ParTejas: A Parallel Simulator for Multicore Processors

Geetika Malhotra†, Pooja Aggarwal†, Abhishek Sagar‡, Smruti R. Sarangi†
† Computer Science Department, Indian Institute of Technology, Hauz Khas, New Delhi, India
‡ Adobe Systems India, Adobe Towers, Noida, U.P, India
E-mail: † {mcs122798, csz112569, srsarangi}@cse.iitd.ac.in, ‡sagar23jan@gmail.com

***Abstract*—In this paper, we present the design of a novel multicore simulator called `ParTejas` . It is a fast shared memory based parallel simulator written in Java. Unlike recently released parallel simulators that mainly rely on sampling, high level models, and highly relaxed synchronization, we primarily rely on novel concurrent data structures. In specific, we use a lock free parallel slot scheduler for synchronizing the accesses of multiple threads at a shared resource, and we use flexible barriers known as *phasers* to relax synchronization within bounds. We leverage additional language specific features of Java, and demonstrate a mean speedup of 11.8X (simulation speed of 4-8 MIPS) with 64 threads for a suite of Splash2 and Parsec benchmarks.**

I. Introduction

An *architectural simulator* is a very important tool in computer architecture education, design, and research to evaluate different architectural designs, and research ideas. Due to continued Moore's law based scaling, the number of cores per chip is doubling roughly every two years. Hence, there is an exponential rise in the number of cores that need to be simulated, and thus it is getting increasingly difficult to use traditional single threaded sequential simulators [1], [2]. Therefore, we need parallel simulators.

In this paper, we present the design of a Java based architectural simulator called `ParTejas` . In terms of performance, Java is regarded to be generally slower. However, with advances in just-in-time (JIT) compiler technology, the performance of Java programs is becoming competitive with heavily optimized C++ programs. For heavily object oriented code, researchers have reported speedups [3] with Java. We use Java because of its built in multi-threading capabilities, concurrent data structures, library support, debugging features, platform independence, garbage collection routines, and ease of programming. `ParTejas` relies on novel concurrent non-blocking data structures like lock free slot scheduling technology [4], and *phasers* to derive speedups and simulate tightly coupled multiprocessors.

II. System Architecture

Figure 1 shows an overview of `ParTejas` . We use Intel's binary Instrumentation Engine, PIN [5], to instrument regular x86 or x86-64 binaries to provide us with execution traces consisting of a stream of packets containing the details of an executed instruction. This stream of packets is passed to waiting Java simulator threads via a fast shared memory based transfer medium. All the application threads run in parallel, and generate gigabytes of data per second. Since the number of segments are limited, we use a single shared memory segment separated into n separate contiguous regions for n application threads. Each region contains a header, and a circular queue. We use a Peterson lock optimized for the Intel x86 TSO (total store order) memory model with a single fence.

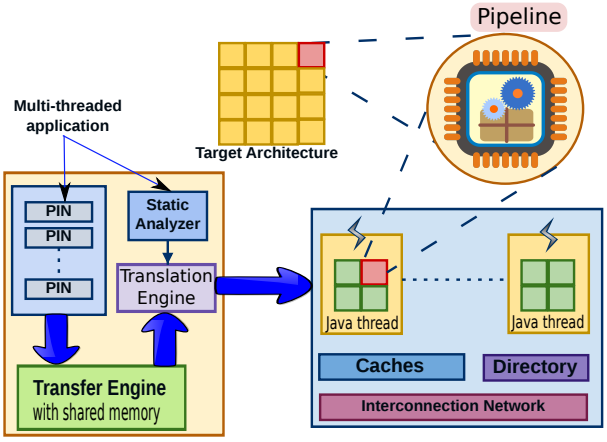

Fig. 1. Overview of `ParTejas`

If we decide to have n_{app} application threads, and n_{sim} Java threads, then each Java thread needs to simulate n_{app}/n_{sim} parallel pipelines (see Figure 1). In `ParTejas`, each Java thread independently reads the circular queues of all the application threads that it simulates. Subsequently, we translate each CISC instruction into a set of RISC instructions as follows.

First, we read the contents of the binary by parsing the output of the GNU *objdump* command and create a static table called the *instruction table* that saves the contents of each instruction. We then form the dynamic instruction stream using the *instruction table* and the execution traces obtained from PIN. Subsequently, we pass the instructions to the pipeline. We model both in-order and out-of-order pipelines. Like other simulators [6], [2], we use both functional simulation, and an event queue for non-deterministic events. Each pipeline simulates the timing of the instruction, and the memory instructions are passed to the memory system, which can model multilevel shared, and coherent caches. We incorporated several optimizations to further speed-up our simulation. Some of them are mentioned here:

1) We designed a fast shared memory based transfer mechanism between the application threads running natively on PIN, and our Java simulator threads.
2) We implemented a parallel port using slot schedulers for shared structures by using lock-free slot scheduling algorithms proposed by Aggarwal and Sarangi [4]. The *parallel port* is a matrix of slots, where the number of rows is equal to the number of simultaneous requests that can be processed in the same cycle (capacity), and the columns represent the cycle number in the current epoch.

