Fast and Cycle-Accurate Modeling of a Multicore Processor

Asif Khan, Muralidaran Vijayaraghavan, Silas Boyd-Wickizer and Arvind

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology, Cambridge, MA
{aik,vmurali,sbw,arvind}@csail.mit.edu

***Abstract*—An ideal simulator allows an architect to swiftly explore design alternatives and accurately determine their impact on performance. Design exploration requires simulators to be easily modifiable, and accurate performance estimates require detailed models. Unfortunately, detailed modeling not only impacts the ease with which a simulator can be modified, but also the speed at which it can be executed, resulting in fidelity being traded for simulation speed. Although FPGA-based simulators have dramatically higher speed than software simulators, sacrificing fidelity is still common.**

In this paper we present Arete, an FPGA-based processor simulator, which offers high performance along with accuracy and modifiability. We begin with a cycle-level specification of a multicore architecture which includes realistic in-order cores and detailed models of shared, coherent memory and on-chip network. We then describe how this specification is implemented faithfully and efficiently on FPGAs. Arete delivers a performance of up to 11 MIPS per core. We run a subset of the PARSEC benchmark suite on top of off-the-shelf SMP Linux, and achieve an average performance of 55 MIPS for an 8-core model. We also describe two significant architectural explorations: one involving three different branch predictors and the other requiring major modifications to the cache-coherence protocol.

I. Introduction

Performance modeling plays a critical role in the design and development of microprocessors. There is an ever-rising need for fast, accurate and flexible simulators to explore new architectural ideas and evaluate their impact on performance. Unfortunately, as processor architectures get more complex, it becomes more difficult to implement processor simulators which are both accurate and have high performance. The result is that simulators often sacrifice accuracy or performance, giving users the choice between an accurate simulator that takes days to execute a workload, and a high performance simulator that does not accurately reflect real hardware. The move towards complicated multicore architectures with shared memory and on-chip network architectures has only exacerbated this problem.

Over the last few years, the availability of large FPGAs and new high-level synthesis tools has provided a new opportunity for cycle-accurate simulations. It is now possible to do cycle-accurate simulations of realistic designs, *e.g.*, a small number of out-of-order processors, on a single FPGA chip. These FPGA-based cycle-accurate simulators are able to provide three-orders of magnitude improvement in performance over software simulators [1]–[4]. The initial effort to develop such FPGA simulators is somewhat greater than that required for software simulators, but it still is a far cry from the effort needed to develop a processor chip. Also, it is possible to design these FPGA simulators in such a way that they are amenable to modular refinement, and facilitate the generation of simulators for many different variants of a base architecture.

In this paper we present Arete, an FPGA-based cycle-accurate simulator for a multicore PowerPC architecture. We developed this simulator adhering to a cycle-level specification of the architecture. For the purpose of efficient FPGA implementation we used the LI-BDN technique [5] which helps to improve the FPGA cycle time and to reduce the FPGA resource requirements by using multiple FPGA cycles to simulate one cycle of the target architecture. We boot off-the-shelf SMP Linux and run applications such as the PARSEC benchmark suite [6] on Arete. Our simulator is also suitable for architectural exploration. We demonstrate this by evaluating three different branch prediction schemes and by extending the cache-coherence scheme to provide software with better control over the contents of the caches. We have ported Arete to two single-FPGA platforms (XUPv5 and ML605) and one multi-FPGA platform (BEE3).

To our knowledge Arete is the first cycle-accurate FPGA-based multicore processor simulator which includes both a realistic core architecture and a detailed cache-coherence engine. Along with modeling this level of detail, Arete delivers high performance, viz, 55 MIPS while simulating eight cores on four FPGAs and up to 11 MIPS while simulating one core on one FPGA.

Paper organization: Section II discusses cycle-accurate modeling along with target simplifications and implementation refinements. Section III describes the use of the LI-BDN technique for implementing models on FPGAs. Section IV provides a detailed description of Arete. Section V discusses some of the architectural exploration that we have conducted on Arete, and provides statistics on its performance and resource utilization. Section VI discusses some of the related work in the areas of multicore processor modeling and the use of FPGAs for implementing these processor models. Section VII summarizes our work and discusses some of the future avenues we are planning to explore.

978-1-4673-1146-5/12/$31.00 ©2012 IEEE

II. Cycle-accurate simulation, simplifications and refinements

The term "cycle-accurate simulation" is used in literature to characterize many different types of simulations. In this paper we define it as a simulation that conforms to the cycle-by-cycle behavior of the target design. The behavior may be characterized in terms of the values of all the state elements of a machine (registers, memories, etc.) for every clock cycle. Sometimes it is sufficient to consider only a subset of the state elements, *e.g.*, the PC and the Register File, and ignore others, *e.g.*, the pipeline registers inside a multiplier.

Cycle-accurate simulators tend to be both slow and complex. To overcome these obstacles, architects often simplify the target design. For instance, when one wants to only study the on-chip network and the memory subsystem of a multicore processor, the details in the architecture of the core will typically be omitted – the core will be simplified to magically execute one instruction every clock cycle, while the on-chip network and the memory subsystem will be modeled very accurately. We use the term *target simplifications* in order to describe such simplifications of the target design. The performance estimates obtained from a simulator that uses target simplifications may not be fully accurate; the ability to choose a target simplification that does not skew performance estimates comes mainly from experience.

Once the cycle-by-cycle behavior of a model (which may include target simplifications) has been specified, the specification can be transformed into a netlist. This netlist can be used to program an FPGA, but it may require too many FPGA resources or present an unacceptably long critical path. In order to reduce the resource requirements and shorten the critical path, an implementation may use several FPGA cycles to simulate one model cycle while preserving model timing accuracy.

