Scalable Multi-Core Simulation Using Parallel Dynamic Binary Translation

Oscar Almer*, Igor Böhm*, Tobias Edler von Koch*, Björn Franke*, Stephen Kyle*, Volker Seeker†, Christopher Thompson*, and Nigel Topham*
* Institute for Computing Systems Architecture, University of Edinburgh, United Kingdom
† Software Engineering for Embedded Systems Group, Technical University of Berlin, Germany

***Abstract*—In recent years multi-core processors have seen broad adoption in application domains ranging from embedded systems through general-purpose computing to large-scale data centres. Simulation technology for multi-core systems, however, lags behind and does not provide the simulation speed required to effectively support design space exploration and parallel software development. While state-of-the-art instruction set simulators (ISS) for single-core machines reach or exceed the performance levels of speed-optimised silicon implementations of embedded processors, the same does not hold for multi-core simulators where large performance penalties are to be paid. In this paper we develop a fast and scalable simulation methodology for multi-core platforms based on parallel and just-in-time (JIT) dynamic binary translation (DBT). Our approach can model large-scale multi-core configurations, does not rely on prior profiling, instrumentation, or compilation, and works for all binaries targeting a state-of-the-art embedded multi-core platform implementing the ARCompact instruction set architecture (ISA). We have evaluated our parallel simulation methodology against the industry standard SPLASH-2 and EEMBC MULTIBENCH benchmarks and demonstrate simulation speeds up to 25,307 MIPS on a 32-core x86 host machine for as many as 2048 target processors whilst exhibiting minimal and near constant overhead.**

I. INTRODUCTION

With the proliferation of multi-core processor implementations in virtually all computing domains ranging from embedded systems through general-purpose desktop machines to large-scale data centres, the need is growing for scalable and multi-core enabled instruction set simulators (ISS). Software and hardware developers alike rely on efficient ISS technology for design space exploration, convenient software development in the early design stages of a new system, hardware and software performance optimisation, and debugging. Unfortunately, development of scalable multi-core ISS technology has not kept pace with the rapid advances in multi-core architecture design. Ever-increasing numbers of cores contained in future computing systems challenge today's simulation technology and pose serious performance and scalability bottlenecks.

Unlike multi-core simulators, simulation technology for single-processor systems is mature and provides high simulation rates for complex, superscalar, out-of-order architectures [1]. For some embedded processors, simulators even exceed the native execution performance of speed-optimised silicon implementations of the target processor [2] whilst, at the same time, providing architectural observability. These remarkable performance levels for single-core instruction set simulation have been enabled by Just-in-Time (JIT) Dynamic Binary Translators (DBT), which rely on advanced compiling techniques for efficient on-the-fly translation of binary code, and powerful simulation hosts providing the necessary CPU performance for efficient native code execution. However, due to its inherent complexity, up until now JIT DBT technology has not been available for multi-core simulation.

In this paper we develop novel technology for execution-driven, JIT DBT based multi-core simulation enabling faster-than-FPGA simulation speeds [3]. Our approach scales up to 2048 target processors whilst exhibiting minimal and near constant overhead. We extend single-core JIT DBT to enable multi-core simulation and effectively exploit the parallelism offered by the simulation host for both parallel JIT compilation *and* native execution. In fact, multi-core JIT DBT is not just a single method, but a *combination* of several techniques. It is a widely held view that "we need a 1,000 to 10,000 MIPS simulator to support effective hardware and software research of a 1,000-way multiprocessor system" [4]. In this paper, we present the first-ever simulator – implemented entirely in software – to reach and exceed this performance level.

Our main contribution is to demonstrate how to effectively apply JIT DBT in the context of multi-core target platforms. The key idea is to model each simulated processor core in a separate thread, each of which feeds work items for native code translation to a parallel JIT compilation task farm shared among all CPU threads. Combined with private first-level caches and a shared second-level cache for recently translated and executed native code, detection and elimination of duplicate work items in the translation work queue, and an efficient low-level implementation for atomic exchange operations we construct a highly scalable multi-core simulator that provides faster-than-FPGA simulation speeds and scales favourably up to 2048 simulated cores.

We have evaluated our simulation methodology against the MULTIBENCH and SPLASH-2 benchmark suites. Our functional ISS models homogeneous multi-core configurations composed of ENCORE [5] cores, which implement the ARCOMPACT ISA [6]. On a 32-core x86 host machine we demonstrate simulation rates up to 25,307 MIPS for as many as 2048 target processors. Across all benchmarks our JIT DBT simulation approach achieves an average simulation performance of 11,797 MIPS (for 64 simulated cores) and outperforms an equivalent system implemented in FPGA.

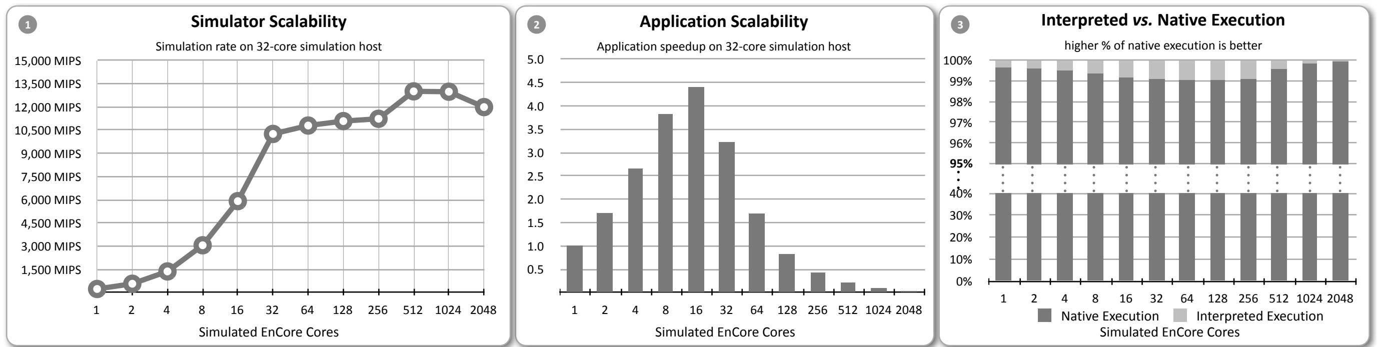

Fig. 1. Full-system simulation of SPLASH-2 LU benchmark - comparison of ① simulation rate in MIPS, ② speedup over single core simulation, and ③ interpreted *vs.* natively executed instructions in %, for the simulation of 1 to 2048 ENCORE cores on a 32-core x86 simulation host.

A. Motivating Example

Consider the full-system simulation of the SPLASH-2 LU program built for the ENCORE processor. On a 32-core Intel Xeon machine we simulate the application with varying numbers of simulated cores, where each simulated core is modelled as a thread. The LU program is a parallel benchmark that factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense $n \times n$ matrix A is divided into an $N \times N$ array of $B \times B$ blocks ($n = NB$) to exploit temporal locality on submatrix elements [7].

