Building and Using the ATLAS Transactional Memory System

Njuguna Njoroge, Sewook Wee, Jared Casper, Justin Burdick, Yuriy Teslyar,
Christos Kozyrakis, Kunle Olukotun
Computer Systems Laboratory
Stanford University
{njoroge, weese, jaredc, justy, yteslyar, kozyraki, kunle}@stanford.edu

1 Introduction

At WARFP 2005, we proposed ATLAS as a scalable implementation for transactional parallel systems [5]. The impetus for the development of ATLAS is to address the significant hurdles that software simulators face in multiprocessor architectural research. In particular, ATLAS is an FPGA-based system that primarily serves as a rapid software development platform for our transactional memory model, TCC [4]. Leveraging commodity hardware and software tools, ATLAS is poised to support research exploring the impact of transactional memory on architecture, operating systems, compilers and programming models.

To date, we have implemented a fully functional 2-processor ATLAS system on Xilinx's XUP board [8], employing the two embedded PowerPC cores in the Virtex II Pro FPGA. To interface with the TCC caches in the FPGA fabric, we have also implemented the TCC API [3]. Additionally, we have developed infrastructure to support debugging and profiling with the aim of increasing visibility and ease of performance tuning of applications.

The objective of this abstract is to discuss our experience in building ATLAS, evaluate its performance and capabilities and discuss future enhancements. Given the limited page count for this abstract, we intend on covering more details in the presentation. Also, we would like to demonstrate our system. In the demo, we will present how to use the ATLAS system to run an application from compilation to execution, demonstrating some key features of our profiling and debugging environment. The rest of the abstract is structured in the following manner: Section 2 focuses on the experience of building ATLAS. Then, in section 3, we evaluate ATLAS using some applications that run on TASSEL, TCC's software simulator. Finally, in section 4, we will conclude by outlining future work and direction for ATLAS.

2 Building ATLAS

At the inception of ATLAS, we decided to use commodity hardware (ML310 board) [5] and the accompanying software tool chain. By using a pre-packed board and software, we were able to bypass the huge up-front cost of designing a custom board and its custom software support infrastructure. After comparing availability and price-points, we migrated to the cheaper XUP board [8], which is about a tenth of the cost of the ML310. The XUP board features the same Virtex II Pro device (XC2VP30) with a higher speed-grade than the ML310. The higher speed-grade permits ATLAS to operate at a faster bus frequency (from 85 MHz to 100 MHz, a 22% improvement). The XUP board also has increased DDR memory capacity, from 256 MB to 512 MB, and there are 3 SATA connections for inter-board communication.

While we have no regrets in using a commodity board, it has not come without significant challenges in implementing the ATLAS hardware and API. In the following two sections, we discuss the issues encountered, the workarounds, the techniques developed, in implementing both ATLAS's hardware and software infrastructure.

2.1 ATLAS Hardware Infrastructure

As illustrated in Figure 1, ATLAS uses the two built-in PowerPC (PPC) hard cores. Attached to each core is a PLB (IBM's Processor Local Bus) that connects the data port of the PPC to the TCC cache. The TCC caches have configurable cache capacity as either 8, 16, or 32 kB with respective associativities of direct-mapped, 2-way, or 4-way and a configurable write-buffer of 2, 4, or 8 kB. The internal PPC data cache is de-activated and bypassed to avoid interference with TCC. Instruction fetches are sent directly to the DDR controller. Since instruction fetches bypass the TCC caches, we activate the internal 16-kB, 2-way set-associative instruction-side caches in the PPC. For transactional check-pointing storage, we use BRAM

