ATLAS: A Chip-Multiprocessor with Transactional Memory Support

Njuguna Njoroge, Jared Casper, Sewook Wee, Yuriy Teslyar,
Daxia Ge, Christos Kozyrakis, Kunle Olukotun

Computer Systems Laboratory
Stanford University
tcc_fpga_xtreme@lists.stanford.edu

Abstract

Chip-multiprocessors are quickly becoming popular in embedded systems. However, the practical success of CMPs strongly depends on addressing the difficulty of multithreaded application development for such systems. Transactional Memory (TM) promises to simplify concurrency management in multithreaded applications by allowing programmers to specify coarse-grain parallel tasks, while achieving performance comparable to fine-grain lock-based applications.

This paper presents ATLAS, the first prototype of a CMP with hardware support for transactional memory. ATLAS includes 8 embedded PowerPC cores that access coherent shared memory in a transactional manner. The data cache for each core is modified to support the speculative buffering and conflict detection necessary for transactional execution. We have mapped ATLAS to the BEE2 multi-FPGA board to create a full-system prototype that operates at 100MHz, boots Linux, and provides significant performance and ease-of-use benefits for a range of parallel applications. Overall, the ATLAS prototype provides an excellent framework for further research on the software and hardware techniques necessary to deliver on the potential of transactional memory.

1 Introduction

Processor vendors are turning *en masse* towards chip-multiprocessors (*CMPs*) as a practical way to turn increasing transistor budgets into scalable performance without the power and complexity challenges of aggressive uniprocessors. The trend towards CMPs in the embedded domain is as strong as it is in the desktop and server domains. Several embedded vendors, such as ARM, ARC, Broadcom, Freescale, NEC, PMC-Sierra, and Raza Microelectronics, are selling chips or licensing designs for CMPs for cache-coherent shared memory configurations. Moreover, there is significant research activity in porting embedded applications for CMPs and optimizing their microarchitecture to meet the domain characteristics [1, 15, 14, 2].

Nevertheless, the practical success of CMP-based systems is limited by the difficulty of parallel programming [24]. While in some systems we can utilize CMP cores by running many programs concurrently, parallel programming is necessary in order to reduce the execution time of one program. Existing models for multithreaded programming using locks or mutexes are challenging for most programmers as they introduce a tradeoff between performance and correctness. The use of coarse-grain locks makes it easy to write a correct parallel program but it typically limits the parallelism in the code. The use of fine-grain locks reveals additional concurrency but often leads to races, deadlocks, livelocks and other difficult bugs [16].

Transactional Memory (*TM*) [13] is a promising technology that can simplify concurrency management in multithreaded programs. TM allows programmers to define coarse-grain parallel tasks (transactions) that will be executed atomically and in isolation. Using optimistic concurrency, TM executes these tasks in parallel on a CMP, providing performance similar to that with fine-grain locks. Furthermore, transactions address other challenges of lock-based parallel code such as deadlocks and provide robustness to failures. Recently, there has been significant research on efficient software and hardware TM implementations from both academia and industry [9, 3, 20, 19, 23, 12, 11, 26, 22, 21, 17]. Hardware support for TM is considered necessary in order to eliminate the overheads of managing concurrent execution of transactions and to allow TM programs to use existing software libraries without the need for recompilation.

This paper presents *ATLAS*, the first prototype of a CMP with hardware support for transactional memory. ATLAS includes 8 embedded PowerPC cores with a shared memory system. The data cache for each core is enhanced to buffer transactional state during optimistic execution and detect potential conflicts between concurrent transactions. Moreover, ATLAS relies on transactional execution mechanisms to keep the data caches coherent, eliminating the need for a conventional cache coherence protocol [9]. We mapped ATLAS to the BEE2 multi-FPGA board [6] to provide a full-featured prototype operating at 100MHz. The system boots the GNU/Linux operating system and provides support for transaction-based parallel programming and parallel application tuning. Early experiments with full applica-

978-3-9810801-2-4/DATE07 © 2007 EDAA

tions demonstrate that ATLAS is easy to program and provides scalable performance. ATLAS provides a valuable tool for validating the advantages of transactional memory. It allows software researchers to investigate transactional programming techniques at hardware speeds. The FPGA mapping allows hardware researchers to tune hardware parameters as further insights are generated from application studies.