Our trace based dynamic binary translator speeds up the simulation by identifying and translating hot traces to native x86 code during simulation using the sequence of steps illustrated in Box ①, Figure 3. Simulated target cores perform simulation as well as hot trace discovery and dispatch in parallel to each other. A light-weight decoupled JIT DBT task farm runs in parallel to this simulation (see Box ⑥, Figure 3) compiling dispatched hot traces to speed up simulation. To ensure that the most profitable traces are compiled first, we use a work scheduling strategy that dynamically prioritises compilation tasks according to their heat and recency [8].

For the purpose of this motivating example we look at the scalability of our ISS with increasing numbers of simulated target cores. In Chart ① of Figure 1 we compare the overall simulation rate in MIPS starting with a single-core simulation and scaling up to a multi-core configuration comprising a total of 2048 target cores. Our simulator scales almost perfectly – *doubling* the simulation rate each time the number of simulated processors is doubled – until the number of simulated target cores equals simulation host cores. Simulating 32 ENCORE cores on a 32-core x86 machine results in an overall simulation rate of **10,500** MIPS for the LU benchmark. Scaling the number of simulated target cores beyond 32 results in a more modest improvement between 10,800 MIPS for 64 simulated cores and the maximum simulation rate of **13,000** MIPS for 1024 cores. For a total of 2048 simulated cores we still achieve a simulation rate of 11,982 MIPS.

Application speedup over single-core simulation is shown in Chart ② of Figure 1. The maximum speedup of 4.4× is reached when simulating 16 target cores and even with 64 simulated cores the application shows a speedup of 1.7×.

Chart ③ of Figure 1 illustrates the ratio of interpreted to natively executed instructions, demonstrating the performance of our parallel JIT DBT. For all configurations up to 2048 simulated target cores the JIT DBT achieves **>99.1%** natively executed instructions.

B. Contributions

Among the contributions of this paper are:

1) The development of a scalable multi-core instruction set simulation methodology extending established single-core JIT DBT approaches to effectively exploit the available hardware parallelism of the simulation host,
2) the use of an innovative parallel task farm strategy to achieve truly concurrent JIT compilation of hot traces,
3) the combination of a multi-level cache hierarchy for JIT-compiled code, detection and elimination of duplicate work items in the translation work queue, and an efficient low-level implementation for atomic exchange operations, and
4) an extensive evaluation of our LLVM-based DBT targeting a multi-core platform implementing the ARCOMPACT ISA and using two benchmark suites: EEMBC MULTIBENCH and SPLASH-2.

Currently, ARCSIM supports cycle-accurate simulation of each processor core [2]. However, faithful performance modelling of the overall multi-core system including its memory hierarchy and interconnect is beyond the scope of this paper and subject of our future work.

C. Overview

The remainder of this paper is structured as follows. In section II we present details of the ENCORE processor and provide an overview of the target system architecture. This is followed by an extensive coverage of our proposed parallel simulation methodology in section III. In section IV we present the results of our empirical evaluation before we discuss related work in section V. Finally, we summarise our findings and give an outlook to our future work in section VI.

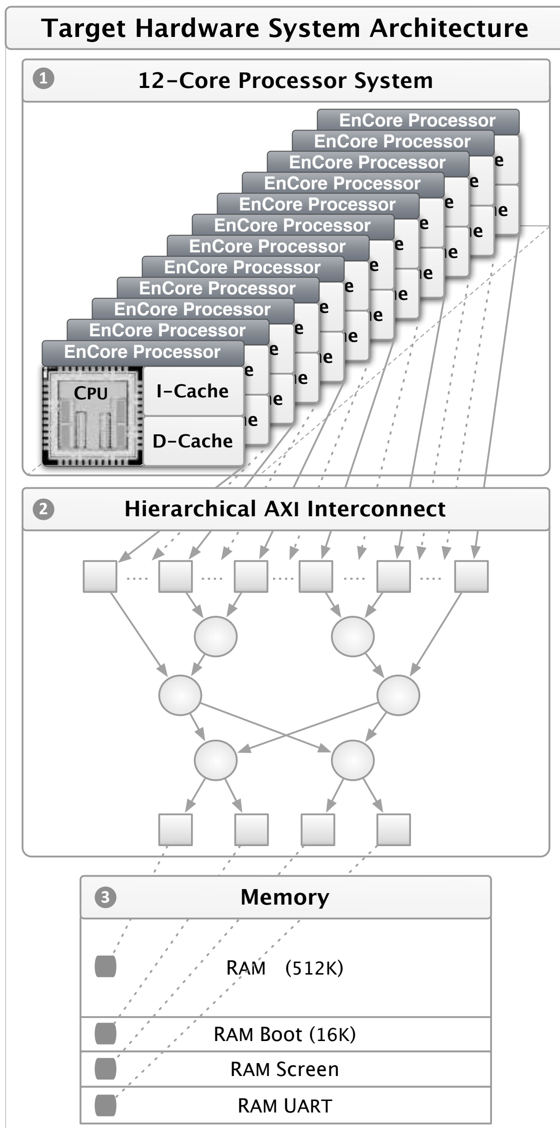

Fig. 2. Multi-core hardware architecture of target system.

II. Background

A. EnCore Processor

Our simulator models a state-of-the-art embedded processor implementing the ARCompact ISA, namely the EnCore [5]. Its micro-architecture is based on a 5-stage interlocked pipeline with forwarding logic, supporting zero overhead loops, freely intermixable 16- and 32-bit instruction encodings, static branch prediction, branch delay slots, and predicated instructions.

Although the above configuration was used for this work, the processor is highly configurable. Pipeline depth, cache sizes, associativity, and block replacement policies as well as byte order (i.e. big endian, little endian), bus widths, register-file size, and instruction set specific options such as instruction set extensions are configurable.

B. Target System Architecture

As a physical reference system we used a 12-core implementation of the multi-core system, synthesised for a Xilinx X6VLX240T FPGA. The system architecture is shown in Figure 2. The twelve processor cores (① in Figure 2) are connected through a 32-bit hierarchical, switched, non-buffered AXI interconnect fabric (② in Figure 2) to RAM and I/O devices (③ in Figure 2). An ASIP implementation of the same EnCore processor, implemented in a generic 90 nm technology node, is currently running in our laboratory at frequencies up to 600 MHz. The processor cores can attain a 50 MHz core clock using this FPGA fabric, while the interconnect is clocked asynchronously to the cores at 75 MHz.

JTAG accessible utility functions and event counters were inserted to be able to record data from the cores. Recorded data for each core includes total clock cycles when not halted, total committed instructions, total I/O operations, and total clock cycles spent on I/O operations. From these counters we calculate the MIPS of each core at 50 MHz (FPGA) and 600 MHz (ASIP), respectively.

