II. CONFIGURATION SPECIFICATIONS OF THE VALIDATION TARGET SERVER WITH INTEL XEON E5540 MULTI-CORE PROCESSOR. IF/CM/IS STANDS FOR FETCH/COMMIT/ISSUE.

Freq (GHz)	2.53	**RS entry**	36	**(IF/CM/IS) width**	4/4/6	**L2$ per core**	256KB, 8-way, inclusive
Cores/chip	4	**L1 I-TLB entry**	128	**L1 I-$**	32KB, 4-way	**L3$ (shared)**	8MB, 16-way, inclusive
ROB entry	128	**L1 D-TLB entry**	64	**L1 D-$**	32KB, 8-way	**Main memory**	3 channels, DDR3-1333

systems, McSimA+ includes user-adjustable memory timing parameters such as row activation latency, precharge latency, row access latency, column access latency, and the row cycle time with different banks.

V. VALIDATION

There are two aspects in the validation of an execution-driven architectural simulator: functional correctness that guarantees programs to finish correctly and performance accuracy that ensures that the simulator faithfully reflects the performance of the execution, as if the applications were running on the actual target hardware. Functional correctness is typically straightforward to verify, especially for the simulators with decoupled functional simulations such as GEMS [30], SimFlex [45] and our McSimA+. We checked the correctness of the simulation results on SPLASH-2 using the correctness check option within each program. However, performance accuracy is much more difficult to verify. Moreover, a recent trend (as in a recent workshop panel [10] with several industrial researchers) argues that provided that academic simulators can foster correct research insights through simulations the validation of the simulators against real systems is not necessary. This trend partially leads to the fact that the majority of existing academic simulators lack sufficient validation against real systems. However, considering that McSimA+ focuses on microarchitecture-level simulations for manycore processors, we believe that a rigorous validation against actual hardware systems is required. We performed the validations in layers, first validating at the entire multicore processor level and then validating the core and uncore subsystems.

The performance accuracy of McSimA+ at an overall multicore processor level was validated using the multithreaded benchmark suite SPLASH-2 [47] against an Intel Xeon E5540 (Nehalem [24]) based real server whose configuration specifications (listed in Table II) were used to configure the simulated target system in McSimA+. For all of the validations, we turned off hyper-threading and the L2 cache prefetcher in the real server and configured McSimA+ accordingly. Figure 5 shows the IPC (Instructions Per Cycle) results of the SPLASH-2 simulations on McSimA+ normalized to the IPCs of the native executions on the real server as collected using Intel Vtune [17]. When running benchmarks on the real machines, we ran the applications multiple times to minimize the system noise. As shown in Figure 5, the IPC results of the SPLASH-2 simulations on McSimA+ are in good agreement with the native executions, which have an average error of only 2.1% (14.2% on average for absolute errors). Its standard deviation is also as low as 12%.

We then validated the performance accuracy of McSimA+ at the core level using SPEC CPU2006 benchmarks, which are good candidates for validation because they are popular and single-threaded. The same validation target shown in Table II was used. Figure 5 shows the IPC results of McSimA+ simulations normalized to native machine executions on the real server for SPEC CPU2006. The simulation results track the native execution result from the real server very well, with an average error of only 5.7% (15.4% on average for absolute errors) and a standard deviation of 17.7%.

While the core subsystem validation is critical, the uncore subsystems of the processor are equally important. To validate the uncore subsystems, we focused on the last-level cache (LLC), as LLC statistics represent the synergy between cache/memory hierarchy and on-chip interconnects. We used SPLASH-2 benchmarks to validate the cache miss rates for the LLC, where both the cache size and the associativity vary to a large degree, ranging from 1KB to 1MB and from one way to fully-associative, respectively. We used the results published in the original SPLASH-2 paper [47] as the validation targets because it is not practical to change the cache size or associativity on a real machine. We configured the simulated architecture as close as possible to the architecture (a 32-processor symmetric multiprocessing system) in the original paper [47]. Validation results on Cholesky and FFT are shown in Figure 6 as representatives. While FFT is highly scalable, Cholesky is dramatically different with poor scalability. As shown in Figure 6, the miss rate results obtained from McSimA+ very closely match the corresponding results reported in the earlier work [47]. For all SPLASH-2 benchmarks (including examples shown in Figure 6), the LLC miss rate difference between McSimA+ and the validation target does not exceed 2% over hundreds of data points collected at one time. This experiment demonstrates the high accuracy of McSimA+'s uncore subsystem models.