As an example, consider a model of a target ALU design which includes a single-cycle multiplier. When this model is implemented on FPGA, we may choose to replace the single-cycle multiplier with a 4-cycle unpipelined multiplier in order to improve the FPGA clock speed and to reduce the resource requirements. This replacement may be achieved through a target simplification where the ALU specification is changed to accommodate a 4-cycle multiplier. The overall performance of the modified ALU may not differ much from that of the original ALU as the multiplier may be used infrequently.

Another way to replace the single-cycle multiplier with a 4-cycle multiplier is to change the implementation of the model in such a way that when the 4-cycle multiplication takes place, the rest of the model remains frozen. We call such a 4-cycle multiplier an *implementation refinement* of the single-cycle multiplier. Implementation refinements enable an efficient implementation of the model while preserving its cycle-level behavior.

We maintain a clear distinction between target simplifications and implementation refinements, and do not simplify the target specification to meet FPGA resource constraints.

III. Implementation Methodology

In the last few years many FPGA-based simulators have been built, and consequently, a number of techniques have been developed to make efficient use of FPGA resources without compromising the functionality or the timing of the model [3]–[5], [7]. All these techniques trade time for resources, *i.e.*, permit several FPGA cycles to simulate the behavior of one model cycle, while reducing the resource consumption of the model. We employ the LI-BDN [5] technique to implement our model on FPGA because it enables the use of implementation refinements while preserving the cycle-accuracy of the model and guaranteeing the absence of deadlocks from the implementation. Moreover, it does not force the use of any target simplifications.

A. Overview of the LI-BDN technique

We give a brief overview of the LI-BDN technique using the example in Figure 1. We start with a cycle-level specification of a module which has 3 input ports, 2 output ports, some state, and some logic, as shown in Figure 1(a). We transform the specification into an LI-BDN by first attaching FIFOs to all the ports and done flags to all the output ports, as shown in Figure 1(b). Now, as Figure 1(c) depicts, the top output depends only on the top input, which is available. So we produce the top output and set its done flag. Similarly, we also produce the bottom output and set its done flag because it depends only on the bottom input, which is also available. Finally, after all the outputs have been produced and all the inputs are available, we update the state, dequeue all the input FIFOs and clear all the done flags, as shown in Figure 1(d). The conversion from a specification into an LI-BDN is what we call the *LI-BDN transformation* of a module. The two requirements, that an output waits only for the inputs that it depends on, and that all the input FIFOs are dequeued when all the inputs are available and all the outputs have been produced, together guarantee the absence of deadlocks from the LI-BDN transformation.

The time duration between the enqueuing of the output FIFOs and the dequeuing of the input FIFOs comprises one model cycle. During one model cycle, the implementation can use any number of FPGA cycles to produce the outputs or to update the state. In this manner, we decouple the model cycle from the FPGA cycle and enable an efficient FPGA implementation of the model through the use of various implementation refinements. For example, a multi-ported register file which consumes many LUTs (a scarce FPGA resource) can be implemented using a dual-ported block RAM (a plentiful FPGA resource). The LI-BDN technique allows us to simulate the many ports and the combinational reads of the register file using the two ports and the one-cycle-latency reads of the block RAM. Similarly, a single-cycle 64-bit divider which runs at 4 MHz can be implemented using a 32-cycle divider which runs at 140 MHz. We use the LI-BDN technique to simulate a single-cycle divider from the 32-cycle divider, and since the divider is infrequently used, we obtain an overall improvement in performance.

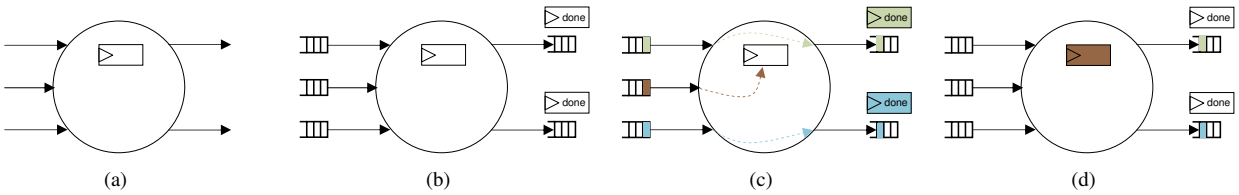


Fig. 1. Transforming a cycle-level specification into an LI-BDN

B. Debugging using the LI-BDN technique

The major requirement for debugging a large and complex model is to have the ability to freeze it in a particular model cycle so that a precise snapshot of all the state can be obtained. Such an ability is similar to taking a snapshot of the architectural state of an out-of-order processor for precise exceptions. The use of the LI-BDN technique allows us to provide this functionality at the granularity of modules that make up a model. One can also choose to freeze the entire model by freezing all the modules in the same model cycle.

We make use of the module from Figure 1(a) to demonstrate how its LI-BDN implementation can facilitate debugging. Consider a model which is comprised of many such modules. If we need to gain precise read and write access to the state inside a module during simulation, we add a 1-bit input port and a 1-bit output port to the module, as shown in Figure 2(a). Every model cycle, the module produces 1 or 0 on the new output port, and ignores the new input port. We then transform the module into an LI-BDN and attach the external interface of the new ports to some logic, as shown in Figure 2(b). The logic can freeze the module in model cycle n by dequeuing n times from the FIFO attached to the new output port, and enqueuing $n-1$ times into the FIFO attached to the new input port. A debugger can now either read or assign the value of the state in the n^{th} model cycle. Also, any such transformed module can be frozen independently of the rest of the model.

