NetTM: Faster and Easier Synchronization for Soft Multicores via Transactional Memory

Martin Labrecque and J. Gregory Steffan
University of Toronto, Canada
{martinl,steffan}@eecg.utoronto.ca

ABSTRACT

We propose NetTM: support for hardware *transactional memory* (HTM) in an FPGA-based soft multithreaded multicore that matches the strengths of FPGAs. We evaluate our system using the NetFPGA [6] platform and four network packet processing applications that are threaded and share memory. Relative to NetThreads [5], an existing two-processor four-way-multithreaded system with conventional lock-based synchronization, we find that adding HTM support (i) maintains a reasonable operating frequency of 125MHz with an area overhead of 20%, (ii) can transactionally execute lock-based critical sections with no software modification, and (iii) achieves 6%, 55% and 57% increases in packet throughput for three of four packet processing applications studied, due to reduced false synchronization.

Categories and Subject Descriptors

C.1.4 [**Processor architectures**]: Parallel Architectures;
C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems

General Terms

Design, Performance

1. INTRODUCTION

FPGA-based systems are increasingly used to implement larger and more complex systems-on-chip composed of multiple processor and acceleration cores that must synchronize and share data. While systems based on shared memory can ease communication between cores, they require correct synchronization (i.e. lock and unlock operations). Consequently, threads executing in parallel wanting to enter the same *critical section* (i.e. a portion of code that accesses shared data delimited by synchronization) at the same time will be serialized, thus losing the parallel advantage of such a system. Therefore designers face two important challenges: (i) writing parallel programs with manually inserted lock-based synchronization is error-prone and difficult to debug, and (ii) multiple processors need to share memory, communicate, and synchronize without serializing the execution.

Transactional memory (TM, see references in [4]) offers a potential solution to both challenges as it (i) offers an
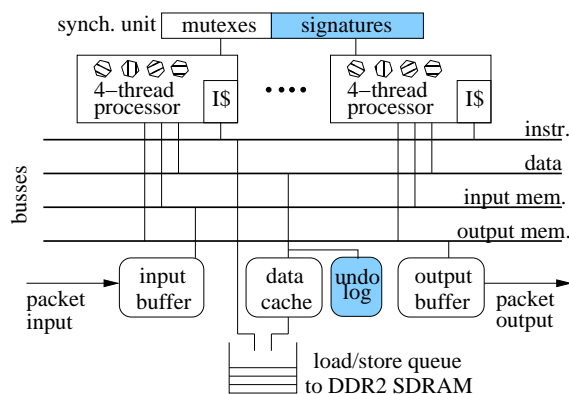
Figure 1: The NetTM architecture, which supports TM by extending NetThreads with signatures and an undo-log.

easier programming model for synchronization, and (ii) can reduce the contention on critical sections. With transactional execution, a programmer is free to employ coarser critical sections, spend less effort minimizing them, and not worry about deadlocks since a properly implemented TM system does not suffer from them. A TM system optimistically allows multiple threads inside a critical section—hence TM can improve performance when the parallel critical sections access independent data locations. To guarantee correctness, the underlying system dynamically monitors the memory accesses of each transaction (the *read set* and *write set*) and detects *conflicts* between them. While TM can be implemented purely or partly in software (STM), an FPGA-based system can be extended to support TM in hardware (HTM) with much lower performance overhead than an STM. Our goal is to use HTM to improve synchronization for FPGA-based multicores, by which we mean interconnected processor or accelerator cores that synchronize and share memory. There are many known methods for implementing HTM for an ASIC multicore processor, although they do not necessarily map well to FPGA-based processors. In this paper we introduce NetTM, which adds support for HTM to the NetThreads [5] system, an FPGA-based multithreaded multicore for high-throughput packet processing.

2. BASE ARCHITECTURE: NETTHREADS

In this work our starting point architecture for comparison is the NetThreads [5] multithreaded multicore architecture that supports only lock-based critical sections, shown in Figure 1. Each NetThreads processor is a single-issue, in-order, 5-stage pipelined processor. In a single core, instructions from four hardware threads are issued in a round-robin fash-

ion to hide pipeline hazards and cache miss latency [5]. The SDRAM controller services a merged load/store queue of up to 64 entries in-order; since this queue and the data cache are shared by all processors, they serve as a single point of serialization and memory consistency, hence threads need only block on pending loads but not stores. Each processor has a 16KB instruction cache. Packets are received into the 10-slot input buffer memory, and written out via the output buffer. There is also a shared data cache capable of 32-bit line-sized data transfers with the DDR2 SDRAM controller. All three of these memories are 16KB. To implement lock-based synchronization, NetThreads provides a synchronization unit containing 16 hardware mutexes (sufficient for our applications). In the NetThreads ISA, each lock/unlock operation specifies a unique identifier, indicating one of these 16 mutexes.