Our validation covers different processor configurations ranging from the entire multicore processor to the core and the uncore subsystems. The validation targets are comprehensive ranging from a real machine to published results. Thus, the validation stresses McSimA+ in a comprehensive and detailed way as well as tests its simulation accuracy with different processor architectures. In all validation experiments, McSimA+ demonstrates good performance accuracy.

VI. CLUSTERING EFFECTS IN ASYMMETRIC MANYCORE PROCESSORS

We illustrate the utility of McSimA+ by applying it to the study of clustering effects in emerging asymmetric manycore architectures. Asymmetric manycore processors, such as ARM big.LITTLE, have cores with different performance and power capabilities (e.g., *fat* OOO and *thin* in-order (IO) cores) on the same chip. Clustered manycore architectures (Figure 2(c)), as proposed in several studies [14], [25], [27] have demonstrated significant performance and power advantages over flat tiled manycore architectures (Figure 2(b)) due to the synergy of cache sharing and scalable hierarchical NoCs. Moreover, clustering has already been adopted in ARM big.LITTLE, the first asymmetric multicore design from industry. Despite the adoption of clustering in asymmetric multicore designs, effectively organizing clusters in a manycore processor remains

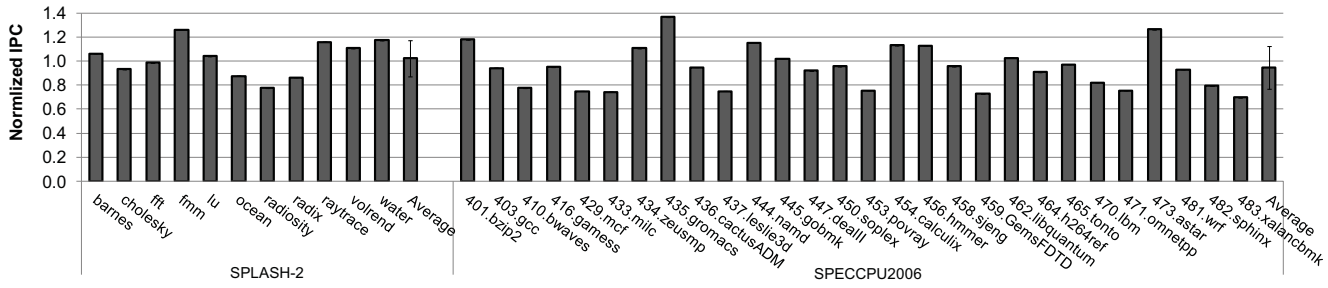


Fig. 5. The relative IPC of McSimA+ simulation results normalized to that of the native machines. We use the entire SPLASH-2 and SPEC CPU2006 benchmark suite.