IV. Flexible Simulation Platform

The design and implementation of Arete provides simulation speed and accuracy along with ease of modification and portability. We started by writing a cycle-level specification of the processor, and then employed the LI-BDN technique described in section III to incorporate various implementation refinements which helped achieve an efficient FPGA implementation. In the process, we built a library of components which may be used for FPGA implementations of other models. We used Bluespec System Verilog (BSV) [8] to develop Arete.

In this section we describe the architecture of the processor being modeled, along with the flexibility of the model, and its portability to various FPGA platforms.

A. Processor Architecture

The processor makes use of a tiled architecture where the number of tiles is a synthesis parameter that is specified according to the resources available on a particular FPGA platform. As shown in Figure 3, each tile is composed of a parameterized number of cores, a shared L2 cache, a cache-coherence engine and a network controller. Each tile directly accesses a region of DRAM memory, the size of which is platform dependent. A network layer connects all the tiles in the processor.

1) Core: The core comprises of a 64-bit, in-order PowerPC pipeline and implements the Power ISA—Embedded Environment [9]. Figure 4 shows the microarchitecture of the core. The pipeline is designed to provide a high degree of flexibility, and includes the following features.

(I) Pipeline stages can be split or combined without modifying the rest of the pipeline because the stages are designed to be latency-tolerant.

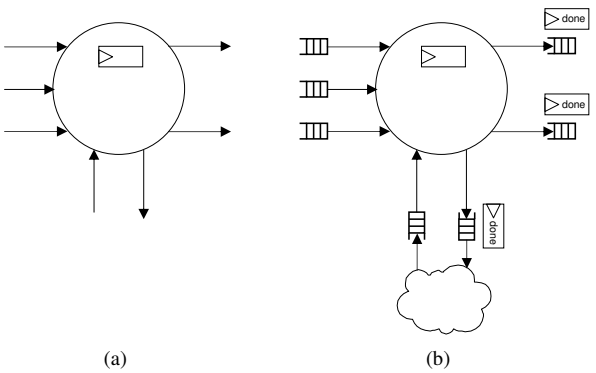


Fig. 2. Decoupled debugging enabled by the use of the LI-BDN technique

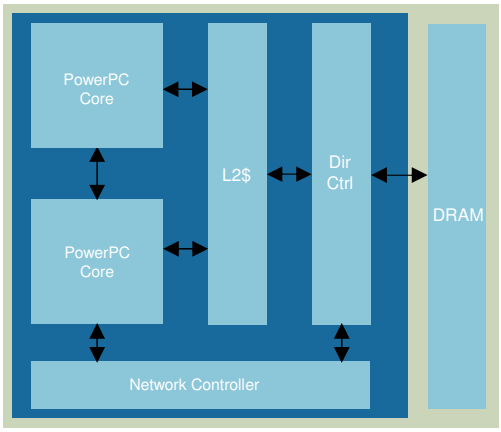


Fig. 3. Architecture of a processor tile

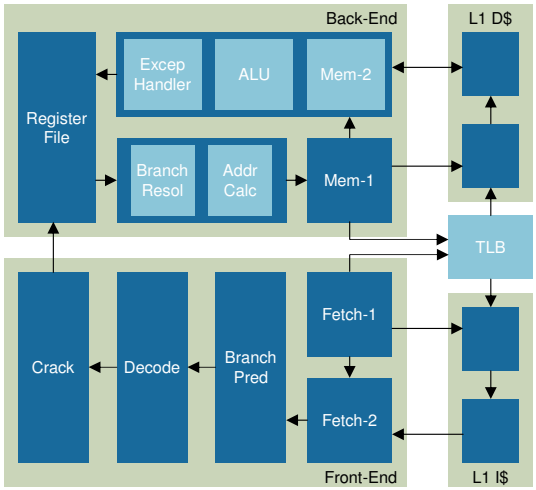


Fig. 4. Architecture of an in-order PowerPC core

(II) The mechanism to handle change in instruction flow allows any stage to perform branch prediction, branch resolution or exception handling.

(III) Any stage can read the register file and the various special purpose registers, but only the last stage updates them when committing instructions. Updated register values are fully bypassed, but the pipeline may still stall due to read-after-write hazards.

Each core has private instruction and data L1 caches with a pipelined hit latency of 1 model cycle. These caches are parameterized for associativity, line size, number of entries and replacement policy. The tag and data arrays of the L1 caches are implemented on block RAMs. The core also has a TLB which is parameterized for number of entries, and is implemented using a combination of block and distributed RAMs. It provides multi-ported combinational access for instruction and data address translation.

One of the key features of the core's design is its modularity. It can support a completely different RISC ISA with appropriate modifications confined to the decode and the MMU modules.

2) Shared memory and cache-coherence: Figure 5 shows the hierarchical structure of the shared, coherent memory architecture which forms the backbone of the multicore processor. We have designed and implemented a hierarchical, directory-based MSI protocol to provide cache-coherence. The protocol maintains a set of invariants which guarantee the absence of deadlocks.

The L2 cache is inclusive and is shared by all the cores in a tile. It is parameterized for associativity, line size, number of entries, replacement policy and access latency. Access latency is a runtime parameter while the rest of the parameters have to be specified before synthesis. The tag arrays and the directory state in the L2 cache are implemented on block RAMs, while the data arrays are mapped to a private region of DRAM.

We have arranged the main memory in a distributed and shared manner where each tile has fast access to the region of main memory to which it is directly connected, but it has to traverse the network layer to access those regions which are connected to other tiles. Off-chip main memory is incorporated into Arete as an LI-BDN module. This enables us to model its access latency which is another runtime parameter of the model. A private region of DRAM is used to implement the directory state in the main memory which provides cache-coherence among L2 caches.