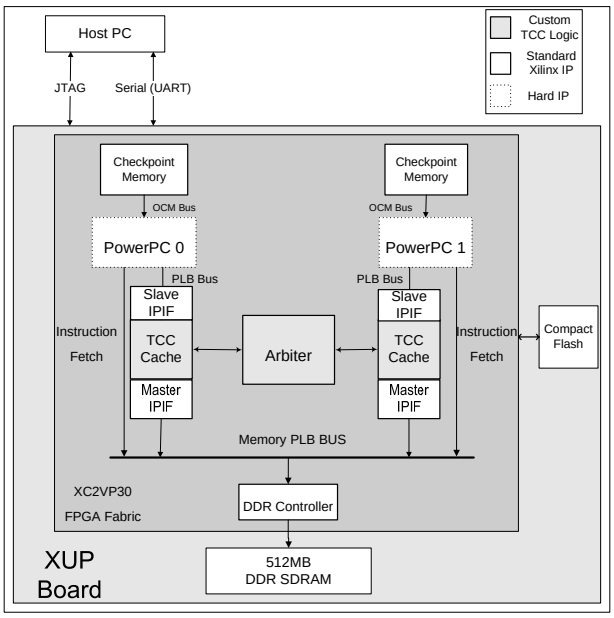

Figure 1: ATLAS Hardware Platform

(Xilinx on-chip SRAM cells) accessed through the On-Chip Memory (OCM) bus that directly interfaces with each PPC.

The first challenge encountered in building ATLAS's hardware was familiarizing ourselves with the board and its software tools. Although Xilinx EDK's capabilities and documentation are improving, it was our experience that one has to invest a substantial amount of time and effort in acclimating to the tool chain. In particular, Xilinx tutorials, examples, documentation, and wizards did not support many of the features we use. For example, it is always assumed that only one of the two processors in the FPGA was in use; there was very little support for using both of them. When implementing the cache's master PLB interface, we used Xilinx's IP Import Wizard to create an IP Interface (IPIF) [7]. Unfortunately, there are only two pages of waveforms in the manual on how to create a master interface. In contrast, the same manual contains 14 pages for the slave interface, which were helpful when we designed the cache's slave interface. Due to this dearth in documentation, the master interface took substantially more time to get working than the slave interface. Consequently, the objective of the IPIF pcore, to abstract away the complexity of interfacing with the PLB bus, was unmet. This lack of documentation for advanced features led to many unnecessary build iterations while fighting with the tools, during which the RTL and design were not changing at all. The problem was compounded by an inflexible and unoptimized build process that often performed unnecessary time consuming steps. These extra debug iterations substantionally increased design time and will hopefully be eliminated as the tools improve.

Also, we learned that using Xilinx pcores (IP soft cores) saves us design time, but the features and/or performance are not always optimal for our design. For instance, one challenge we encountered is maintaining the design at a 100 MHz bus frequency. Because Xilinx's PLB DDR controller does not function properly below 100 MHz, we considered using Xilinx's OPB (IBM's On-Chip Peripheral Bus) DDR controller, which allows you to clock the OPB bus at the desired frequency and still operate the DDR memory at 100 MHz. Unfortunately, this requires a PLB to OPB bus bridge, and while straightforward to set-up, a bridge would introduce a performance impact of an additional 125 cycles of latency per access. Thus, we chose to aggressively pipeline the TCC cache design to meet 100 MHz timing with substantial amount of effort.

Using a hard core processor also introduced a variety of challenges. In particular, the lack of access to the datapath introduced significant delays in cache accesses. We chose to interface each PPC to its TCC cache via a PLB bus and Xilinx IPIF pcore. Because of this interface, it takes more than ten cycles for a cache hit: 2 cycles from the PPC to the PLB arbiter, 3 cycles for PLB bus arbitration, 3 cycles in the PLB IPIF pcore, and 2 more in the actual TCC cache. Had we chosen to design our own soft processor core, the TCC cache could have been nestled into the processor RTL without incurring long latencies. Similarly, if we did not use the PLB bus and IPIF and instead directly connected the PPC to the TCC cache, we would also have reduced cache hit latency. However, using a soft core with a similar level of features as the PPC 405 (such as TLB and Memory Management Unit needed to run an OS like Linux, debug features, etc.) would have taken too much FPGA LUT real estate. Additionally, designing and maintaining a soft core processor would substantially increase the project's complexity.