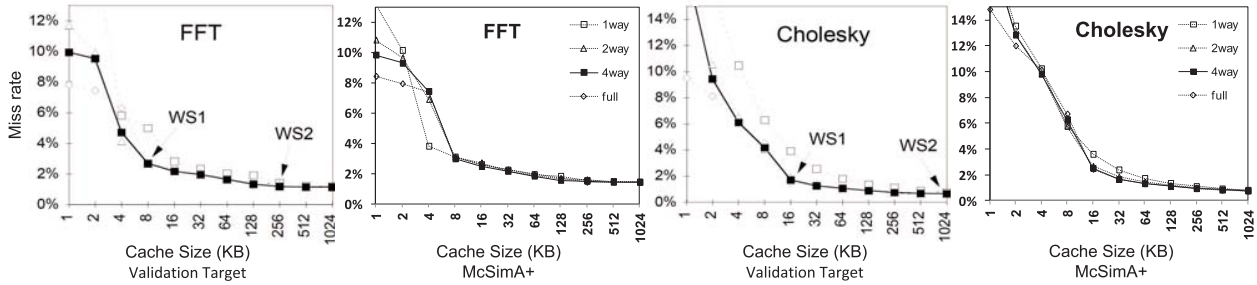


Fig. 6. Validation of McSimA+ L2 cache simulation results to the simulation results from [47].

an open question. Here, we perform a detailed study of clustered asymmetric manycore architectures to provide insights regarding this question.

A. Manycore with Asymmetry Within or Between Clusters

There are two clustering options for an asymmetric manycore design as shown in Figure 7. The first option is to have Symmetry Within a cluster and Asymmetry Between clusters (SWAB) as illustrated in Figure 7(a), where cores of the same type are placed within a single cluster but where different clusters can have different core types. SWAB is the clustering option used in the ARM big.LITTLE design. The second option, which we propose, is to have Asymmetry Within a cluster and Symmetry Between clusters (AWSB), as illustrated in Figure 7(c). AWSB places different cores in a single cluster and forms an asymmetric cluster, but all clusters in a chip are symmetric despite the asymmetry within a single cluster.

Generally, thin (e.g., in-order) cores can achieve good performance for workloads with inherently high degrees of (static) instruction-level parallelism (ILP) (where ILP does not need to be dynamically extracted because the subsequent instructions in the stream are inherently independent), while fat (e.g., OOO) cores can easily provide good performance for workloads with hidden ILP (where the instructions in the stream need to be reordered dynamically to extract ILP). Thus, it is critical to run workloads on appropriate cores to maximize the performance gain and energy savings. In addition, the behavior of an application can vary at a fine-grained time scales during execution because of phase changes (e.g., a switch between computation-intensive and memory-intensive phases). Thus, frequent application/thread migrations may be necessary to fully exploit the performance and energy advantages of asymmetric manycore processors.

However, thread migrations are not free. In typical manycore architectures as shown in Figure 2, thread migrations have two major costs: 1) the architecture-state migration cost, including the transfer of visible architecture states (e.g., transferring register files, warming up a branch prediction table and TLBs) and allowing invisible architecture states to become visible (drain a core pipeline, finish/abort speculation execution, for example); and 2) the cache data migration cost. In this paper, we focus on a heterogeneous multi-processing system (i.e., the MP mode of the big.LITTLE [12] processor), in which all cores are active at the same time. Thus, a thread migration always involves at least a pair of threads/cores, and all cores involved in the migration will have new tasks to execute after the migration. The cache data migration cost varies significantly according to the cache architecture. Migration within a shared cache does not involve any extra cost, while migration among private caches requires the transfer of data from an old private cache to a new private cache. Although it can be handled nicely by coherence protocols without off-chip memory traffic, data migration among private caches is still very expensive when the capacity of the last-level caches are large, especially when all cores involved in the thread migration will have new tasks to execute after the migration and thus will have to update their private caches.

Because the architecture-state migration cost is inevitable, it is critical to reduce the amount of cache data migration to support fine-grained thread migration so as to fully exploit the performance and energy advantages of asymmetric manycore processors. Thus, we propose AWSB, as in shown Figure 7(c) to support finer-grained thread migrations via its two-level thread migration mechanism (i.e., intra-cluster and inter-cluster migrations). Because AWSB has clusters consisting of asymmetric cores, thread migration can be and is preferred within a cluster. Only when no candidates can be found within the same cluster (and the migration is very necessary to achieve higher performance and energy efficiency), an inter-cluster migration is performed. However, for SWAB, as shown in Figure 7(a), only high-overhead inter-cluster migrations are

TABLE III. PARAMETERS INCLUDING AREA AND POWER ESTIMATIONS OBTAINED FROM MCPAT [25] OF BOTH OOO AND IO CORES.