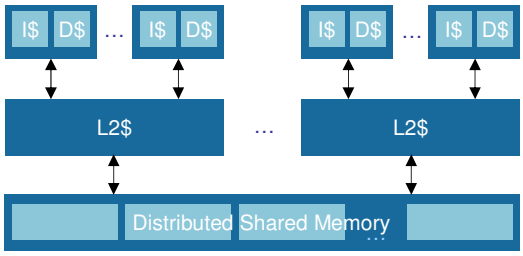


Fig. 5. Shared memory architecture

Just like the core, the memory subsystem is designed to be quite flexible. One can implement a new cache-coherence protocol by modifying the cache-coherence engine alone. Similarly, memory organization can be completely altered without modifying the rest of the system, namely the core and the on-chip network.

3) On-Chip network: The current implementation of the network architecture supports a bidirectional, all-to-all topology. It is capable of handling four types of traffic: cache-coherence, inter-core messaging, debugging and display, as shown in Figure 6. All messages received by the network layer are first packetized, and then each packet is broken down into flits with parameterized bit width, before being sent across the network. We provide virtual channels for the four kinds of traffic and for each pair of nodes. These virtual channels include appropriate amount of buffering and utilize flow control mechanisms to ensure a deadlock-free network. The network model can be modified in isolation to support various other topologies as well as routing algorithms.

4) Message-passing support: We have added a message-passing layer to the model which allows any core in the processor to communicate with all other cores via messages defined by the Power ISA. The message-passing layer supports

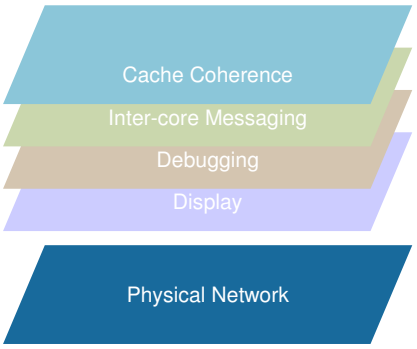


Fig. 6. Various types of traffic supported by the on-chip network

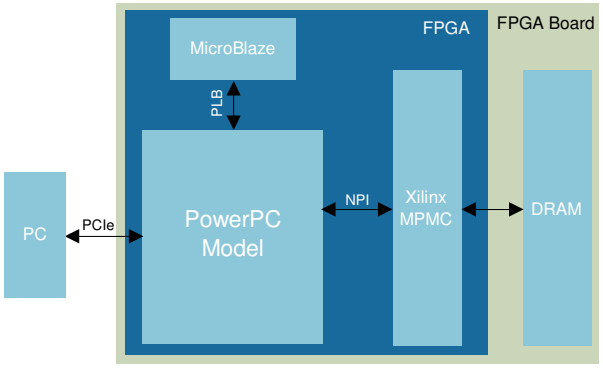


Fig. 7. A complete view of the FPGA implementation of Arete

	Prototype	LI-BDN
LUTs	105104	24153
Flip Flops	638678	16165
Block RAMs	0	43
DSP Slices	12	12
FPGA Freq (MHz)	4.8	110
FMR	1	9
Eff Freq (MHz)	4.8	12.2

Fig. 8. Comparison of the prototype and the refined LI-BDN implementations of PowerPC on the XUPv5 board. Model parameters: 1 tile, 1 in-order 10-stage core, 64 KB 4-way associative L1, 512 KB 4-way associative L2, 512MB DRAM

both unicast and multicast messages. These messages are used either by the primary core to wake up the secondary cores or by any core to raise a doorbell interrupt in another core.

B. Flexibility

Due to our platform's modularity and parameterization, we were able to conduct two significant and distinct architectural explorations on Arete with limited effort. The design, verification and evaluation of three different branch prediction schemes required only 2 man-days worth of work. A significant overhaul of the cache-coherence protocol to support software management of caches was carried out in 30 man-days.

C. Portability

As shown in Figure 7, the model communicates with three external resources: a Xilinx multi-ported memory controller (MPMC) which provides access to DRAM, a Microblaze soft core which runs debugging software, and a PC which provides access to a text terminal. For a particular FPGA platform, we wrap the interfaces to the three resources in order to present latency-insensitive, request-response interfaces to the model. We have ported Arete to three FPGA boards: XUPv5, ML605 and BEE3. This portability does not require any modifications to the design of the model; one only needs to specify appropriate values of certain parameters before synthesis.

When porting a model to a multi-FPGA platform several issues arise. One of the main issues is that the model has to be explicitly partitioned, and a different configuration file has to be generated for each FPGA, which can become tedious. We have made use of functionally-identical partitions and a distributed protocol for assigning identifiers. Together they enable one configuration file to program all the FPGAs.

Another issue is that implementing a model on multiple FPGAs can alter its timing behavior. We, however, are able to preserve the timing behavior through the use of the LI-BDN technique.

D. Simulation infrastructure

We have attempted to provide a comprehensive simulation infrastructure for architectural exploration and verification. We make use of the debugging feature enabled by the use of the LI-BDN technique to build a debugging environment for Arete. A Microblaze soft core runs debugging software, written in C, which provides a GDB-like interface to the user. The debugging software handles low-level model initialization and provides access to all model state during simulation. Linux 2.6.32 boots on Arete and we use Buildroot [10] to generate a cross-compilation toolchain for the PowerPC architecture, and a root filesystem. We also run the BusyBox package [11] which provides many common UNIX utilities.

V. Evaluation

In this section we evaluate Arete in three ways. First we determine the resource savings and performance improvements obtained from using various implementation refinements enabled by the use of the LI-BDN technique. We then evaluate the performance of Arete by running the PARSEC benchmark suite on top of SMP Linux. Finally, we evaluate the flexibility of Arete by presenting two projects which modified it significantly.