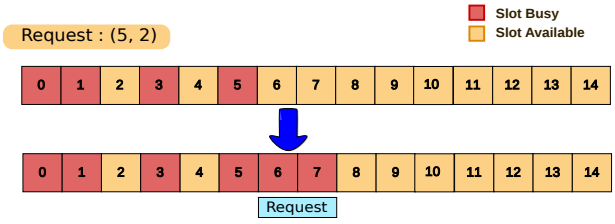

Fig. 2. Operation of a parallel port

Figure 2 shows an example of a parallel port that can process only one request in a given cycle. The first argument of a request is the requested starting slot(cycle) number, and the second argument is the number of consecutive slots that are required. The parallel port schedules the request in slots 6 and 7.

3) The threads periodically synchronize at every epoch boundary. An imbalance between the fastest thread, and the slowest thread introduces a lot of idling in the system. Since programs have very long phases in which the amount of interaction between threads is low, we propose an optimization that replaces barriers with *phasers*. We use phasers to consider two epochs at a time. A thread signals other threads after finishing the first epoch. Instead of stopping it continues till it reaches the boundary of the second epoch. At that point, if there is any thread that has not reached the end of the first epoch, it stops.
4) To reduce the cache misses due to swapping of an application thread with a simulator thread, we statically partition the set of cores such that the operating system knows that there are two classes of threads using the $sched\_affinity()$ call in Linux.
5) Lastly, we propose a host of Java specific optimizations namely selection of appropriate data structures, fine grained locking, and selective use of pooling.

III. Conclusion and Results

We evaluated the performance of `ParTejas` on a four socket, 64 bit, Dell PowerEdge R820 server with a suite of Splash and Parsec benchmarks. It had four 8 core 2.20GHz Intel Xeon cpus (with hyper-threading enabled), 16 MB L2 cache, and 64 GB of main memory. This server runs Ubuntu Linux 12.10. All our code is written in Java 6 using Sun OpenJDK 1.6.0_27 with the latest patches. We use Intel PIN (rev:49306) [5], with gcc 4.7.2. We show our results with a multi-issue in-order pipeline. For all our experiments, we simulate a 64 core system. We use an epoch size of 1000 cycles for all our simulations.

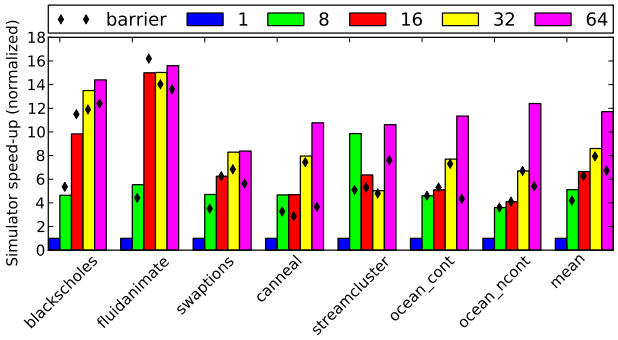

Fig. 3. Performance results (speedups with respect to 1 core)

Our performance results are shown in Figure 3. We partition the set of hardware threads between PIN and `ParTejas`. PIN runs 64 application threads, and we instantiate n (8,16,32, and 64) Java threads, where each Java thread simulates (64/n) parallel pipelines. For each benchmark, we show the normalized speedup with respect to the sequential execution time on 1 core. The bar chart shows the results with phasers and diamond shaped dots show the results with simple barriers. `ParTejas` provides a mean speedup of 11.8X with phasers, as compared to sequential execution. The speedups are 42.6% lower if we use regular barriers. We observe a slowdown with 32 and 64 threads for $streamcluster$ because of increased contention among threads in the parallel port.

`ParTejas` primarily relies on novel concurrent data structures such as the parallel port, and phasers, along with Java specific features, and intelligent core partitioning. Given, the fact that 70-80% of the time is spent in the phasers, and the parallel ports, we postulate that there might be a significant room for improvement with more sophisticated data structures, and simulation techniques.

References

[1] M. Rosenblum, E. Herrod, S.and Witchel, and A. Gupta, "Complete computer system simulation: The SimOS approach," *IEEE Parallel and Distributed Technology: Systems and Applications*, vol. 3, no. 4, pp. 34–43, 1995.

[2] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *IEEE Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[3] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman, "Benchmarking java against c and fortran for scientific applications," in *Joint ACM-ISCOPE conference on Java Grande*, 2001.

[4] P. Aggarwal and S. Sarangi, "Lock-free and wait-free slot scheduling algorithms," in *IPDPS*, 2013.

[5] Pin - A Dynamic Binary Instrumentation Tool. [Online]. Available: http://www.pintool.org

[6] E. K. Ardestani and J. Renau, "Esesc: A fast multicore simulator using time-based sampling," in *HPCA*, 2013.