Parameters	Issue width	RS	ROB	L1D cache	L2 cache	Area (mm^2)	Power (W)
OOO (Nehalem [24]-like)	6 (peak)	36	128	32KB, 8-way	2MB 16-way	6.56	3.97
IO (Atom [16]-like)	2	N/A	N/A	16KB, 4-way	512KB 16-way	2.15	0.66

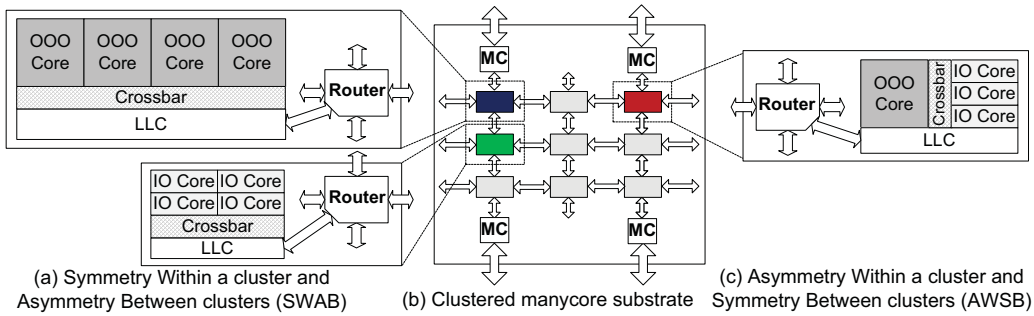


Fig. 7. Clustered manycore architectures. (a) Symmetry Within a cluster and Asymmetry Between clusters (SWAB), fat OOO clusters (blue) and thin IO core cluster (green). (b) Generic clustered manycore processor substrate. (c) Asymmetry Within a cluster and Symmetry Between clusters (AWSB) (red).

possible when the mapping between workloads and core types needs to be changed. Thus, by supporting two-level thread migrations, AWSB has the potential to reduce the migration cost and increase the migration frequency for a better use of the behavioral changes in the application execution and to achieve better system performance and energy efficiency than SWAB.

B. Evaluation

Using McSimA+, we evaluate our AWSB proposal, as shown in Figure 7(c), and compare it to the SWAB design adopted in the ARM big.LITTLE, as shown in Figure 7(a). We assume two core types (both 3GHz) are used in the asymmetric manycore processors, an OOO Nehalem [24]-like fat core and an in-order Atom [16]-like thin core. The parameters of both cores, including the area and the power estimations obtained from McPAT [25], are listed in Table III. We assume a core count ratio of fat cores to thin cores of 1:3 so that both fat and thin cores occupy a similar silicon area overall. Each fat core is assumed to have a 2MB L2 cache based on the Nehalem [24] design, while each thin core is assumed to have a 512KB L2 cache based on the Pineview Atom [16] design. Based on the McPAT [25] modeling results, a processor with 22nm technology with a $\sim$260mm^2 die area and a $\sim$90W thermal design power (TDP) can accommodate 8 fat cores and 24 thin cores together with L2 caches, an NoC, and 4 single-channel memory controllers with DDR3-1600 DRAM connected. The AWSB architecture has 8 clusters with each cluster containing 1 fat core and 3 thin cores. The SWAB architecture has 2 fat clusters each containing 4 identical fat cores and 6 thin clusters each containing 4 thin cores. All of the cores in a cluster share a multi-banked L2 cache via an intra-cluster crossbar. Because both AWSB and SWAB have 8 clusters, the same processor-level substrate as shown in Figure 7(b) is used with an 8-node 2D mesh NoC having a data width of 256 bits for inter-cluster communication. A two-level hierarchical directory-based MESI protocol is deployed to maintain cache coherency and to support private cache data migrations. Within a cluster, the L2 cache is inclusive and filters the coherency traffic between L1 caches and directories. Between clusters, coherence is maintained by directory caches associated with the on-chip memory controllers.