2.2 ATLAS Software Infrastructure

The ATLAS software infrastructure is designed to be used as an architecture research tool and software development environment. For architectural study, ATLAS provides as much profiling information as possible while minimizing the overhead to collect information. It provides the parallelization factor, computation time, memory reference behavior, TCC overhead (arbitration and commit time), violation and overflow counts, overhead for an individual violation or overflow, and overall violation/overflow overhead. Hard-

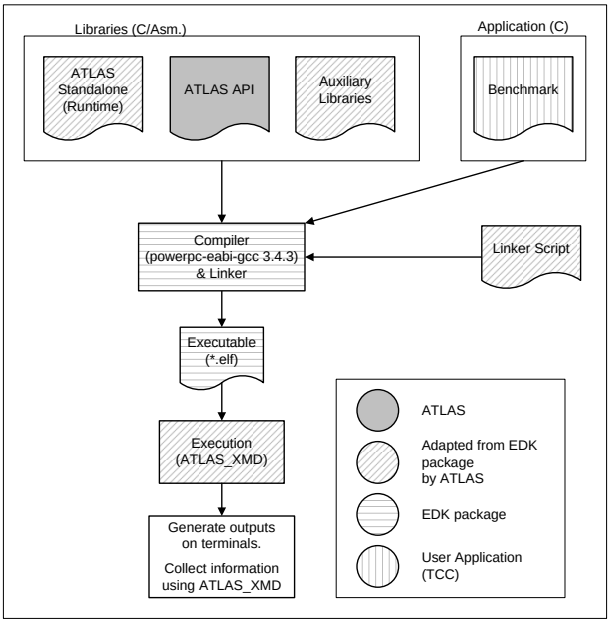

Figure 2: ATLAS Software Platform

ware counters were used when feasible, but when a hardware counter unnecessarily increased complexity or when a hard core's visibility limitations prevented hardware profiling, software was used to collect the data.

For software development, we feel easy debugging capability is an important feature. XMD, Xilinx's hardware debugging tool that uses the JTAG interface, has most of the capabilities that programmers may need. We developed ATLAS_XMD on top of that to provide TCC specific debugging commands, such as primitive TCC operations, some of the higher level TCC API calls, and macros of commonly performed tasks such as connecting to the FPGA and downloading executables. In addition to debugging support, ATLAS provides the profiling information characterized in [2] to aid in performance optimization.

The Xilinx EDK allowed easy implementation of basic functionalities that are out of our research focus, such as low level device drivers for I/O. Figure 2 shows the ATLAS software infrastructure components and design flow. However, to execute TCC applications, the TCC API was developed for the PowerPC on the FPGA and the Xilinx runtime system was modified.

The ATLAS API library is designed at the assembly level because it needs to modify the machine status manually to checkpoint and rollback. Because the PPC has no external interface to the registers, all register values had to be saved and restored sequentially. In contrast, a custom soft processor could snapshot all registers in a single clock cycle. The ATLAS API also performs profiling information collection.

Even though Xilinx's runtime system, standalone, handles many details, we had to modify it to support multiprocessor and TCC specific issues. The ATLAS runtime, ATLAS standalone, implements the statically allocated private stack for each processor, shared heap with malloc support, bss and sbss clearance by only one processor, and low level synchronization before platform initialization. These additions also required modification of the standard linker script. Also, Xilinx's standalone interrupt vector code needed changing to handle TCC violation and overflow interrupts.

ATLAS uses mostly standard EDK libraries, but we ported some optimized string and floating point operation libraries to improve performance, as described in the next section.

3 Results and Evaluation

In this section, we present the results of 5 applications that were run on TASSEL and ATLAS, with both 1 and 2 processor configurations on each platform, as graphed in Figure 3. On average, 1-p ATLAS is nearly 5 times faster than the 1-p TASSEL, and 2-p ATLAS is 8 times faster than 2-p TASSEL. Although ATLAS is faster than the software simulator, the speedup is lower than expected. Even though we labored to improve the performance of ATLAS, several bottlenecks persist because of the limitations of hardcore IPs. The primary bottleneck is the lack of a hardware FPU in the PPC core. Aggravating this deficiency is the fact that most benchmarks evaluated are floating point intensive scientific parallel applications. Initially, we considered inserting a soft core FPU attachment to the PPC, but we concluded that it would offer marginal improvement in performance. In particular, the transfer of operands and results would incur large latencies in traversing the PLB bus (Section 2.1). Moreover, two soft core FPUs would consume too many LUTs with the current 80% LUT utilization. As an alternative, we adopted a hand-optimized assembly-level library [6], which offered an average of 5 times speedup. To see how ATLAS performs on integer applications, we evaluated the integer portion of radix (where the actual sorting takes place), and we measured a speedup factor of 75 (2-p ATLAS versus 2-p TASSEL). Note that radix should be considered as a floating point intensive application, since it spends more than 90 percent of its execution time generating random numbers to sort using floating point operations.