C. Just-in-Time Dynamic Binary Translation

Efficient DBT relies heavily on Just-in-Time (JIT) compilation for the translation of target machine instructions to host machine instructions. Although JIT compiled code generally runs much faster than interpreted code, JIT compilation incurs an additional overhead. For this reason, only the most frequently executed code regions are translated to native code whereas less frequently executed code is still interpreted (see Box ① in Figure 3). In a single-threaded execution model, the interpreter pauses until the JIT compiler has translated its assigned code block and the generated native code is executed directly. However, it has been noted earlier [9] that program execution does not need to be paused to permit compilation, as a JIT compiler can operate in a separate thread while the program executes concurrently. This *decoupled* or *asynchronous* execution of the JIT compiler increases complexity of the DBT, but is very effective in hiding the compilation latency – especially if the JIT compiler can run on a separate processor. In section III-B we extend this concept and introduce a *parallel* JIT task farm (see Box ⑥ in Figure 3) to further reduce compilation overhead and cope with the increased pressure on the JIT compiler.

III. Methodology

In this section, we discuss the implementation of the ArcSim simulator, highlighting a number of key contributions which lead to high performance in a multi-core context. These include a carefully designed software architecture, sharing of translations between cores to benefit data-parallel applications, lightweight multi-threading support and an efficient mapping of atomic exchange operations to benefit synchronisation.

A. Simulator Architecture

Figure 3 shows the architectural design of our simulator. Every simulated core runs within its own interpreter as an

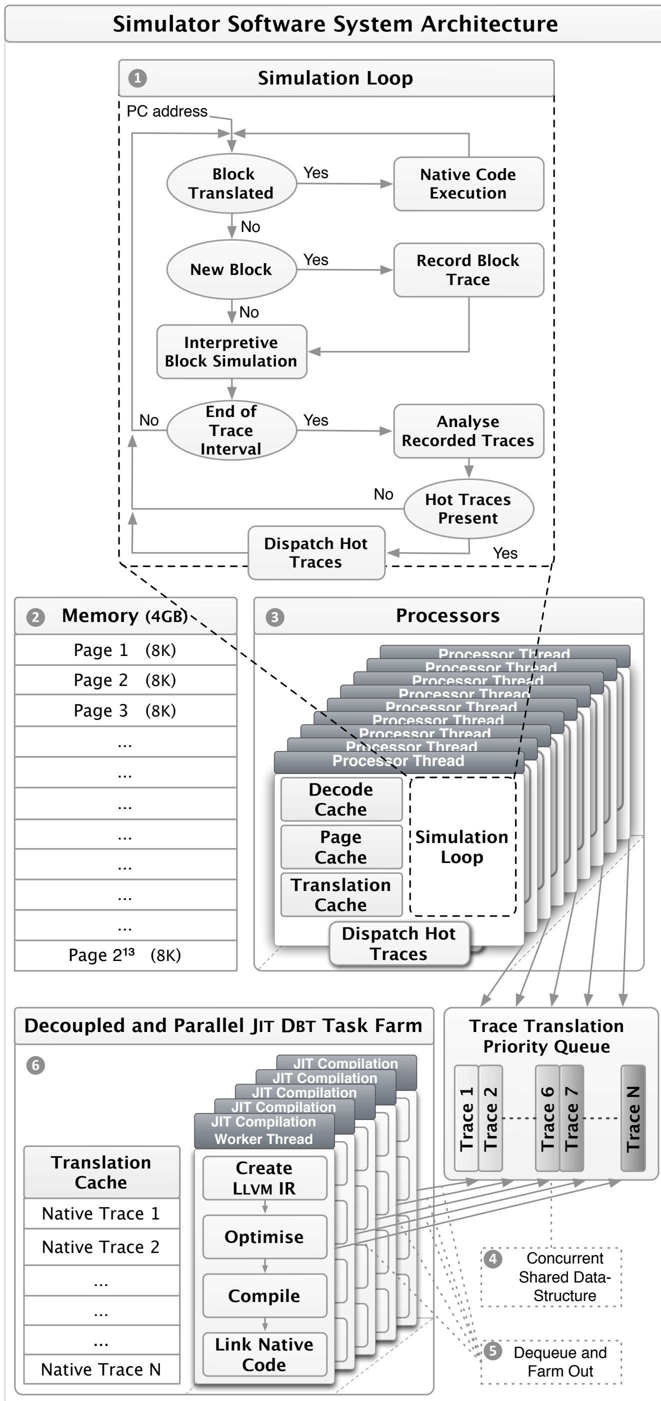

Fig. 3. Software architecture of our multi-core simulation capable ISS using parallel trace-based JIT DBT.

individual thread with its own set of CPU state variables. The operation of the interpreter is shown in Box ①, Figure 3. As this is a functional simulation and is not yet required to be cycle-accurate, these threads are then allowed to simulate the program without restraint. The host operating system is relied upon for scheduling of the cores if more cores are simulated than are available on the host system. Simulated devices, such as screen and terminal I/O, are run in separate threads from the main simulation. Finally, JIT compilation workers also execute in separate threads in a task farm approach where all workers obtain work from a single queue, which all cores dispatch tasks to – see Box ⑥, Figure 3. The operation of the JIT workers is discussed further in the following paragraph.

B. Parallel Just-in-Time Dynamic Binary Translation

Our method for JIT DBT begins with the recording of hot traces as the binary is interpreted. Interpreted simulation time is partitioned into *trace intervals*. After each trace interval every core dispatches the hottest recorded traces, packaged as work units, to a priority queue for JIT compilation before its simulation loop continues. Decoupled from these simulation loops, JIT compilation workers dequeue and compile the dispatched work units, as shown in Box ⑤, Figure 3. A priority queue, shown in Box ④, Figure 3, is used to ensure that the most important traces are scheduled for compilation first. Important traces are determined from a combination of a trace's execution frequency (=*heat*) and time since last execution (=*recency*). The main benefit of this approach is that compilation latency is hidden by performing it in parallel with the main simulation loops of the cores. Trace granularity can be adjusted from the level of single basic blocks to as large as an 8 KB page of memory. We use a light-weight tracing scheme and only record basic block entry points as nodes, and pairs of source and target entry points as edges to construct a CFG.

C. Caching of Translated Code

Data parallel applications may run the same code across different cores. To avoid repeated translation of identical traces we employ a multi-level *translation cache* in our simulator.

As translations are generated for traces by the workers, they must be registered with their associated entry points within the simulator's internal representation of the simulated code. When the simulation reaches any basic block entry point, a lookup must be made to see if a translation exists for this physical address, returning a function pointer which can be called to execute this section of code natively. A direct-mapped *translation cache* was implemented which associates physical addresses with their translations, to avoid costly lookup. This cache is checked before consulting the simulator's internal representation, improving performance within the critical path of simulation. Each core has its own private *translation cache*, shown within Box ③, Figure 3.

This cache also inspired a method for sharing translations between cores. By adding a second level *translation cache*, shown within Box ⑥, Figure 3, which only the JIT compilation workers can access, the workers can check this cache when handling a work unit. Every time a JIT worker completes a task, it loads the generated translations into the second level *translation cache*. Whereas for the private first level cache the JIT worker had to always generate the code itself to be able to register the translation with the requesting core, the worker can now check the second level cache to see if any other

workers have recently produced this translation. If it finds the translation in the cache, it can immediately register it with the requesting core, without having to generate the translation itself, saving time.