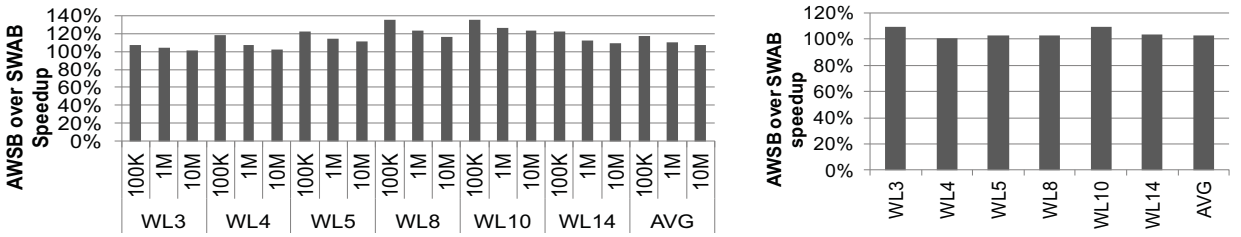
We constructed 16 mixed workloads, as shown in Figure 8 using the SPEC CPU2006 [15] suite for evaluating SWAB and AWSB. Because there are 32 cores on the chip in total, each of the workloads contains 32 SPEC CPU2006 benchmarks, and some benchmarks are used more than once in a workload. Some of the workloads (e.g., WL-5, as shown in Figure 8) contain more benchmarks with high IPC speedup, while others (e.g., WL-1) contain more benchmarks with low IPC speedup.

We first evaluated the thread migration overhead on the SWAB and AWSB architectures. We deployed all 32 benchmarks on all 32 cores for both SWAB and AWSB with the same benchmark to core mapping and then initiated a thread migration to change the mapping after an interval with 100K, 1M, or 10M instructions. The thread migration occurs during every interval until the simulation reaches 10 billion instructions or finishes earlier. Figure 9(a) shows the AWSB over SWAB speedup (measured as the ratio of the aggregated IPC) of the asymmetric 32 core processors. As shown in Figure 9(a), AWSB demonstrated much higher performance, especially when the thread migration interval is small. For example, AWSB shows a 35% speedup over SWAB when running workload 8 (WL-8) at a thread migration interval of 100K instructions. On average, the AWSB architecture achieves 18%, 11%, and 8% speedup over the SWAB architecture with a thread migration interval of 100K instructions, 1M instructions, and 10M instructions, respectively. While the benchmark to core mapping changes from interval to interval, the SWAB and AWSB architectures have the same mapping at each interval. Thus, the performance differences observed from Figure 9(a) are solely caused by the inherent differences in the thread migration overhead between the SWAB and AWSB architectures, and the results demonstrate AWSB's better support of thread migration among the asymmetric cores.

We then evaluated the implications of the thread migration overhead on the overall system performance. We deployed 32 benchmarks in each workload to all cores in SWAB and AWSB with the same benchmark to core mapping scheme and then initiated a thread migration every 10M instructions. Unlike the previous study, in which SWAB and AWSB always