2 Transactional Memory Overview

To parallelize an application, a programmer must break the code up into multiple threads that can execute in parallel. The programmer must also synchronize the threads when they potentially operate on the same data in memory. The conventional synchronization approach is the use of lock primitives. If the system supports transactional memory, the programmer can synchronize threads simply by wrapping all the code that operates on the potentially shared data into a transaction. Transactions are guaranteed by the system to appear to execute atomically and isolated, even if multiple transactions are executed concurrently. TM systems achieve high performance through optimistic concurrency. A transaction runs without acquiring locks, optimistically assuming no other transaction operates concurrently on the same data. If this is true at the end of its execution, the transaction commits its writes to shared memory. If not, the transaction violates, its writes are rolled back, and it is re-executed.

A TM system must implement the following mechanisms: (1) isolation of stores until the transaction commits; (2) conflict detection between concurrent transactions; (3) atomic commit of stores to shared memory; (4) rollback of stores when conflicts are detected. Conflict detection requires tracking the addresses read (read-set) and written (write-set) by each transaction. A conflict occurs when the write-set of one transaction intersects with the read-set of another concurrently executing transaction. These mechanisms can be implemented either with hardware support (HTM) [9, 3, 20, 19] or in a software only manner (STM) [23, 12, 11, 26, 22, 21, 17]. HTM systems support transactional mechanisms at minimal overheads and make the implementation details transparent to software.

HTM systems implement speculative buffering and track read- and write-sets in caches [9]. The data caches in CMP cores are sufficient to buffer the state for the transactions in most applications [7] and simple virtualization techniques can handle the rare case of a cache overflow. For conflict detection, an HTM system can utilize the cache-coherent protocol. As messages are exchanged between cores to locate data on cache misses, it is possible to detect the overlap between a read-set and a write-set that triggers a conflict. Simulation studies have shown that such hardware TM implementations can allow multithreaded programs to synchronize in a high performance manner using fairly simple, coarse-grain, transactions in their code [18].

3 The ATLAS CMP System

ATLAS is the first full-system prototype for a CMP with transactional memory support. This section presents its basic architecture, the hardware design, and its software environment. The prototype is currently operational at 100MHz, boots the Linux operating system, and runs multithreaded applications that use transactional memory.

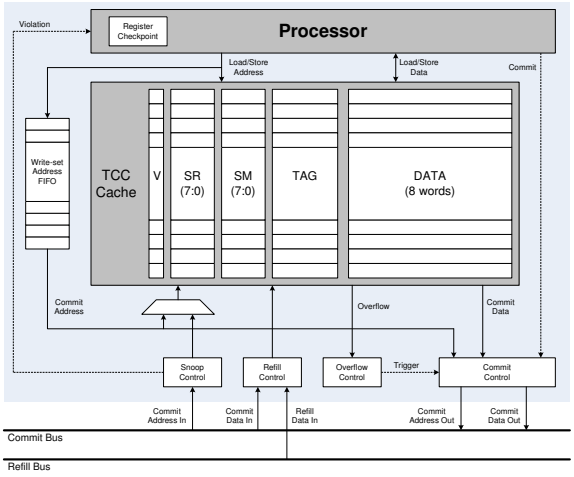


Figure 1. The data cache organization for ATLAS.

3.1 The TCC TM Architecture

ATLAS prototypes the Transactional Coherence and Consistency (TCC) architecture for hardware-based transactional memory [18]. TCC assumes a set of processor cores with private first-level caches that are connected through a snooping bus to the shared memory (shared caches and DRAM). The cores execute transactions speculatively, while tracking the read- and write-sets in the data cache organization shown in Figure 1. As a core performs loads and stores within a transaction, it sets the SR and SM bits to show that the corresponding word is now part of the transaction's read-set or write-set, respectively. The first time a word is written in a specific cache line, the cache also pushes a pointer to it into the write-set address FIFO. The cache operates as a write-buffer, isolating all writes from shared memory until the transaction completes.