Depending on the chosen granularity of the translation units, determining if a translation is in the cache may be more complicated than only checking if the physical addresses match. When our translation units are page-sized, two cores may trace different paths through the page from the same initial physical address, and in this case it would be incorrect to try and use the translation of the first trace to simulate the second trace. For this reason, we also associate a signature with each work unit. The generation of this signature will be discussed in Section III-D. For now, it is enough to say that if the physical address and signature of a JIT task matches those of a translation in the second level cache, it is possible to reuse that translation.

D. Detection and Elimination of Duplicate Work Items

The architecture of our multi-core JIT compilation system presents another opportunity to prevent redundant translation work. As multiple cores add tasks to the queue from which JIT workers receive their work, it would be beneficial to identify when a core attempts to dispatch a task which has already been added by another core. Once identified, we can prevent the task from being added to the queue, and instead register that the requesting core would also like to be updated once the queued task has been handled by a worker.

In order to identify whether two tasks are for the same trace, we generate signatures for each task as it is created. The signature is the result of a hash function applied to the physical addresses of all the basic blocks that are to be translated. This provides us with a means to quickly determine if traces starting at the same address are different. While this may lead to false negatives, the error rate is diminished and is negligible in practice. The hash function is simple in our implementation to maximise performance, but any hash function which accepts 32-bit integers with more robust collision-avoidance could replace it.

A hash table stores lists of all tasks that have a particular key as their signature. Upon attempting to dispatch a new task to the queue, the table is checked for the task's signature. If found, the task is considered a duplicate, and is added to the corresponding list. If not, the task is new, and a new list containing it is added to the hash table, while the task is additionally added to the queue. The JIT compilation workers continue to take tasks from the queue as previously described, and, upon completing a task, check the hash table to see if any other tasks were added to the JIT system while this one was waiting or being processed. If any exist, the cores which created these tasks are also notified about the generated translations.

This technique, in addition to the shared caching of translations described in Section III-C, has the dual effect of reducing the waiting period between the dispatch of a task and the receipt of its translation for many cores. It also reduces the amount of similar tasks in the work queue, resulting in a greater percentage of the simulated code being translated earlier.

E. Atomic Exchange Operations

The atomic exchange instructions in the ARCOMPACT instruction set are the way in which the ISA exposes explicit multi-core synchronisation. In ARCSIM, this is implemented using a global lock for all atomic exchange instructions. Whilst potentially expensive, this enables the use of the underlying `x86` hardware synchronisation instructions to directly implement the atomic exchange instruction; in practice, it maps to a single `x86` instruction.

IV. EMPIRICAL EVALUATION

We have evaluated our parallel JIT DBT multi-core simulator on over 20 benchmarks from the MULTIBENCH and SPLASH-2 benchmark suites. In this section we describe our experimental approach and present and evaluate our results for the speed and scalability of our simulator.

A. Experimental Setup and Methodology

We have evaluated our simulator using the MULTIBENCH 1.0 benchmark suite [10], which comprises a total of 14 application kernels from the networking and consumer domains which can be combined in various ways to reflect typical application workloads of embedded systems. We run each application kernel separately with a varying number of worker threads. Some of the kernels are not multi-threaded, so in these cases we run a separate instance of the kernel on each of the simulated cores.

The SPLASH-2 benchmark suite [7] comprises 12 benchmarks which cover a number of common complex calculations in areas such as linear algebra, complex fluid dynamics, and graphics rendering. Each benchmark is designed to partition its work using threads[1].

Our results are reported in terms of MIPS achieved by the simulator. Each core can calculate its own MIPS rate, where the number of instructions the core has executed is divided by the length of time between when the core itself starts to execute instructions, and when the core is halted. These individual MIPS rates are summed to provide the total simulation rate. Each benchmark and core configuration was run 5 times, with the arithmetic mean taken from these runs to present our results.

The system used to run the simulator was a `x86` host with 4 Intel Xeon L7555 1.87 GHz (8-core) processors with hyper-threading disabled, resulting in 32 cores being made available to the openSUSE 11.3 Linux operating system. The system also had 64GB of RAM available, and all experiments were run under conditions of low system load.

[1] We were unable to build versions of `fmm` and `water-nsquared` which could run on 64 cores, so these are excluded from our results. In the case where contiguous and non-contiguous versions of the same benchmark were available, we have built the contiguous version.

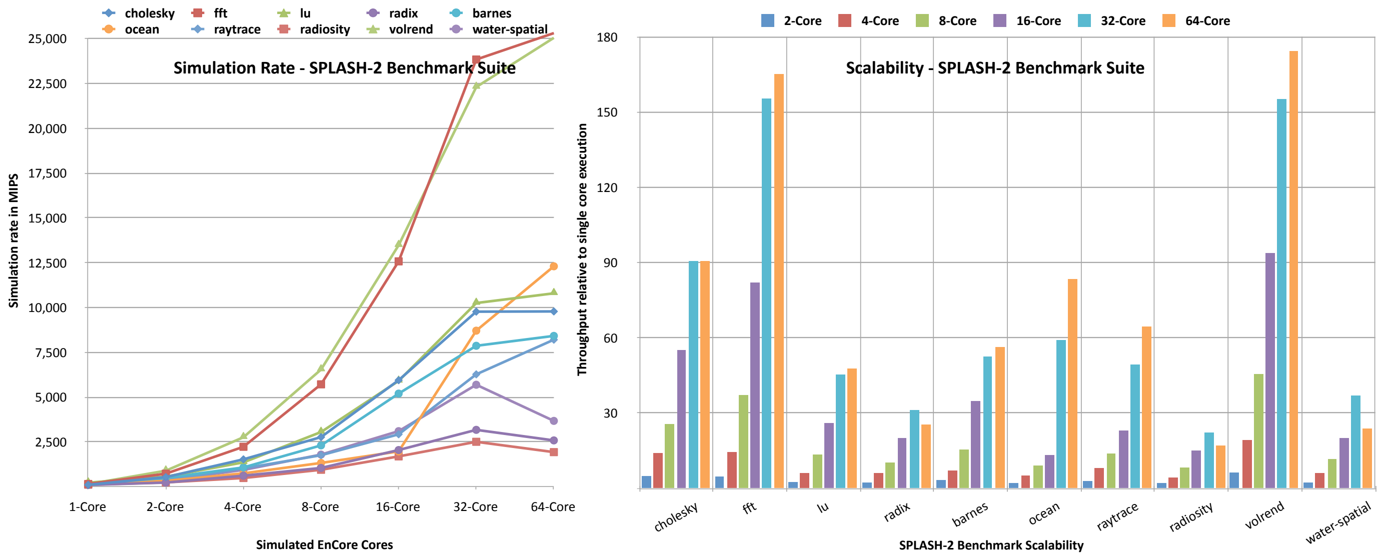

Fig. 4. Simulation rate in MIPS (left chart) and throughput relative to single-core execution (right chart) using the SPLASH-2 benchmark suite for varying multi-core configurations of our ISS.