SPEC CPU2006	445	458	400	453	483	471	473	470	437	410	459	450	429	436	482	433	464	465	403	401	416	447	462	444	434	454	456
WorkLoad \ Spdup	2.3	2.5	2.7	2.7	2.8	2.9	3.3	3.3	3.5	3.6	3.7	3.7	3.8	3.8	3.8	3.9	3.9	4.0	4.0	4.1	4.5	4.6	4.7	5.2	5.2	6.3	6.4
WL-1	2	3	3	5	2	1	1	1	3		1	3	1		1	1		3	1								
WL-2	2	3	3	2	2		3	2			1		1			2	1	5		1		1	2	1			
WL-3		2	1	1	1		2		2	1		1	1	2	1		1	2	2	3	2	2	1		2	2	
WL-4	1	1		1		1	1	1	1		2		1	1			3	1	1	1	4	2		2	2	1	4
WL-5										1	1	1		1		1		4	1	2	2	1	3	5	1	3	5
WL-6	2	1	5	1			2		2	1		2	2			2		3	2	2	3				1	1	
WL-7	1	3		4			1	1	3	2		2	2			1		2	3		3				1	2	1
WL-8			2		2		1	1	2		1	3	3		1	1	2	2	1	2	4		1	1	2		
WL-9	2	1	3	1	2			2	1	1	2	1	1	1		1	1	1		2			3			4	2
WL-10	3	1	1	1							1	2		1	1	3	2	3	3	2		3	1	1	2		1
WL-11	1	2	2	3		1	1		1		2	1	1		2	2	2		3	1	2	2				2	1
WL-12	2	3	2	1	1		3	2	2		3	2					1				2	2	2			1	3
WL-13	1	3	1	1	2		1	3	1	1	2	1				1	3	2	1	1	2	2			1	2	
WL-14	1	1	1		2		2	2	1	1	2	2	2		1	2	2	1	2		3	1		1		1	1
WL-15			1	1			1	2	1		2	2	1	2	1	1	1	1	3	1	3	1	1	1	3		2
WL-16	4	1	3		1		2		2		1	2	2	1	1	2	1	1	1	2	1	2	1			1	

Fig. 8. Mixed workloads used in the case study constructed from SPEC CPU2006 benchmarks. The benchmarks are sorted by IPC speedup (the IPC on fat cores over the IPC on thin cores) from the lowest to the highest. Each row represents a mixed workload, where the box representing a benchmark is marked gray if it is selected and the number in the box indicates the number of copies of this benchmark used in the workload.



(a) Thread migration induced performance difference.

(b) Performance difference with optimized thread migration.

Fig. 9. Performance comparison between SWAB and AWSB architectures. (a) Thread migration induced performance difference on SWAB and AWSB architectures with different thread migration intervals of 100K instructions, 1M instructions, and 10M instructions. (b) Performance difference between SWAB and AWSB architectures when running workloads with dynamic thread migration to run applications on appropriate cores with intervals of 10M instructions. Both figures show a subset of the 16 workloads due to limited space, but the averages (AVGs) are over all 16 workloads for both figures.

have the same benchmark to core mapping so as to isolate the thread migration overhead, this study allows both SWAB and AWSB to select the appropriate migration targets for each benchmark. At the end of each interval, McSimA+ initiates a thread migration to place the high IPC speedup benchmarks on the fat cores with the low IPC speedup on the thin cores, as in earlier work [44].[2] As shown in Figure 9(b), AWSB demonstrates a noticeable performance improvement of more than 10% for workloads 3 and 8, with a 4% improvement on average for all 16 workloads. It is expected that the benefits of AWSB will be higher with finer-grained thread migrations, because the thread migration overhead of AWSB becomes much smaller than that of SWAB when moving to finer-grained thread migrations, as shown in Figure 9(a).

VII. Limitations and Scope of McSimA+

There is no single "silver bullet" simulator that can satisfy all of the research requirements of the computer architecture community, and McSimA+ is no exception. Although it takes advantages of full-system simulators and application-level simulators by having an independent thread management layer, McSimA+ still lacks the support of system calls/codes (the inherent limitation of application-level simulators). Therefore, research on OSes and applications with extensive system events (e.g. I/Os) is not suitable for McSimA+. Because the Pthread controller in the frontend Pthread library is specific to the thread interface, non-Pthread multithreaded applications cannot run on McSimA+ without re-targeting the thread interface despite the fact that the frontend Pthread scheduler and backend global process/thread scheduler are feasible despite the particular thread interface used. McSimA+ targets emerging manycore architectures with reasonably detailed microarchitecture modeling, and outside its scope it is most likely suboptimal as compared to other suitable simulators.