Benchmark Applications For 2009, both Altera and Xilinx reported that communications comprised 44% of their net revenues. NetThreads targets network packet processing applications, in particular those that require deeper packet inspection. We focus on *stateful* applications—i.e., applications in which shared, persistent data structures are modified during the processing of most packets. To take full advantage of the software programmability of our processors, our focus is on the control-flow intensive applications described in detail in earlier work [5]. Note that each of these is a full application, with significant numbers of loads and stores, designed for the NetFPGA platform and integrated with a host.

3. PROGRAMMING NETTM

Specifying Transactions TM semantics imply that any transaction will appear to have executed atomically with respect to any other transaction. To denote the start and end of a transactional critical section, NetTM uses the same instruction API as the lock-based synchronization for NetThreads—i.e., lock(ID) can mean "start transaction" and unlock(ID) can mean "end transaction". Hence existing programs need not be modified, since NetTM can use existing synchronization in the code and simply interpret critical sections as transactional. We next describe how the lock identifier, ID, is interpreted by NetTM.

Locks vs Transactions NetTM supports both lock-based and TM-based synchronization, since a code region's access patterns can favor one approach over the other. For example, lock-based critical sections are necessary for I/O operations since they cannot be undone in the event of an aborted transaction: specifically, for processor initialization, to protect the sequence of memory-mapped commands leading to sending a packet, and to protect the allocation of output memory (Figure 1). We use the software identifier associated with lock/unlock operations to distinguish whether a given critical section should be executed via a transaction or via traditional lock-based synchronization: this mapping of the identifiers is provided by the designer as a parameter to the hardware synchronization unit (Figure 1). When using locks, a programmer would typically need to worry about which identifier encloses accesses to which shared variables. NetTM simplifies the programmer's task when using transactions: NetTM enforces the atomicity of all transactions regardless of the lock identifier value. Therefore only one identifier can be designated to be of the transaction type, and doing so frees the remaining identifiers/mutexes to be used as traditional locks. However, to support legacy software, a designer is also free to designate multiple identifiers to be of the transaction type.

4. IMPLEMENTING NETTM

Version Management Version management refers to the method of segregating transactional modifications from other transactions and regular memory. For a simple HTM, the two main options for version management are *lazy* [2] versus *eager* [9]. In an eager approach, writes modify main memory directly and are not buffered—therefore any conflicts must be detected before a write is performed. To support rollback for aborts, a backup copy of each modified memory location must be saved in an *undo-log*. Hence when a transaction aborts, the undo-log is flushed to regular memory, and when a transaction commits, the undo-log is discarded. A major benefit of an eager approach is that reads proceed unhindered and can directly access main memory, and hence an undo-log is much simpler than a write-buffer since the undo-log need only be read when flushing to regular memory on abort. Because eager schemes modify regular memory directly: (i) they cannot support multiple transactions writing to the same location (this results in a conflict), (ii) conflict detection must be performed on every memory access, (iii) aborts are slow because the undo-log must be flushed to regular memory, and (iv) commits are fast because the undo-log is simply discarded.

After serious consideration of the lazy and eager approaches, we concluded that eager version management was the best match to FPGA-based systems such as NetTM for several reasons. First, while requiring a similar minimum amount of storage, a write buffer is necessarily significantly more complex than an undo-log since it must support fast reads via indexing and a cache-like organization. Our preliminary efforts found that it was extremely difficult to create a write-buffer with single-cycle read and write access. To avoid replacement from a cache-organized write-buffer (which in turn must result in transaction stall or abort), it must be large or associative or both, and these are both challenging for FPGAs. Second, an eager approach allows spilling transactional modifications from the shared data cache to the next level of memory (in this case off-chip), and our benchmarks exhibit large write sets. Third, via simulation, we observed that disallowing multiple writers to the same memory location(s) (a limitation of an eager approach) resulted in only a 1% increase in aborts for our applications in the worst case. Fourth, we found that transactions commit in the common case for our applications, and an eager approach is fastest for commit.