B. Bare-Metal POSIX *Multi-Threading Support*

We use a light-weight library that provides the essentials of operating system functionality to run the benchmarks on bare-metal hardware and within the simulator. This library provides features such as startup code, I/O device management, memory management primitives, and in particular basic multi-threading support in the form of a `pthreads` API.

We restrict the number of threads to one per core – which is permitted by the `pthreads` specification – since the implementation of a full preemptive scheduler would have exceeded the scope of this work. Mutexes are implemented using spin locks based around the ARCOMPACT atomic exchange instruction. Our implementation also fully supports condition variables, semaphores, and thread joining.

This approach differs from that taken for instance by UNISIM [11] where threading support is emulated by the simulator. This requires applications to be linked against a `pthreads` emulation library which re-routes `pthreads` API calls to trigger simulator intervention such as suspending or waking up of simulated cores. Our approach, in contrast, produces binaries that can be run both on real hardware and in the simulation environment without modification.

C. Summary of Key Results

Our results shown in Figures 4 and 5 demonstrate that the initial target of "1,000 to 10,000 MIPS" [4] was easily attained, with seven of our benchmarks exceeding 20,000 MIPS when simulating a 32-core target. This is better than the performance of a theoretical 600 MHz 32-core ASIP. For the SPLASH-2 `fft` benchmark we achieve a maximum overall simulation rate of 25,307 MIPS for a 64-core simulation target, whilst on average we still provide 11,797 MIPS for the same simulation target. For large-scale configurations of up to 2048 cores our results shown in Figure 6 demonstrate the ability of our simulator to scale with the number of processors and to sustain its simulation rate beyond the point at which the number of simulated cores exceeds those of the host system.

D. Simulation Speed

All of the MULTIBENCH baseline three-core simulations exceed 1,000 MIPS, with `rgbhpg03` reaching 3,100 MIPS. Due to their higher complexity the single-core performance of the SPLASH-2 benchmarks ranges between 100 to 225 MIPS. On the other hand, they exhibit far greater scalability (see Section IV-E).

Simulating 64 target cores we achieve simulation rates in excess of 20,000 MIPS for `fft` and `volrend` from SPLASH-2, and for `md5`, `rgbcmyk`, `mpeg2`, `rgbyiq03`, and `rgbhpg03` from MULTIBENCH. Only 5 out of 24 applications fail to deliver more than 3,200 MIPS (equivalent to 100 MIPS simulation rate per host core) while the average performance across all benchmarks for this configuration is close to 12,000 MIPS.

Not all benchmarks maintain this simulation rate as the number of cores increases, showing that simulation performance is application-specific. For instance, the MULTIBENCH networking benchmarks (`ippktcheck`, `ipres`, `tcp`) show little, if any, improvement over the baseline for higher numbers of simulated cores. The profile of the instructions executed by these benchmarks indicates a very high rate of memory accesses and memory-dependent branches which quickly saturate the available memory bandwidth of the host system. These findings are in line with the data sheets provided by EEMBC [10].

E. Scalability

It is important to evaluate how the simulator scales beyond the number of host processor cores for simulating tomorrow's many-core systems on today's commodity hardware. Most

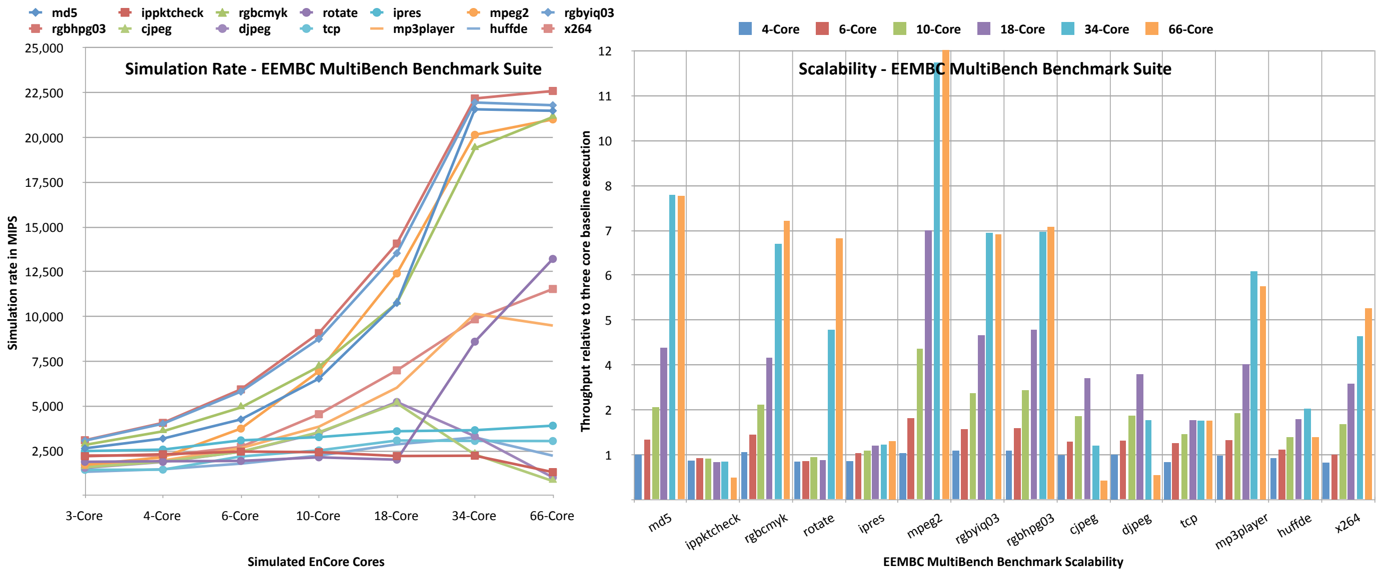

Fig. 5. Simulation rate in MIPS (left chart) and throughput relative to *three*-core execution (right chart) using the EEMBC MULTIBENCH benchmark suite for varying multi-core configurations of our ISS. Note that the minimum MULTIBENCH core configuration is three due to the test harness infrastructure.

benchmarks demonstrate that the simulator scales well up to the number of physical cores on the host. Beyond this point we occasionally see modest further improvements (e.g. `cholesky`, `lu`, and `md5`) as shown in Figure 6.

For the same seven benchmarks that deliver the highest simulation rates, we observe linear scalability as we increase the number of target cores up to 32. Other benchmarks such as `ocean`, `lu`, `cholesky`, `barnes`, `raytrace`, and `x264` do not achieve such high aggregate simulation rates, but still scale favourably.

For six representative benchmarks (three from each benchmark suite) we show scalability up to 2048 simulated target cores in Figure 6. Chart ① in Figure 6 shows the best result, with `cholesky` continuing to scale from 9,767 MIPS for 32 cores, to 17,549 MIPS for 2048 cores, with the performance always increasing. Chart ④ in Figure 6 shows a similar result for `md5`.