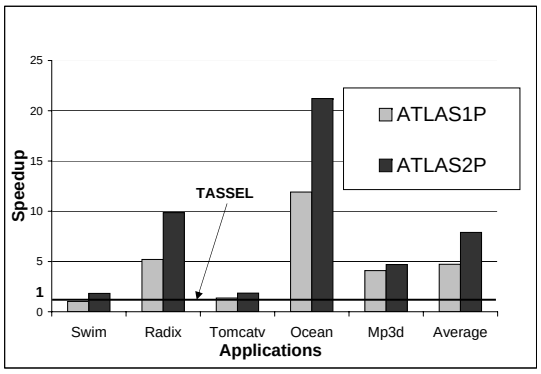

Figure 3: 1-p and 2-p ATLAS Speedups versus TASSEL

4 Conclusions

Having a 2-way ATLAS system running is the first step in scaling-up to a 16-32 processor ATLAS system. We are currently implementing an interconnection network that will be the basis for a ring topology of XUP boards. We have chosen this topology because an XUP board can only connect point to point with 2 other boards. We envision minimal changes to the actual TCC caches and API to support the multiboard configuration. Additionally, we will port Linux and start exploring the behavior of transactional applications with it.

In implementing ATLAS, we can see that FPGAs are promising platforms for conducting multiprocessor system research. However, it is apparent that the current state of boards and tools need substantial improvement. In essence, the tool infrastructure needs to evolve to a comparable maturity level that software simulators have attained over the last decade in order for FPGA-based research to gain traction. In particular, small-scale multiprocessor research typically employs 8-16 processors, whereas most FPGAs support at most 2 hard-core processors (or at most one soft-core of similar sophistication), necessitating an array of FPGAs for the research. With today's commodity boards, such projects require one to build a network of boards, which introduces more obstacles, thus increasing the implementation complexity. The RAMP project [1], currently under development, is proactively addressing these concerns. Given its promising potential, we plan to migrate ATLAS to this platform when it becomes readily available.

5 Acknowledgements

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) through the Department of the Interior National Business Center under grant NBCH104009. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

Additional support was also provided by the Samsung Lee Kun Hee Scholarship Foundation and the Stanford School of Engineering.

References

[1] Arvind, K. Asanović, D. Chiou, J. C. Hoe, C. Kozyrakis, S.-L. Lu, M. Oskin, D. Patterson, J. Rabaey, and J. Wawrzynek. RAMP: Research accelerator for multiple processors - a community vision for a shared experimental parallel HW/SW platform. Technical report, 2005.

[2] H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, J. Chung, L. Hammond, C. Kozyrakis, and K. Olukotun. TAPE: A transactional application profiling environment. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 199–208. June 2005.

[3] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 1–13. ACM Press, Oct 2004.

[4] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 102–113, June 2004.

[5] C. Kozyrakis and K. Olukotun. Atlas: A scalable emulator for transactional parallel system. In *Workshop on Architecture Research using FPGA Platforms, 11th International Symposium on High-Performance Computer Architectur*. Feb 2005.

[6] N. Leeder. Powerpc performance libraries. Sourceforge website: http://sourceforge.net/projects/ppcperflib.

[7] Xilinx. PLB IPIF. PLB IPIF data sheet: http://www.xilinx.com/bvdocs/ipcenter/data_sheet/plb_ipif.pdf.

[8] Xilinx. Xilinx XUP virtex II pro development system. XUP Website: http://www.xilinx.com/univ/xupv2p.html.