At the end of the transaction, the core arbitrates for permission to commit the write-set to the shared memory. While we execute multiple transactions in parallel, only one is allowed to commit at any point in time. Once granted a commit token, the core uses the write-set address FIFO to traverse the cache and commit the transaction's writes. To detect conflicts, all other cores snoop the commit messages, searching their data caches for the committed addresses. If a word currently committed belongs to the read-set of a second transaction (the SR bit is set), a violation is triggered for the second transaction. The committing transaction is always guaranteed to complete. Before the commit is over, the processor resets the SR and SM bits in the cache. On a core that executes a transaction that violates, we undo its effect by invalidating its write-set from the cache (invalidate lines with SM bits set). Register checkpointing at the boundaries of transactions are also necessary to undo register updates.

An interesting feature of TCC is that communication be-

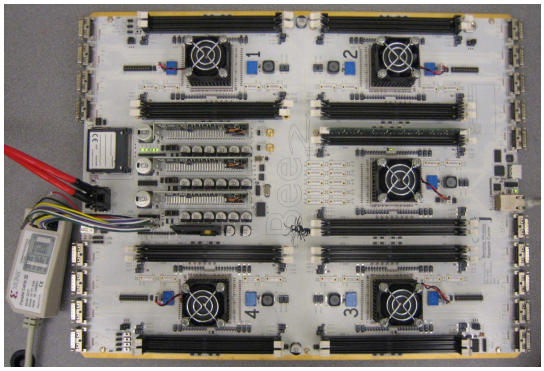


Figure 2. ATLAS on The BEE2 multi-FPGA board.

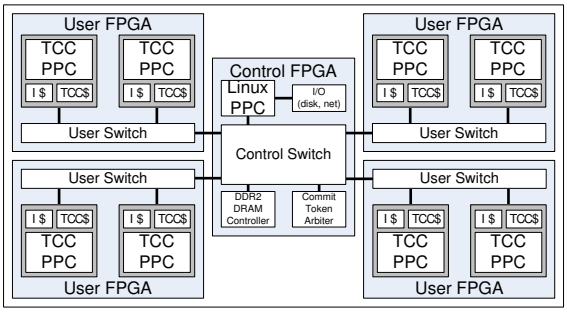


Figure 3. Block diagram of the ATLAS system.

tween cores occurs only in bulk at transaction boundaries. This is the only point where caches are kept coherent and access ordering (consistency) is enforced. In other words, TCC uses this simple commit mechanism to replace the complex cache coherence protocols like MESI. This is possible because well synchronized programs should access truly shared data only within user-defined transactions.

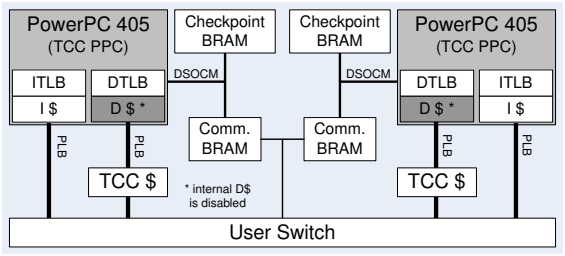
A user-defined transaction may be large enough to overflow the data cache. In this case, the core arbitrates immediately for the commit token, performs a partial commit, and does not release the token until it reaches the real end of the transaction. Such overflow events slowdown commits but are rare in tuned programs. In Section 3.3, we discuss how ATLAS provides explicit support to help programmers identify and quickly eliminate overflows and other bottlenecks in their parallel code.

3.2 The ATLAS Hardware Design

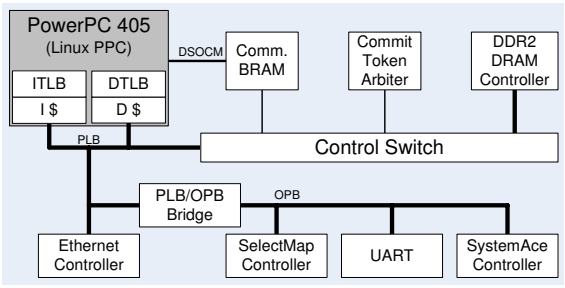
ATLAS implements the TCC architecture for CMPs with transactional memory support. ATLAS includes 8 PowerPC 405 cores that run multithreaded code for applications and a ninth core that handles the operating system and I/O devices. The design has been mapped onto the BEE2 multi-FPGA board shown in Figure 2[1] [6].