In Figure 4 we see super-linear scalability for a number of benchmarks (e.g. `fft`, `ocean`, `volrend`, `cholesky`). This is due to excessive synchronisation in the benchmarks beyond 64 cores, and the fact that our simulator can execute tight spin-lock loops at near native speed. It is a well-known fact that the SPLASH-2 benchmarks attract high synchronisation costs for large-scale hardware configurations, as shown by other research [12]. The MULTIBENCH results are not affected in the same way due to less synchronisation.

F. Comparison to Native Execution on Real Hardware

Chart ① of Figure 7 shows a comparison between our simulator and two hardware platforms (FPGA and ASIP, see Section II-B) in terms of MIPS. The application is a parallelised fractal drawing algorithm, executed across 12 cores. We chose this application because of its low memory footprint and its embarrassingly parallel nature, thus avoiding application scalability issues.

Actual FPGA performance is 249 MIPS. We also show the performance of a 600 MHz ASIP implementation which achieves an execution rate of 2,985 MIPS. On the other hand, our instruction set simulator reaches a simulation rate of 6,752 MIPS, thus surpassing a silicon implementation by more than a factor of 2 for this application.

For equivalent configurations, our simulator consistently outperforms the theoretical maximum of the FPGA on a per-core basis. Our 12-core FPGA implementation of the multi-core system is capable of 50 MIPS per core. On the contrary, across all benchmarks, the lowest per-core simulation rate for a 16-core target was 105 MIPS, attained in the SPLASH-2 `radiosity` benchmark. These results show that even in the worst case the simulator maintains more than twice the theoretical maximum execution rate of the FPGA. Compared to the average simulation rate of 11,797 MIPS across all benchmarks, the theoretical maximum of 600 MIPS for a 12-core FPGA implementation is an order of magnitude slower.

V. RELATED WORK

Due to the wealth of related research work we restrict our discussion to the most relevant approaches in software and FPGA based multi-core simulation.

A. Software Based Multi-Core Simulation Approaches

Early work on efficient simulation of parallel computer systems dates back to the 1990s [13] and examples of conventional sequential simulators/emulators include SIMPLESCALAR [14] , RSIM [15], SIMOS [16], SIMICS [17], and QEMU [18]. Some of these are capable of simulating parallel target architectures but all of them execute sequentially on the host machine.

Over the last two decades a large number of parallel simulators of parallel target architectures have been developed:

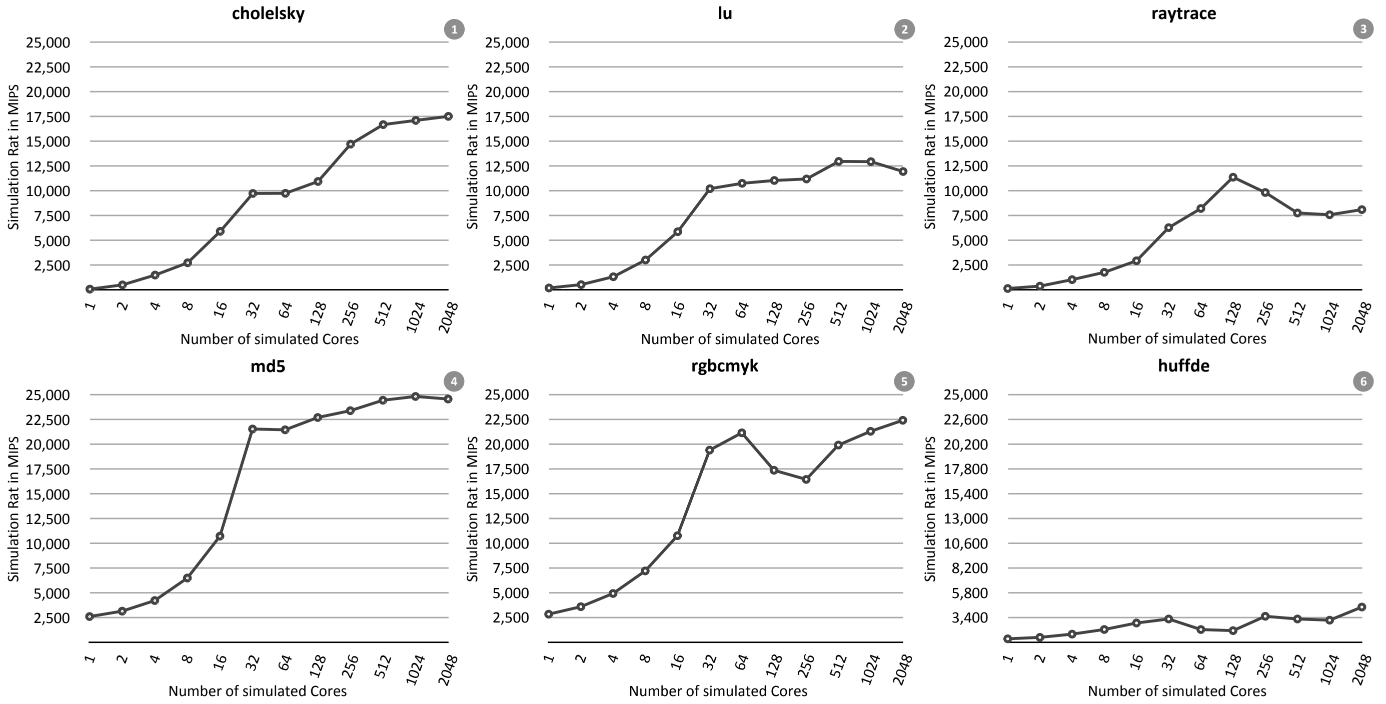

Fig. 6. Results for selected benchmarks from SPLASH-2 ①②③ and EEMBC MULTIBENCH ④⑤⑥ demonstrating the scalability with the number of simulated target cores.

SIMFLEX [19], GEMS [20], COTSON [21], BIGSIM [22], FASTMP [23], SLACKSIM [24], PCASIM [25], Wisconsin Wind Tunnel (WWT) [26], Wisconsin Wind Tunnel II (WWT II) [27], and those described by Chidester and George [28], and Penry et al. [29]. SIMFLEX and GEMS both use an off-the-shelf sequential emulator (SIMICS) for functional modeling plus their own models for memory systems and core interactions. GEMS uses their timing model to drive SIMICS one instruction at a time, but results in low performance. SIMFLEX avoids this problem by using statistical sampling of the application, but therefore does not observe its entire behaviour. COTSON [21] uses AMD's SIMNOW! for functional modeling and suffers from some of the same problems as SIMFLEX [19] and GEMS [20].

BIGSIM [22] and FASTMP [23] assume distributed memory in their target architectures and do not provide coherent shared memory between the parallel portions of their simulators. WWT [26] is one of the earliest parallel simulators but requires applications to use an explicit interface for shared memory. WWT II [27] does not model anything other than the target memory system and requires applications to be modified to explicitly allocate shared memory blocks. ARCSIM also models compute cores and implements a transparent shared memory system.