A. Synthesis statistics

In section III we described how a refined LI-BDN implementation of a cycle-level specification can achieve both higher performance and reduced resource utilization on FPGAs. To gauge the impact of the use of the LI-BDN technique and implementation refinements, we synthesized both the cycle-level specification of a single-core processor model, that we call the prototype, and its transformed and refined LI-BDN counterpart. The LI-BDN version of the model included such implementation refinements as the 5-ported register file being simulated by a dual-ported block RAM, and complex combinational logic with long critical path being simulated by its multi-cycle counterpart.

Figure 8 shows the comparison between the two implementations. The refined LI-BDN implementation uses a fourth of the LUT resources consumed by the prototype and provides a twenty times speedup in the FPGA clock speed. The FMR (FPGA to model cycle ratio) statistic listed in the table is the average number of FPGA cycles used to simulate a model cycle. As mentioned before, the multi-cycle implementation of complex combinational logic is infrequently used. This results in an FMR of 9 for the LI-BDN implementation, even though it takes up to 32 FPGA cycles to simulate some combinational

	LUTs	Flip Flops	Block RAMs	DSP Slices
Branch Prediction	357	611	1	0
Decode	1016	392	0	0
ALU	11134	4426	0	12
L1 I-Cache	1982	1795	20	0
L1 D-Cache	2923	2203	20	0
TLB	2330	896	1	0
Miscellaneous	5165	6137	1	0
PowerPC Core	24907	16460	43	12
L2 Cache	4597	5407	24	0
Directory Controller	3238	3674	0	0
Network Layer	5653	6816	2	0
Peripherals	5980	7207	8	0
Overall	69282	56024	120	24
Utilization	71%	57%	56%	18%

Fig. 9. Resource utilization for the refined LI-BDN implementation of the PowerPC model and peripherals on the BEE3 board. Model parameters: 1 tile, 2 in-order 10-stage cores, 64 KB 4-way associative L1, 512 KB 4-way associative L2, 1GB DRAM

logic. The low FMR allows the LI-BDN implementation to provide a 2.5× improvement in performance over the prototype.

Figure 9 provides a detailed breakdown of the resources used by the various modules in the refined LI-BDN implementation of a dual-core processor model on one FPGA chip of the BEE3 board [12]. The first section of the table lists the major components of the processor core, while the second section lists the components of the tile.

B. Performance evaluation

We implemented an 8-core processor model on the BEE3 board, where each FPGA chip was programmed to simulate one tile of the processor. The model was implemented with the configuration provided below.

Tiles	4, all-to-all connected
Cores	8, in-order, 10-stage
L1 ICache	private, 64 KB, 4-way set-associative
L1 DCache	private, 64 KB, 4-way set-associative
L2 Cache	shared, 512 KB, 4-way set-associative
DRAM	distributed, shared, 4GB

We ran a subset of the PARSEC benchmark suite on top of SMP Linux, and calculated the performance using counters built into the model. Figure 10 shows the speedup for the various benchmarks as the number of allocated cores increases from 1 to 8. For each benchmark, the speedup is normalized with respect to single-core performance. Ferret, which is very communication intensive, and Freqmine, which is parallelized with OpenMP, exhibit almost no speedup. The remaining benchmarks are parallelized using the Pthreads library, and scale between 4× to 8× from 1 to 8 cores. When all the 8 cores were allocated, the processor model was able to achieve a performance of 55 MIPS on average.

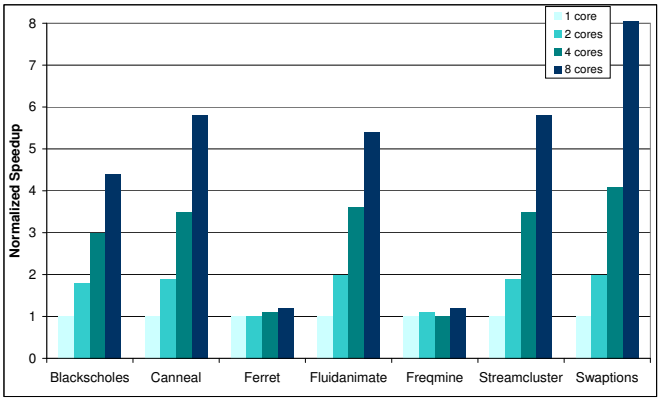


Fig. 10. Performance evaluation using the PARSEC benchmark suite running on top of SMP Linux. Model parameters: 4 tiles, 8 in-order 10-stage cores, 64 KB 4-way associative L1, 512 KB 4-way associative L2, 4GB DRAM

C. Comparing branch predictors

While designing the processor pipeline, we considered three branch prediction schemes. The first scheme simply predicts every branch to be not taken (the ANT scheme). The second scheme predicts the direction of the branch using a 2-bit, 1K-entry branch history table which is added to the branch prediction stage (the BHT scheme). The third scheme adds a 4-way set-associative, 1K-entry branch target buffer to the fetch-1 stage (the BTB scheme).

The three variants of the single-core model were implemented on an XUPv5 board. Figure 11 shows the IPCs of each of these variants for booting Linux and running 5 benchmarks selected from the SPECINT2000 suite. This graph and the synthesis statistics obtained from Xilinx tools show that the BHT scheme provides a 7% to 20% improvement in IPC over the ANT scheme while adding 1 block RAM, and increasing LUT utilization by 2% and flip flop utilization by 1%. The BTB scheme provides an additional 0.5% to 12.5% improvement over the BHT scheme, but it adds another 8 block RAMs and increases LUT utilization by 5% and flip flop utilization by 3%.

The use of an accurate, full-system simulator to conduct these experiments provided insight into the impact of the architectural changes on the system as a whole. Not only were we able to explore and understand the reasons behind the improvement in performance, we also gained an appreciation for the hardware constraints, namely, limited resources and short critical paths.