Figure 3 illustrates how ATLAS is mapped to the BEE2 board. The four outer FPGAs, labeled as User FPGAs, are connected in a star topology through the 5th FPGA, designated as the Control FPGA. Figure 4(a) shows the block

[1]ATLAS is also part of the RAMP project that aims at developing FPGA-based technology for prototyping modern CMP systems [4].



(a) User FPGA



(b) Control FPGA

Figure 4. Block diagram of the User (a) and Control (b) FPGAs.

diagram of the User FPGA. Each FPGA includes two PowerPC 405 cores enhanced with a TCC data cache. We use the hardcore PowerPC cores in the VirtexII-Pro FPGAs. We chose to use the hardcore PPC, instead of a synthesized softcore, for a shorter design time and to allow us to focus our efforts on the memory system and software components. Moreover, retrofitting a hardcore for transactional memory is an interesting challenge that many embedded SoC developers will face when building CMPs based on pre-existing hardcores.

The data cache design, written in synthesizable Verilog, is attached to the PowerPC cores through IBM's *Processor Local Bus* (PLB). The cache has 32 byte lines and has configurability in its set associativity, data capacity and write-set address FIFO capacity (4-way, 32 KB and 8 KB default setting respectively). The internal data cache in the PowerPC cores is disabled. The TCC cache is in turn connected to a network switch (shared by two cores) that forwards cache misses and commit requests to the control FPGA through a central switch. Overall, TCC's FPGA implementation adds 14% overhead in the control logic, and 29% in on chip memory as compared to a non-speculative incarnation of our cache. For instructions, we use the built-in instruction cache in each PowerPC core. As we run Linux on ATLAS, each core activates the built-in TLB (unified, fully-associative, 64 entries). We also connect two SRAM blocks to the processor through the *On-Chip Memory* (OCM) bus. The first SRAM serves for register checkpointing (in software), while the second SRAM is to coordinate system calls and exceptions for Linux (refer to Section 3.3). The two SRAMs are implemented on the FPGA using the on-chip block SRAM (BRAM) blocks available.

The Control FPGA provides a hub to connect all the processors to the shared memory and I/O devices. The current

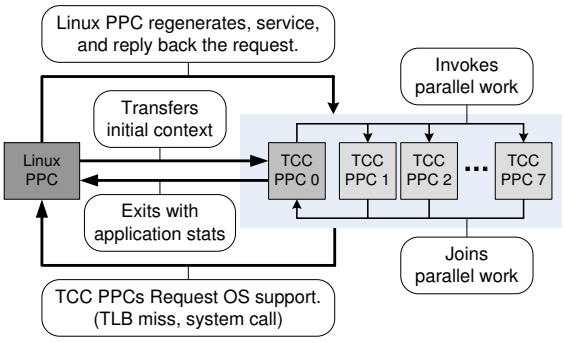


Figure 5. Overview of the ATLAS system software.

ATLAS design does not use secondary caches, though this is not fundamental. As depicted in Figure 4b, the Control Switch interfaces with the TCC token arbiter, the DDR controller and the PowerPC 405 core that runs the Linux OS. The PowerPC core for Linux uses its built-in data cache but broadcasts its memory writes to all other processors for coherence purposes. All traffic sent through switches is packetized with a single packet format to simplify routing around the system.

Another function for the Control FPGA is to provide I/O capabilities for ATLAS. A number of I/O peripherals are connected to the PowerPC core that runs Linux, including an Ethernet controller, a UART controller (for console input/output), a SelectMap controller for programming the user FPGAs through Linux, and a SystemACE controller that interfaces to a CompactFlash card. Using Network File System (NFS), ATLAS can access external storage over the Ethernet.

3.3 The ATLAS Software Design

The ATLAS software stack consists of the API for multithreaded programming with transactions and the system software.