Like ARCSIM, PARALLEL EMBRA [30] is a fast functional simulator for shared-memory multiprocessors which is part of the PARALLEL SIMOS complete machine simulator. It takes an aggressive approach to parallel simulation; while it runs at user level and does not make use of the MMU hardware, it combines binary translation with loose timing constraints and relies on the underlying shared memory system for event ordering, time synchronisation, and memory synchronisation. While PARALLEL EMBRA shares its use of binary translation with ARCSIM it lacks its scalability and parallel JIT translation facility.

Another effort to parallelise a complete machine software simulator was undertaken with MAMBO [31]. It aims to produce a fast functional simulator by extending a binary translation based emulation mode; published results include a speedup of up to 3.8 for a 4-way parallel simulation. Similarly, the MALSIM [32] parallel functional simulator has only been evaluated for workloads comprising up to 16 threads. Despite some conceptual similarities with these works our work aims at larger multi-core configurations where scalability is a major concern.

In [33] a DBT based simulator targeting a tiled architecture is presented. It aims at implementing different portions of a superscalar processor across distinct parallel elements thus exploiting the pipeline parallelism inherent in a superscalar microarchitecture. However, this work does not attempt to simulate a multi-core target platform.

ARMN [34] is a cycle-accurate simulator for homogeneous platforms comprising several ARM cores and with support for various interconnect topologies. Whilst this provides flexibility, the performance of ARMN is very low (approx. 10*k* instructions per second) and, thus, its suitability for both HW design space exploration and SW development is limited.

The GRAPHITE [35] multi-core simulation infrastructure is most relevant to our work. GRAPHITE is a distributed parallel multi-core simulator that combines direct execution,

seamless multi-core and multi-machine distribution, and lax synchronisation. GRAPHITE has been demonstrated to simulate target architectures containing up to 1024 cores on ten 8-core servers. Application threads are executed under a dynamic binary instrumentor (currently PIN) which rewrites instructions to generate events at key points. These events cause traps into GRAPHITE's backend which contains the compute core, memory, and network modelling modules. In contrast, our simulator uses DBT to implement any ISA (currently ARCOMPACT), which can also be different from the target system's ISA. In addition, the primary design goal of our simulator has been highest simulation throughput as showcased by the parallel JIT task farm comprised in ARCSIM. As a result we achieve speedups over native execution for many multi-core configurations, whereas GRAPHITE suffers up to 4007× slowdown.

In [12] a methodology to simulate shared-memory multiprocessors composed of hundreds of cores is proposed. The basic idea is to use thread-level parallelism in the software system and translate it into core-level parallelism in the simulated world. An existing full-system simulator is first augmented to identify and separate the instruction streams belonging to the different software threads. Then, the simulator dynamically maps each instruction flow to the corresponding core of the target multi-core architecture, taking into account the inherent thread synchronisation of the running applications. This approach treats the functional simulator as a monolithic block, thus requiring an intermediate step for de-interleaving instructions belonging to different application threads. ARCSIM does not require this costly preprocessing step, but its functional simulator explicitly maintains parallel threads for the CPUs of the target system.

Outside the academic community the PCXS2 software simulator [36] for the PS2 games console is a good example of a simulator targeting a multi-core platform which utilises dynamic binary translation.

B. FPGA *Based multi-core Simulation Approaches*

We can generally distinguish between two approaches to FPGA based multi-core simulation: The first approach essentially utilises FPGA technology for *rapid prototyping*, but still relies on a detailed implementation of the target platform (see section II-A), whereas the second approach seeks to speed up performance modelling through a combined implementation of a functional simulator and a *timing model* on the FPGA fabric. In the following paragraphs we discuss approaches to this latter FPGA architecture simulation.

RAMP GOLD [3] is a state-of-the-art FPGA based "many-core simulator" supporting up to 64 cores. In functional-only mode, RAMP GOLD achieves a full simulator throughput of up to 100 MIPS when the number of target cores can cover the functional pipeline depth of 16. For fewer target cores (and non-synthetic workloads), the fraction of peak performance achieved is proportionally lower. In comparison to the 25,307 MIPS peak performance of our software-only simulation approach (based on an ISA of comparable

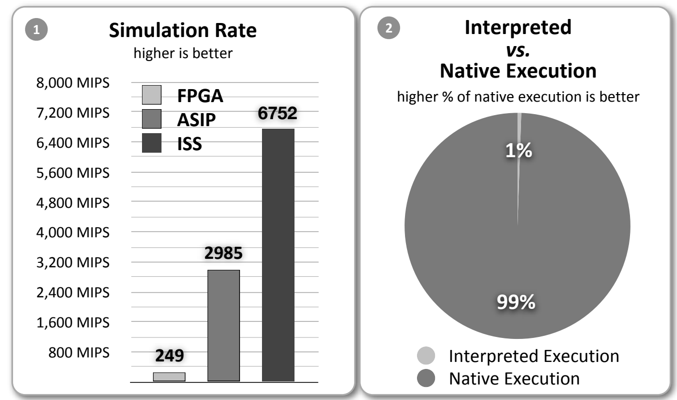

Fig. 7. Comparison of simulator and hardware implementations. ① shows a comparison of maximum achievable simulation rate in MIPS for a 12-core configuration running a parallel Mandelbrot fractal benchmark on an FPGA, ASIP, and ISS platform. ② depicts the ratio of interpreted vs. natively executed instructions on the ISS platform.

complexity and similar functional-only simulation) the performance of the FPGA architecture simulation is more than disappointing. Other approaches to FPGA architecture model execution such as RAMP [37], FAME [38], FAST [39], [40] and PROTOFLEX [41] suffer from the same performance issues and for none of the mentioned systems has scalability beyond 64 cores been demonstrated.

VI. SUMMARY, CONCLUSIONS AND FUTURE WORK

In this paper we have developed an innovative methodology for high-speed multi-core instruction set simulation based on Just-in-Time Dynamic Binary Translation. We have integrated a JIT task farm for parallel translation of hot code traces with a combination of performance-enhancing techniques such as private and shared caching of translated code, detection and elimination of identical translation units in the JIT work queue, and efficient low-level code generation for atomic exchange operations. Through this unique combination of techniques we achieve unprecedented simulator throughput of up to 25,307 MIPS and near-optimal scalability of up to 2048 simulated cores for the SPLASH-2 and EEMBC MULTIBENCH benchmarks on a standard 32-core x86 simulation host. Through our empirical results we demonstrate a simulation performance advantage by two orders of magnitude over leading and state-of-the-art FPGA architecture simulation technology [3] for a comparable level of simulation detail.

In our future work we will extend our cycle-accurate single-core performance model [2] to efficiently capture the detailed behaviour of the shared second level cache, processor interconnect and external memory of the simulated multi-core platform.