D. Data movement control

Software performance is often dependent on the performance of the cache-coherence engine and the latencies of the interconnect. We extended the cache-coherence engine in Arete to explore if performance could be improved by providing software with more control over the contents of the caches. We refer to the extensions as Data Movement Control (DMC) and they are in the form of three new instructions: a) `cpush`, which allows a thread running on one core to move cache lines into another core's cache, b) `clookup`, which returns the location of a cache line and c) `cmsg`, which provides an

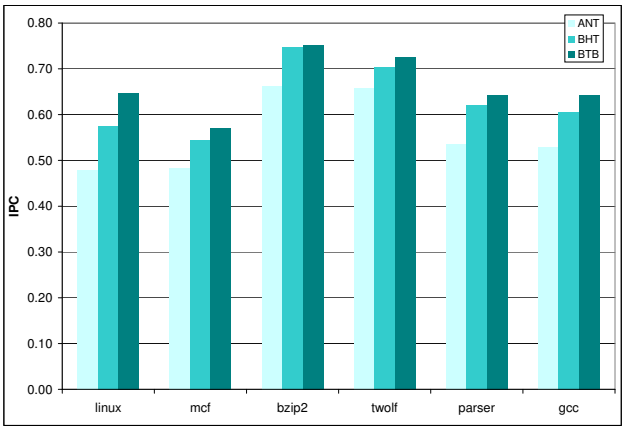


Fig. 11. Effect of different branch prediction schemes on IPC. Model parameters: 1 tile, 1 in-order 10-stage core, 64 KB 4-way associative L1, 512 KB 4-way associative L2, 512MB DRAM

efficient mechanism for software to send an active message [13] to a remote core, allowing a thread to efficiently manipulate data in the remote core's cache. Collectively, these instructions address some of the shortcomings of previous software-only cache management solutions.

The majority of our modifications to Arete were isolated to the cache-coherence engine. We did, however, modify the instruction decoder to add support for the new DMC instructions and made few minor modifications to the ALU to improve the efficiency of `cmsg`. On the software side, we developed a DMC run-time, which includes threads, a thread stealing scheduler, locks, a linked list implementation, and a memory allocator.

We evaluated the DMC extensions using a dual-core processor model implemented on the BEE3 FPGA board. The model included 64 KB L1 caches and a 512 KB shared L2 cache. The access latency for L2 cache was 32 cycles while that for DRAM was 256 cycles. We used microbenchmarks, programmed to run with and without making use of the DMC instructions, to measure performance.

To evaluate the potential performance improvement from using `cpush` we wrote a microbenchmark that repeatedly migrates a thread between two cores and measures the average round-trip time. When the source core migrates the thread to the destination core, it uses `cpush` to move the thread context to the destination core's cache. This reduces cache misses when the destination core starts executing the thread.

Figure 12(a) presents the results from the thread migration microbenchmark. The x-axis shows the number of cache lines in the thread context that the benchmark uses `cpush` to move from the source to the destination core. The y-axis measures the round-trip time in cycles to migrate a thread from the source to the destination and back to the source. As the benchmark uses `cpush` to move more cache lines the round-trip time reduces steadily up to 6 cache lines, where the round-trip time is 831 cycles. Pushing more than 6 cache lines does not decrease the round-trip time further because the network buffers become full.

One of the performance bottlenecks of many Linux kernel subsystems lies in the updates to linked lists that are protected by spin locks [14]. Often a thread will acquire a spin lock, update the linked list and some metadata associated with the linked list, then release the spin lock. We wrote a linked list microbenchmark to measure the potential improvement in performance from using `cmsg`. The microbenchmark uses `cmsg` to update the linked list, and the associated metadata on the core that caches them.

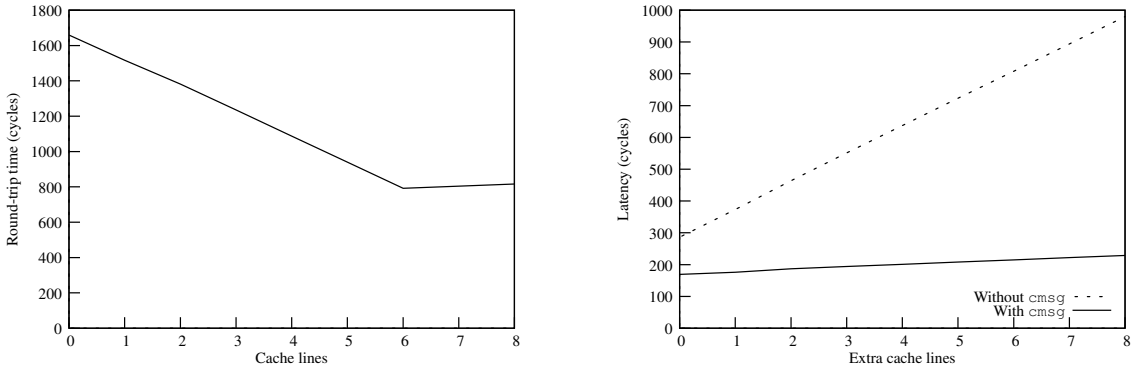
The benchmark initializes the list by inserting some elements into the list. It then creates two threads that insert into or remove from the list. For each insertion or deletion, the benchmark modifies a variable number of shared cache lines, which is meant to represent updates to metadata that take place in real workloads. The microbenchmark uses `cmsg` to update the linked list, by invoking `cmsg` with the address of the spin lock protecting the linked list and the PC of the addition or removal function. Since a thread always updates the linked list and metadata after acquiring the spin lock, it is likely that all data will be held in the same cache.