For application programming, the user partitions work into parallel threads and defines atomic transactions within each thread. As explained in [8], this approach allows for both non-blocking synchronization of parallel code with coarse-grain code and speculative parallelization of sequential code. We currently provide an API for C-based transactional programming, which allows users to define transaction boundaries. The application is compiled with a regular C compiler (gcc in our case) and linked with the ATLAS library that provides optimized assembly-based implementations of the API calls. At the start of a transaction, the API call checkpoints the processor's register file, clears SM and SR bits in the data cache, and registers the transaction with the commit token arbiter. When the transaction completes, the API call first requests the commit token and, when granted, triggers the commit of the write-set to the shared memory. The library also defines an exception handler triggered when a data cache detects a violation for the currently executing transaction. The handler invalidates the write-set in the data-cache, restores the register checkpoint, and restarts the transaction.

The ATLAS system software is summarized in Figure 5. ATLAS runs Linux only on the PowerPC core in the control FPGA. The 8 PowerPC cores in the user FPGAs run a simple runtime kernel that coordinates with Linux. This approach is similar to the Intel MISP concept for efficient CMPs [10]. Each user application starts on the PowerPC core in the control FPGA as a regular user application under Linux. On the first call to the ATLAS library, the context of the application (private stack pointer, program counter, and process information) is transferred to the primary TCC PowerPC core (core number 0). The primary core executes the application and invokes the other 7 processors as needed on parallel regions in the program.

During the program execution, the Linux PowerPC core handles all interrupts due to external devices. It also handles any OS code needed to support the execution on the TCC PowerPC cores, like system calls or an exception such as a TLB miss. On a TLB exception,the TCC PowerPC core sends the faulting address to the core running Linux. The OS core regenerates the exception, runs the corresponding OS code to resolve it (e.g., access page table for the proper translation entry), and sends the information back to the requesting core. We handle other exceptions and system calls in a similar manner (I/O, memory allocation etc). Hence, the user code can run assuming full OS support at any point of the program. At the current scale of the ATLAS system, a single core running the OS is sufficient to serve 8 cores running application code. Using a single core for the OS allows us to run conventional Linux without special consideration for concurrency in the OS code. In larger scale configurations of ATLAS, the number of OS cores may need to be scaled as well. In this case, we will have to port SMP Linux onto ATLAS. We will also have the opportunity to explore the use of transactional synchronization in system code.

An important part of the ATLAS system is the support for performance tuning of user applications. Transactional memory makes it easy to write a correct parallel program. Nevertheless, a program may still include performance bottlenecks such as frequent transaction violations or expensive overflows [8]. To help the user identify the most significant bottlenecks, ATLAS includes a profiler framework that utilizes performance counters built in the PowerPC cores and additional counters and filters introduced in the TCC cache [5]. The hardware tracks the occurrence of all violations and overflows, the corresponding instruction and data references that triggered them, and an approximation of their cost. The profiler software uses this information to identify the most important problems and pinpoint the offending variables or lines in the user source code. The accurate feedback on performance bottlenecks allows ATLAS users to quickly tune their applications as they can focus on the important issues only and avoid the need to understand the whole application in detail.

4 Evaluation

Table 1 presents the basic statistics of the ATLAS design (default configuration). The design is currently operational on the BEE2 board running at 100MHz.

To demonstrate the performance potential of ATLAS, we ran five typical multiprocessor benchmarks: three scientific

CPUs	9 IBM PowerPC 405 cores at 100MHz 16KB 2-way I-cache 32KB 4-way TCC data cache (8 cores) 16KB 2-way data cache (1 core)
Main Memory	512MB DDR2 at 200 MHz
I/O	10/100 Mbps Ethernet, RS232 UART, 512MB Compact Flash
OS	Montavista 3.1 Linux (ver 2.4.30)
EDA Tools	Xilinx EDK 7.1i
User FPGA	Xilinx XC2VP70, 17,641 LUTs (26%), 212KB BRAMs (32%)
Ctrl FPGA	Xilinx XC2VP70, 16,284 LUTs (24%), 66KB BRAMs (10%)

Table 1. ATLAS design statistics.