REFERENCES

[1] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *Computer*, vol. 35, pp. 59–67, February 2002.

[2] I. Böhm, B. Franke, and N. P. Topham, "Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator," in *ICSAMOS*, F. J. Kurdahi and J. Takala, Eds. IEEE, 2010, pp. 1–10.

[3] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanović, "RAMP gold: an FPGA-based architecture simulator for multiprocessors," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 463–468.

[4] E. S. Chung, E. Nurvitadhi, J. C. Hoe, B. Falsafi, and K. Mai, "PROToFLEX: FPGA-accelerated hybrid functional simulator," *Parallel and Distributed Processing Symposium, International*, vol. 0, p. 326, 2007.

[5] "ENCORE embedded processor." [Online]. Available: http://groups.inf.ed.ac.uk/pasta/hw_encore.html

[6] Synopsys, Inc., "ARCompact instruction set architecture," 700 East Middlefield Road, Mountain View, California 94043, United States. [Online]. Available: http://www.synopsys.com

[7] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd annual international symposium on Computer architecture*, ser. ISCA '95. New York, NY, USA: ACM, 1995, pp. 24–36.

[8] I. Böhm, T. J. Edler von Koch, B. Franke, and N. Topham, "Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, 2011.

[9] J. Aycock, "A brief history of just-in-time," *ACM Comput. Surv.*, vol. 35, pp. 97–113, June 2003.

[10] The Embedded Microprocessor Benchmark Consortium, "MultiBench 1.0 Multicore Benchmark Software," 02 February 2010.

[11] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. A. Penry, O. Temam, and N. Vachharajani, "Unisim: An open simulation environment and library for complex architecture design and collaborative development," *IEEE Comput. Archit. Lett.*, vol. 6, pp. 45–48, July 2007.

[12] M. Monchiero, J. H. Ahn, A. Falcón, D. Ortega, and P. Faraboschi, "How to simulate 1000 cores," *SIGARCH Comput. Archit. News*, vol. 37, pp. 10–19, July 2009.

[13] R. Covington, S. Dwarkada, J. R. Jump, J. B. Sinclair, and S. Madala, "The efficient simulation of parallel computer systems," in *International Journal in Computer Simulation*, 1991, pp. 31–58.

[14] R. Zhong, Y. Zhu, W. Chen, M. Lin, and W.-F. Wong, "An inter-core communication enabled multi-core simulator based on simplescalar," *Advanced Information Networking and Applications Workshops, International Conference on*, vol. 1, pp. 758–763, 2007.

[15] C. Hughes, V. Pai, P. Ranganathan, and S. Adve, "RSIM: Simulating shared-memory multiprocessors with ILP processors," *Computer*, Jan 2002.

[16] R. Lantz, "Parallel SimOS: Scalability and performance for large system simulation," *www-cs.stanford.edu*, Jan 2007.

[17] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, pp. 50–58, February 2002.

[18] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. of the 2005 USENIX Annual Technical Conference.*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41.

[19] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "SimFlex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, pp. 31–34, March 2004.

[20] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News*, vol. 33, pp. 92–99, November 2005.

[21] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "COTSon: infrastructure for full system simulation," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 52–61, January 2009.

[22] G. Zheng, G. Kakulapati, and L. V. Kalé, "BigSim: A parallel simulator for performance prediction of extremely large parallel machines," *Parallel and Distributed Processing Symposium, International*, vol. 1, p. 78b, 2004.

[23] S. Kanaujia, I. E. Papazian, J. Chamberlain, and J. Baxter, "FastMP: A multi-core simulation methodology," in *Proceedings of the Workshop on Modeling, Benchmarking and Simulation (MoBS 2006)*, Boston, Massachusetts, 2006.

[24] J. Chen, M. Annavaram, and M. Dubois, "SlackSim: a platform for parallel simulations of CMPs on CMPs," *SIGARCH Comput. Archit. News*, vol. 37, pp. 20–29, July 2009.

[25] X. Zhu, J. Wu, X. Sui, W. Yin, Q. Wang, and Z. Gong, "PCAsim: A parallel cycle accurate simulation platform for CMPs," in *Proceedings of the 2010 International Conference on Computer Design and Applications (ICCDA)*, June 2010, pp. V1–597 – V1–601.

[26] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood, "The wisconsin wind tunnel: virtual prototyping of parallel computers," in *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '93. New York, NY, USA: ACM, 1993, pp. 48–60.

[27] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, M. D. Hill, D. A. Wood, S. Huss-Lederman, and J. R. Larus, "Wisconsin Wind Tunnel II: A fast, portable parallel architecture simulator," *IEEE Concurrency*, vol. 8, pp. 12–20, October 2000.

[28] M. Chidester and A. George, "Parallel simulation of chip-multiprocessor architectures," *ACM Trans. Model. Comput. Simul.*, vol. 12, pp. 176–200, July 2002.

[29] D. A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. I. August, and D. Connors, "Exploiting parallelism and structure to accelerate the simulation of chip multi-processors," in *in Proc. of the Twelfth Int. Symp. on High-Performance Computer Architecture*, 2006, pp. 29–40.

[30] R. Lantz, "Fast functional simulation with parallel Embra," in *Proceedings of the 4th Annual Workshop on Modeling, Benchmarking and Simulation*, 2008.

[31] K. Wang, Y. Zhang, H. Wang, and X. Shen, "Parallelization of IBM Mambo system simulator in functional modes," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 1, pp. 71–76, 2008.

[32] X. Sui, J. Wu, W. Yin, D. Zhou, and Z. Gong, "MALsim: A functional-level parallel simulation platform for CMPs," in *2nd International Conference on Computer Engineering and Technology (ICCET), 2010*, vol. 2. IEEE, Jan 2010, p. V2.

[33] D. Wentzlaff and A. Agarwal, "Constructing virtual architectures on a tiled processor," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 173–184.

[34] X. Zhu and S. Malik, "Using a communication architecture specification in an application-driven retargetable prototyping platform for multiprocessing," in *Proceedings of the conference on Design, automation and test in Europe - Volume 2*, ser. DATE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 21 244–.

[35] J. E. M. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *The 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, January 2010.

[36] "PCSX2." [Online]. Available: http://pcsx2.net/

[37] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic, "RAMP: Research Accelerator for Multiple Processors," *IEEE Micro*, vol. 27, pp. 46–57, March 2007.

[38] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson, "A case for FAME: FPGA architecture model execution," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 290–301.

[39] D. Chiou, D. Sunwoo, H. Angepat, J. Kim, N. Patil, W. Reinhart, and D. Johnson, "Parallelizing computer system simulators," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1 –5.

[40] D. Chiou, H. Angepat, N. Patil, and D. Sunwoo, "Accurate functional-first multicore simulators," *IEEE Comput. Archit. Lett.*, vol. 8, pp. 64–67, July 2009.

[41] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi, "ProtoFlex: Towards scalable, full-system multiprocessor simulations using FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, pp. 15:1–15:32, June 2009.