Conflict Detection A key consequence of our decision to implement eager version management is that we must be able to detect conflicts with every memory access; hence to avoid undue added sources of stalling in the system, we must be able to do so in a single cycle. This requirement led us to consider implementing conflict detection via *signatures*, which are bit-vectors that track the memory locations accessed by a transaction via hash indexing, with each transaction owning two signatures to track its read and write sets. Signatures can represent an unbounded set of addresses,
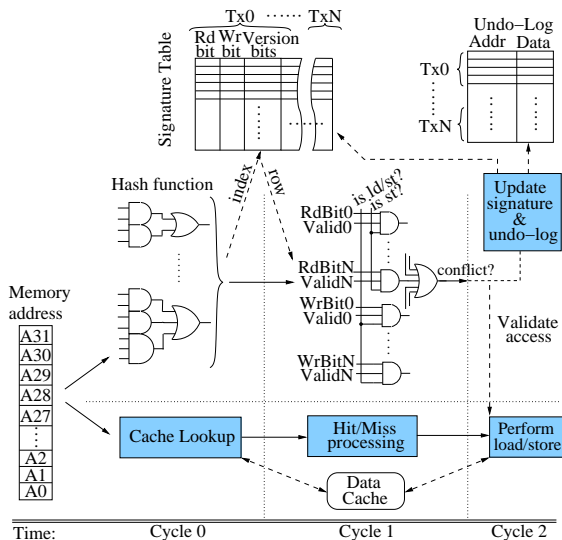
Figure 2: Integration of conflict detection hardware with the processor pipeline for a memory access.

they decouple conflict detection from version management, and provide an opportunity to capitalize on the bit-level parallelism of FPGAs. Previous work [4] explored the design space of signature implementations for an FPGA-based two-processor system (i.e., for two total threads), and proposed a method for creating application-specific hash functions to reduce signature size without impacting their accuracy. We extended this prior scheme to support a multithreaded multicore for eight total threads.

Signature Table   In contrast with prior work [4], in NetTM we store signatures in BRAMs. As shown in Figure 2, the hash function indexes a row of the BRAM, and we map the corresponding read and write bits for every transaction/thread-context on the same memory row. Therefore with one BRAM access, we can read all of the read and write signature bits for a given address for all transactions in parallel. Note in Figure 2 that memory instructions undergo an extra pipeline stage (*cycle 1*) to allot for the hashing and conflict detection. A challenge is that we must clear the signature bits for a given transaction when it commits or aborts, and it would be too costly to visit all of the rows of the BRAM to do so. Instead we add a version number per transaction (incremented on commit or rollback), that we can compare to a register holding the true version number of the current transaction for that thread context. Comparing version numbers produces a `Valid` signal (Figure 2) that is used to ignore the result of comparing signature bits when appropriate. We clear signature bits lazily: for every memory reference, a row of the signature table is accessed, and we clear the corresponding signature bits for any transaction with mismatching version numbers. This lazy-clear works well in practice, although it is possible that the version number may completely wrap-around before there is an intervening memory reference to cause a clear, resulting in a false conflict (which hurts performance but not correctness). We are hence motivated to support version numbers that are as large as possible.

Our target device has 36-bit-wide BRAMs, and we determined experimentally that a signature table composed of

at most two block RAMs could be integrated in the NetTM CPU pipeline while preserving the 125MHz target frequency. We could combine the BRAMs horizontally to allow larger version numbers, or vertically to allow more signature bits; we determined experimentally that the vertical option produced the fewest false conflicts. Hence in NetTM, each BRAM row contains a read bit, a write bit, and two version bits (four bits total) for each of eight transactions/thread-contexts, for a total of 32 bits (slightly under-utilizing the available 36 bit width). For our applications, we implement signatures of maximum length ranging from 618 to 904 bits (limited by the speed of the hash function).

Undo-Log   The undo-log is implemented as a single physical structure logically divided equally for each thread context. On a transaction commit, a per-thread undo-log is cleared in one cycle by resetting a write-pointer. On a transaction abort, the undo-log requests exclusive access to the shared data cache, and flushes its contents to the cache in reverse order. This flush is performed atomically with respect to any other memory access, although processors can continue to execute non-memory-reference instructions during an undo-log flush. We buffer data in the undo-log at a word granularity because that matches our conflict detection resolution. In NetTM, the undo-log must be provisioned to be large enough to accommodate the longest transactions. We save undo-log capacity via a filtering mechanism that ignores the uninitialized portion of the stack.