Another limitation is the modeling of speculative wrong-path executions. Because McSimA+ is a decoupled simulator that relies on Pin for its functional simulation, wrong-path instructions cannot be obtained naturally from Pin, as they were never committed in the native hardware and are thus invisible beyond the ISA interface. However, this limitation is different from the inherent limitation of lacking the support of system calls. Although speculative wrong-path executions are not supported at this stage, they can be implemented via the context (architectural state) manipulation feature of Pin, as used to implement the thread management layer. The same approach can be employed to guide an application to execute a wrong path, roll back an architectural state, and execute a correct path.

VIII. Conclusions and User Resources

This paper introduces McSimA+, a cycle-level simulator to satisfy new demands of manycore microarchitecture research. McSimA+ supports asymmetric manycore systems in detail for

[2] We made oracular decisions on migration targets as we have the IPC values of each application on specific moments in McSimA+. The actual implementation of IPC estimators for thread migration is a hot research topic [42], [44] and beyond the scope of this paper.

comprehensive core and uncore subsystems, and can be scaled to support 1,000 cores or more. As an application-level+ simulator, McSimA+ takes advantage of full-system simulators and application-level simulators, while avoiding the deficiencies of both. McSimA+ enables architects to perform detailed and holistic research on emerging manycore architectures. Using McSimA+, we explored clustering design options in asymmetric manycore architectures. Our case study showed that the AWSB design, which provides asymmetry within a cluster instead of between clusters, reduces the thread migration overhead and improves performance noticeably compared to the state-of-the-art SWAB-style clustered asymmetric manycore architecture. McSimA+ and its documentation are available online at http://code.google.com/p/mcsim/.

ACKNOWLEDGMENTS

We gratefully acknowledge Ke Chen from University of Notre Dame for his helpful comments. Jung Ho Ahn is partially supported by the Smart IT Convergence System Research Center funded by the Ministry of Education, Science and Technology (MEST) as Global Frontier Project and by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the MEST (2012R1A1B4003447).

REFERENCES

[1] D. Abts *et al.*, "Achieving Predictable Performance through Better Memory Controller Placement in Many-Core CMPs," in *ISCA*, 2009.

[2] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A Detailed On-Chip Network Model Inside a Full-System Simulator," in *ISPASS*, 2009.

[3] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer*, vol. 35, no. 2, 2002.

[4] J. L. Baer and W. H. Wang, "On the Inclusion Properties for Multi-Level Cache Hierarchies," in *ISCA*, 1988.

[5] A. Bakhoda *et al.*, "Analyzing CUDA Workloads Using a Detailed GPU Simulator." in *ISPASS*, 2009.

[6] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *ATEC*, 2005.

[7] D. R. Butenhof, *Programming with POSIX threads*, 1997.

[8] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation," in *SC*, 2011.

[9] G. Chrysos, "Intel Many Integrated Core Architecture," in *Hot Chips*, 2012.

[10] D. Burger, S. Hily, S. Mckee, P. Ranganathan, and T. Wenisch, "Cycle-Accurate Simulators: Knowing When to Say When," in *ISCA Panel Session*, 2008.

[11] K. Ghose *et al.*, "MARSSx86: Micro Architectural Systems Simulator," in *ISCA Tutorial Session*, 2012.

[12] P. Greenhalgh, "Big.LITTLE Processing with ARM Cortex^{TM}-A15 & Cortex-A7," *ARM White Paper*, 2011.

[13] A. Gupta, W.-D. Weber, and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," in *ICPP*, 1990.

[14] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," in *ISCA*, 2009.

[15] J. L. Henning, "Performance Counters and Development of SPEC CPU2006," *Computer Architecture News*, vol. 35, no. 1, 2007.