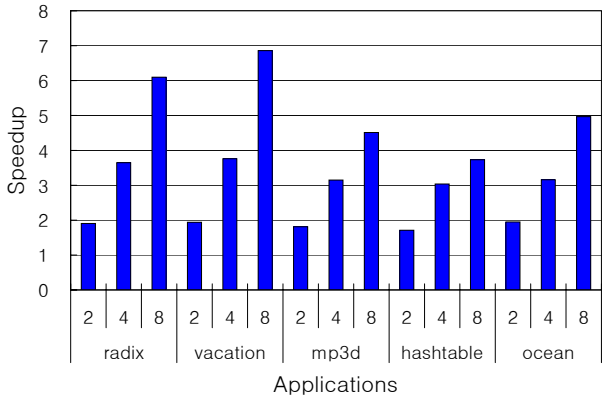


Figure 6. Speed up of execution time in ATLAS normalized to uniprocessor execution time.

benchmarks from the SPLASH2 suite [27] (radix, mp3d, ocean); a hashtable microbenchmark that performs random insert, remove, and lookup operations on a hashtable; and vacation, a benchmark that emulates the travel reservation management and database system. All applications include multiple threads that operate on data in the shared memory in an irregular manner. Coarse-grain transactions allow for a simple way to synchronize the parallel threads. There are two sets of metrics we analyze in our evaluation. First, we examine the speedup each application can achieve using 8 processors over the sequential (original) version of the application running on a single processor. Despite the use of coarse-grain transactions, we would like to see near-linear speedups. Second, we are interested in breaking down the execution time of each benchmark to its basic components such as useful or busy cycles, stall cycles due to cache misses, cycles spent on transactional execution overhead such as commit time, and idle time and violation idle due to lack of parallelism. The ability to provide such breakdowns is important for both verification purposes and to provide programmers with further profiling information.

Figure 6 shows the speedup each application achieves on ATLAS using 2, 4, or 8 of the PowerPC cores for its execution. In general, most applications scale well with the number of processors, with the exception of hashtable, which barely sped up from four to eight processors. Figure 7 shows the execution time breakdown for each appli-

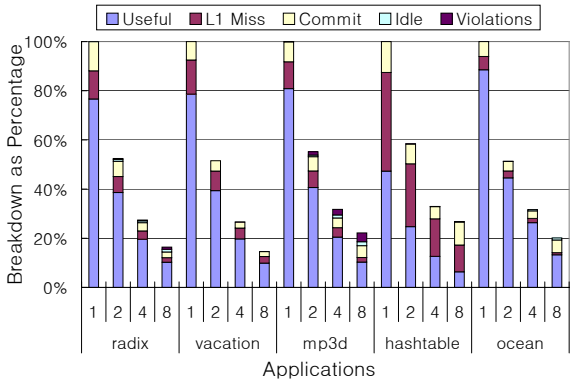


Figure 7. Breakdown of profiled real execution time in ATLAS and reported execution time from TASSEL. Numbers are normalized to the one processor case of each method.

cation. The graph indicates that hashtable does not scale to 8 processors due to the poor locality it exhibits in its data references as it performs random accesses to a large hash table. Apart from the latency of cache misses, the high frequency leads to bandwidth saturation and contention effects on the interconnect network. Specifically, the average cache refill latency for hashtable is 80 cycles, while the typical case for uniprocessor execution is 57. It is also interesting to observe, in Figure 7, that transactional execution overheads (commit time) contributes 13.3% to execution time. We have extensively compared the performance reported by ATLAS to that predicted by architectural simulators for the TCC architecture and the results match with high accuracy, except higher commit time. This mismatch is mainly from the sacrifices made to fit the design onto an FPGA, which are described in [25].

We should also point out that, despite running at 100MHz on FPGAs, the ATLAS design is 100 times faster than the TCC simulator running on a 2GHz PowerPC G5 systems.

5 Conclusions & Future Work

ATLAS is the first full-system prototype of a CMP with hardware support for transactional memory. Mapped on the BEE2 multi-FPGA board, ATLAS operates at 100MHz, boots Linux, and exhibits good performance for parallel applications written with transactional memory. It also provides direct support for performance profiling and guided tuning to further simplify parallel software development.

Looking forward, we intend to use ATLAS as the framework for further exploration of transactional memory in CMP systems. From a software perspective, ATLAS allows for fast software development and evaluation using the demanding applications and large datasets that CMPs are targeted towards. The insights from application studies will be very useful in terms of improving TM implementations and programming models. ATLAS will also allow us to study the use to transactions in the operating system code. From a hardware point of view, we are interested in studying further hardware support for parallel application development (debugging and tuning). We are also interested in larger-scale

configurations of ATLAS which should be possible given the latest generation of high density FPGAs.