Figure 12(b) presents the results for the linked list microbenchmark. The x-axis shows the number of extra cache lines that the benchmark modifies while performing a list operation. The y-axis shows the average latency for executing a list operation. The results indicate that, even when modifying no extra cache lines, using `cmsg` decreases latency by about 22%. As the number of extra cache lines increases, the cost of performing a list operation increases in both cases. However, the cost without `cmsg` increases much faster because every additional cache line modification incurs a cache miss, whereas the extra cache lines are quite likely to be present in the destination core's L1 cache when using `cmsg`.

Experiments, such as the ones described above, are usually carried out on software simulators which architects often modify in either C or some high-level language, such as Python. Without the timing and resource constraints of hardware, architects may implement overly simplistic or completely unrealistic designs. In our case, the use of a fast and accurate model allowed us to gauge the complexity of making architectural changes to a real multicore processor implementation. It also enabled us to swiftly determine the impact of these changes on performance and resource consumption.

VI. Related Work

Many software-based multicore simulators have been developed in recent years. Rsim [15] is a discrete event-driven simulator written in C++ and C, and provides detailed models of out-of-order superscalar processors connected via coherent shared memory. It does not run an operating system and only models user-level activity of applications. Simics [16] is a popular commercial functional simulator which, on the other hand, can boot an operating system and run applications on top of it. Simics can be coupled with detailed execution-driven performance models like Gems [17], and M5 [18]. Gems and M5 provide accurate models of the memory hierarchy and the on-chip network for a multi-core system allowing detailed evaluation of these components. Garnet [19] is one



(a) Thread migration (b) Linked list

Fig. 12. Results for the data movement control microbenchmarks. Model parameters: 1 tile, 2 in-order 10-stage cores, 64 KB 4-way associative L1, 512 KB 4-way associative L2, 1GB DRAM

such accurate model of the on-chip network which uses the Gems framework. COTSon [20] is another multicore simulator framework based on AMD's SimNow [21] which is a JIT-based dynamically-translating emulator. COTSon runs an operating system and applications on top of it. The MPARM SystemC framework [22] is a complete system-level simulator, and includes cycle-accurate cores, complex memory hierarchies and bus-based interconnection mechanisms. A linux port for MPARM is underway. BigSim [23] is another multi-core simulator which simulates a distributed memory as opposed to the shared memory model that we simulate. All of the above simulators are at least an order of magnitude slower than the FPGA-based Arete.

A recent multicore processor simulator called Graphite [24] targets systems with thousands of cores. It relaxes cycle-accuracy to attain a higher simulation speed ranging in tens of MIPS. Unlike Arete, Graphite is not a full system simulator, and it does not run an operating system.

In the past few years FPGAs have gained popularity as a promising platform for multicore architectural research, and many efforts have been made to take advantage of the higher simulation speeds that they offer. In the RAMP GOLD [4] effort, Tan *et. al* have demonstrated a 64-core shared-memory target architecture. They have built a detailed memory model which does not include cache-coherence. They have a perfect core model which only stalls due to cache misses, and their network model comprises of a magic crossbar. Pellauer *et. al* [25] have recently developed a multicore simulator on FPGAs with detailed core and network models. Making use of time multiplexing, the largest system that they have demonstrated on a single FPGA chip comprises of 16 cores, but the architecture lacks support for cache coherence. Pellauer's technique uses what are called A-Ports [7], which are FIFOs connecting modules. Their methodology is similar to LI-BDNs, but they do not enforce the conditions needed to avoid deadlocks the way the LI-BDNs do. Chiou's FAST simulator [3] is split between a QEMU-based [26] functional emulator and an FPGA-based accurate timing model. They have also developed a multi-core simulator using a functional-timing split [27]. Many other efforts in this domain [28]–[32] make use of either an off-the-shelf MicroBlaze or MIPS soft core, or the PowerPC hard core found in FPGAs to build large multicore systems with detailed memory and network models.

Arete differs from earlier FPGA-based simulators, such as HAsim, in that Arete was developed using a cycle-level specification of the target processor design. This specification was transformed into FPGA-optimized RTL using the LI-BDN technique which makes Arete cycle-accurate by construction. Moreover, we maintain a clear distinction between target simplifications and implementation refinements described in Section II.

VII. Conclusion

We have presented a fast and cycle-accurate simulator for a multicore PowerPC architecture. The simulator accurately models a shared memory subsystem which includes a cache-coherence engine. We are able to run off-the-shelf SMP Linux along with several applications. We have also ported the simulator to several FPGA platforms with both single and multiple FPGAs. The simulator is highly parameterized and modular, and we have demonstrated its flexibility by performing two significant architectural explorations with little effort.

We employed several novel ideas to provide a user-friendly simulation infrastructure, which others may want to adopt.

(I) A distributed debugging environment using the LI-BDN technique enables us to independently freeze any module in any model cycle.

(II) The use of standardized interfaces makes it possible to port Arete to multiple FPGA platforms without any modifications.

(III) Functionally-identical partitions and a distributed protocol for assigning identifiers makes it possible to use one configuration file for all the FPGAs in a multi-FPGA platform.

FPGA-based modeling has a come a long way in the past few years. Although it offers substantially higher simulation speed than software, a few key issues have prevented its widespread

adoption for architectural research. We have addressed these issues in the design and development of Arete.

(I) Programmability: FPGAs are typically programmed in low-level RTL languages like Verilog or VHDL. Designing a large and complex system in RTL requires a tremendous effort. Moreover, these designs are very inflexible for architectural exploration. These issues are mitigated by the use of a high-level specification language and BSV.