[16] Intel, http://www.intel.com/products/processor/atom/techdocs.htms.

[17] Intel, "Intel VTune Performance Analyzer," http://software.intel.com/en-us/intel-vtune/.

[18] Intel, "P6 Family of Processors Hardware Developer's Manual," *Intel White Paper*, 1998.

[19] J. Edler and M. D. Hill, "Dinero IV," http://www.cs.wisc.edu/~markhill/DineroIV.

[20] A. Jain *et al.*, "A 1.2 GHz Alpha Microprocessor with 44.8 GB/s Chip Pin Bandwidth," in *ISSCC*, 2001.

[21] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "Cmp$im: A Binary Instrumentation Approach to Modeling Memory Behavior of Workloads on CMPs," University of Maryland, Tech. Rep., 2006.

[22] N. Jiang *et al.*, "A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator," in *ISPASS*, 2013.

[23] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, vol. 25, no. 2, 2005.

[24] R. Kumar and G. Hinton, "A Family of 45nm IA Processors," in *ISSCC*, 2009.

[25] S. Li *et al.*, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *MICRO*, 2009.

[26] G. H. Loh, S. Subramaniam, and Y. Xie, "Zesto: A Cycle-Level Simulator for Highly Detailed Microarchitecture Exploration," in *ISPASS*, 2009.

[27] P. Lotfi-Kamran *et al.*, "Scale-Out Processors," in *ISCA*, 2012.

[28] C. K. Luk *et al.*, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, 2005.

[29] P. S. Magnusson *et al.*, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[30] M. M. Martin *et al.*, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *Computer Architecture News*, vol. 33, no. 4, 2005.

[31] J. E. Miller *et al.*, "Graphite: A Distributed Parallel Simulator for Multicores," in *HPCA*, 2010.

[32] M. Monchiero *et al.*, "How to Simulate 1000 Cores," *Computer Architecture News*, vol. 37, no. 2, 2009.

[33] J. Moses *et al.*, "CMPSched$im: Evaluating OS/CMP interaction on shared cache management," in *ISPASS*, 2009.

[34] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.

[35] N. Binkert, et al., "The GEM5 Simulator," *Computer Architecture News*, vol. 39, no. 2, 2011.

[36] U. Nawathe *et al.*, "An 8-core 64-thread 64b power-efficient SPARC SoC," in *ISSCC*, 2007.

[37] B. W. O'Krafka and A. R. Newton, "An Empirical Evaluation of Two Memory-Efficient Directory Methods," in *ISCA*, 1990.

[38] P. M. Ortego and P. Sack, "SESC: SuperESCalar Simulator," UIUC, Tech. Rep., 2004.

[39] P. Rosenfeld et al, "DRAMSim2," http://www.ece.umd.edu/dramsim/.

[40] H. Pan, K. Asanović, R. Cohn, and C. K. Luk, "Controlling Program Execution through Binary Instrumentation," *Computer Architecture News*, vol. 33, no. 5, 2005.

[41] J. T. Pawlowski, "Hybrid Memory Cube (HMC)," in *Hot Chips*, 2011.

[42] K. K. Rangan, G.-Y. Wei, and D. Brooks, "Thread motion: Fine-grained power management for multi-core systems," in *ISCA*, 2009.

[43] S. Rixner *et al.*, "Memory Access Scheduling," in *ISCA*, 2000.

[44] K. Van Craeynest *et al.*, "Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE)," in *ISCA*, 2012.

[45] T. F. Wenisch *et al.*, "SimFlex: Statistical Sampling of Computer System Simulation." *IEEE Micro*, vol. 26, no. 4, 2006.

[46] D. Wentzlaff *et al.*, "On-Chip Interconnection Architecture of the Tile Processor," *IEEE Micro*, vol. 27, no. 5, 2007.

[47] S. C. Woo *et al.*, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *ISCA*, 1995.

[48] M. T. Yourst, "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator," in *ISPASS*, 2007.