6 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 National Science Foundataion Grant CCF-0444470, the Samsung Foundation of Culture, and the Stanford School of Engineering.

References

[1] ARM11 MPCore. http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html.

[2] Fall processor forum: The road to multicore. http://www.instat.com/fpf/05/index05.htm.

[3] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proc. of the 11th Intl. Symp. on High-Performance Computer Architecture (HPCA'05)*, pages 316–327, San Franscisco, California, February 2005.

[4] 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.

[5] 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: Proc. of the 19th Ann. Intl. Conf. on Supercomputing*, pages 199–208. June 2005.

[6] C. Chang, J. Wawrzynek, and R. W. Brodersen. BEE2: A high-end reconfigurable computing system. *IEEE Design and Test of Computers*, 22(2):114–125, Mar/Apr 2005.

[7] J. Chung, H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The Common Case Transactional Behavior of Multithreaded Programs. In *Proc. of the 12th Intl. Conf. on High-Performance Computer Architecture*, February 2006.

[8] 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: Proc. of the 11th Intl. Conf. on Architectural support for programming languages and operating systems*, pages 1–13, October 2004.

[9] 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 *Proc. of the 31st Intl. Symp. on Computer Architecture*, pages 102–113, June 2004.

[10] R. A. Hankins, G. N. Chinya, J. D. Collins, P. H. Wang, R. Rakvic, H. Wang, and J. P. Shen. Multiple instruction stream processor. In *ISCA '06: Proc. of the 33rd Intl. Symp. on Computer Architecture*, pages 114–127, 2006.

[11] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proc. of the 18th Ann. ACM SIGPLAN Conf. on Object-oriented programing, systems, languages, and applications*, pages 388–402, 2003.

[12] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proc. of the twenty-second Ann. Symp. on Principles of distributed computing*, pages 92–101, July 2003.

[13] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th Intl. Symp. on Computer Architecture*, pages 289–300, 1993.

[14] M. Ito and et. al. Sh-mobileg1: A single-chip application and dual-mode baseband processor. In *Conf. Record of Hot Chips 18*, 2006.

[15] S. Kaneko and et. al. A 600-mhz single-chip multiprocessor with 4.8-gb/s internal shared pipelined bus and 512-kb internal memory. *IEEE Journal of Solid-State Circuits*, 39(1):184–193, Jan. 2004.

[16] B. Lewis and D. J. Berg. *Multithreaded Programming with Pthreads*. Prentice Hall, 1998.

[17] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *19th Intl. Symp. on Distributed Computing*, September 2005.

[18] A. McDonald, J. Chung, H. Chafi, C. Cao Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on Chip-Multiprocessors. In *PACT '05: Proc. of the 14th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 63–74, September 2005.

[19] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-Based Transactional Memory. In *12th Intl. Conf. on High-Performance Computer Architecture*, February 2006.

[20] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *ISCA '05: Proc. of the 32nd Ann. Intl. Symp. on Computer Architecture*, pages 494–505, June 2005.

[21] M. F. Ringenburg and D. Grossman. Atomcaml: first-class atomicity via rollback. In *ICFP '05: Proc. of the tenth ACM SIGPLAN Intl. Conf. on Functional programming*, pages 92–104, 2005.

[22] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. A high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proc. of the eleventh ACM SIGPLAN Symp. on Principles and practice of parallel programming*, March 2006.

[23] N. Shavit and D. Touitou. Software transactional memory. In *Proc. of the 14th Ann. ACM Symp. on Principles of Distributed Computing*, pages 204–213, Ottawa, Canada, August 1995.

[24] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.

[25] S. Wee, J. Casper, N. Njoroge, Y. Teslyar, D. Ge, C. Kozyrakis, and K. Olukotun. A practical fpga-based framework for novel cmp research. In *FPGA '06: Proc. of the 2006 ACM/SIGDA 14th Intl. Symp. on Field programmable gate arrays*, 2007.

[26] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In M. Odersky, editor, *Proc. of the European Conf. on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 519–542. Springer-Verlag, 2004.

[27] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Intl. Symp. on Computer Architecture*, pages 24–36, June 1995.