(II) Resource management: Unlike software simulators, FPGA-based simulators have hard resource constraints. To meet these constraints, one has to either time-multiplex the limited resources or map the system to multiple FPGAs. Both approaches can result in a loss of efficiency if the cycle-by-cycle timing behavior of the implemented design has to be preserved. The LI-BDN technique provides a much more efficient solution because it decouples implementation from specification, and only preserves the timing behavior of the specification.

(III) Interfacing with off-chip memory or host PC: These interfaces tend to be quite complicated and ill-documented. We have minimized this problem by wrapping these low-level interfaces with split-transaction (send/receive) interfaces. We have done this to port Arete to the three FPGA boards that are being commonly used for academic research.

Moving forward, we are developing a new high-level hardware description language that allows architects to conveniently specify the cycle-by-cycle behavior of a target design. One of the goals of this work is to generate efficient synthesizable RTL from these specifications. Another goal is to develop a tool that will automatically transform these specification into LI-BDNs.

We also plan to augment Arete with a detailed model of the network architecture that will allow us to experiment with various network topologies as well as routing algorithms. We are also extending Arete to facilitate research on hardware-software co-design. One of the key challenges in this area of research is to figure out the optimal hardware-software partitioning of algorithms for performance and power. Due to its modularity Arete can readily accommodate algorithm-specific hardware accelerators for exploring many such partitions.

ACKNOWLEDGEMENTS

We would like to thank Kattamuri Ekanadham and Jessica Tseng for their help and guidance. The research work presented in this paper was funded in large part by IBM. We are thankful to Quanta Computer for supporting this work in the later stages. We would also like to thank Xilinx for donating FPGAs, FPGA boards and development tools. Many thanks to all members of the CSG-Arvind group whose valuable input helped us shape the narrative in this paper.

REFERENCES

[1] 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, no. 2, pp. 1–32, 2009.

[2] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "Quick Performance Models Quickly: Closely-Coupled Partitioned Simulation on FPGAs," in *ISPASS '08: Proceedings of the International Symposium on Performance Analysis of Systems and software*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 1–10.

[3] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators," in *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 249–261.

[4] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanovic, "RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors," in *DAC '10: Proceedings of the 47th Annual Design Automation Conference*, 2010, pp. 463–468.

[5] M. Vijayaraghavan and Arvind, "Bounded dataflow networks and latency-insensitive circuits," in *MEMOCODE'09: Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 171–180.

[6] C. Bienia and K. Li, "PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[7] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "A-Port Networks: Preserving the Timed Behavior of Synchronous Systems for Modeling on FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 3, pp. 1–26, 2009.

[8] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," in *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, June 2004, pp. 69 – 70.

[9] *Power ISA Version 2.05*, IBM, October 2007.

[10] Buildroot: making Embedded Linux easy. [Online]. Available: http://buildroot.uclibc.org/

[11] N. Wells, "BusyBox: A Swiss Army knife for Linux," *Linux Journal*, november 2000.

[12] J. Davis, C. Thacker, and C. Chang, "BEE3: Revitalizing computer architecture research," *Technical Report MSR-TR-2009-45*, April 2009.

[13] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992, pp. 256–266.

[14] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of linux scalability to many cores," in *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI 2010)*, Vancouver, Canada, October 2010.

[15] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve, "Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors," *IEEE Computer*, pp. 40–49, 2002.

[16] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *IEEE Computer*, pp. 50–58, 2002.

[17] 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, "Multifacets General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, September 2005.

[18] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, pp. 52–60, July 2006.

[19] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, April 2009.

[20] 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.

[21] R. Bedichek, "SimNow: Fast Platform Simulation Purely In Software," in *HotChips 16*, August 2004.

[22] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC," *Journal of VLSI Signal Processing*, pp. 169–182, 2005.

[23] G. Zheng, G. Kakulapati, and L. Kale, "BigSim: a parallel simulator for performance prediction of extremely large parallel machines," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, April 2004.

[24] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A Distributed Parallel Simulator for Multicores," in *HPCA-16: Proceedings of the 16th International Symposium on High-Performance Computer Architecture*, January 2010.

[25] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer, "HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing," in *Proceedings of the 17th International Symposium on High-Performance Computer Architecture*, February 2011, pp. 406 –417.

[26] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX 2005 Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[27] D. Chiou, H. Angepat, N. A. Patil, and D. Sunwoo, "Accurate Functional-First Multicore Simulators," *Computer Architecture Letters*, July 2009.

[28] P. G. Del Valle, D. Atienza, I. Magan, J. G. Flores, E. A. Perez, J. M. Mendias, L. Benini, and G. D. Micheli, "A Complete Multi-Processor System-on-Chip FPGA-Based Emulation Framework," in *Proceedings of the IFIP Conference*, 2006, pp. 140–145.

[29] M. D. Nava, P. Blouet, P. Teninge, M. Coppola, and T. Ben-Ismail, "An Open Platform for Developing Multiprocessor SoCs," *IEEE Computer*, pp. 60–67, July 2005.

[30] A. Krasnov, A. Schultz, J.Wawrzynek, G. Gibeling, and P. Droz, "RAMP Blue: A Message-Passing Manycore System In FPGAs," in *Proceedings of International Conference on Field Programmable Logic and Applications*, 2007, pp. 27–29.

[31] M. A. Kinsy, M. Pellauer, and S. Devadas, "Heracles: Fully Synthesizable Parameterized MIPS-Based Multicore System," *International Conference on Field Programmable Logic and Applications*, pp. 356–362, 2011.

[32] N. Sonmez, O. Arcas, G. Sayilar, O. S. Unsal, A. Cristal, I. Hur, S. Singh, and M. Valero, "From Plasma to BeeFarm: Design Experience of an FPGA-based Multicore Prototype," in *Proceedings of the 7th International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, 2011, pp. 350–362.