5. EVALUATING NETTM

In this section, we report the maximum sustainable packet rate for a given application with the configuration in Table 1 as the packet rate with 90% confidence of not dropping any packet over a five-second run—thus our results are conservative given that network appliances are typically allowed to drop a small fraction of packets.

Resource Utilization   In total, with two four-threaded processors, NetTM consumes 32 more BRAMs than Net-Threads, so its BRAM utilization is 71% (161/232) compared to 57% (129/232) for NetThreads. The additional BRAMs are used as follows: 2 for the signature bit vectors, 4 for the log filters (to save last and checkpoint stack pointers) and 26 for the undo-log (1024 words and addresses for 8 threads). NetThreads consumes 18980 LUTs (out of 47232, i.e. 40% of the total LUT capacity) when optimized with high-effort for speed; NetTM design variations range from 3816 to 4097 additional LUTs, an overhead of roughly 21% over NetThreads.

NetTM Throughput   NetTM improves packet throughput by 57%, 6% and 55% for `Classifier`, `NAT`, and `UDHCP` re-
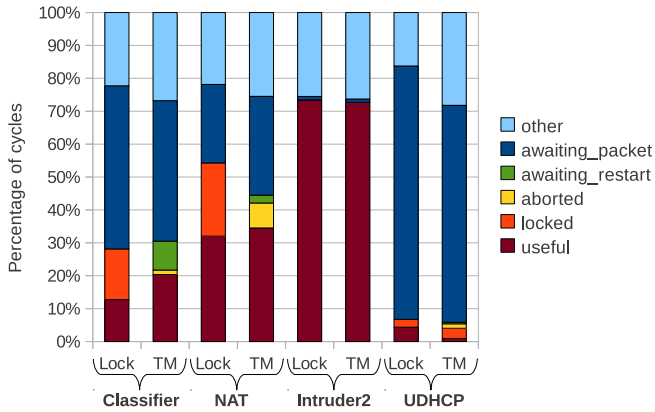
Figure 3: Breakdown (from simulation) of cycles spent: on pipeline hazards, and data and instruction misses (*other*); de-scheduled or busy-waiting for a packet (*awaiting_packet*); waiting to re-start a transaction (*awaiting_restart*); on instructions canceled due to transaction aborts (*aborted*); de-scheduled waiting for a lock (*locked*); or executing useful instructions (*useful*).

spectively, by exploiting the optimistic parallelism available in critical sections. The TM speedup is the result of reduced time spent awaiting locks, but moderated by the number of conflicts between transactions and the time to recover from them. `Classifier` has occasional long transactions that do not always conflict, providing an opportunity for reclaiming the corresponding large wait times incurred with locks. Similarly, `UDHCP` has a significant fraction of read-only transactions that do not conflict. `NAT` has a less pronounced speedup because of shorter transactions and higher abort rates. For `Intruder2`, despite having a high average commit rate, TM results in an 8% lower throughput due to bursty periods of repeated transaction aborts, leading to a reduced parallelism and packet rate.

Processor utilization To provide a detailed view of processor activity, we profile the execution of our applications using our cycle-accurate simulator. While it is unable to faithfully model the timing of DRAM refreshes and packet arrivals, our simulator gives predicted speedups of 1.6 for `Classifier`, 1.1 for `NAT`, 1.6 for `UDHCP` and 1.0 for `Intruder2`—i.e., a maximum error of 8% and a mean error of 4%. Figure 3 gives the breakdown of how time (in cycles) is spent for each application for both NetThreads and NetTM. `NAT` and `Classifier` experience a significant fraction of cycles waiting for locks in NetThreads, e.g. 22% for `NAT`. In our 4-way multithreaded cores, this is indicative of high lock contention because the thread scheduler could hide `locked` cycles, instruction hazards (included in the *other* category) and wait times for packets and transaction restarts, if there was any other thread context to schedule on those cycles. Compared to `Classifier`, `NAT` has a smaller speedup due to a considerable fraction of aborted work. Also, we can see that `UDHCP` spends most of the time waiting for packets despite functioning at its fixed maximum packet rate, indicating the highest variability in critical section size across our benchmarks. Comparatively, `Intruder2` has the highest CPU utilization, meaning that wait times for synchronization are smaller and data dependences across

transactions will be harmful because of transaction aborts. This case demonstrates that TM isn't necessarily the best option for every region of code or application.

6. RELATED WORK

The goal of most previous FPGA implementations of HTM [1, 7, 8] was to prototype ASIC attempts at TM—as opposed to targeting the strengths of an FPGA as a final product. To provide a low-overhead implementation, our work distinguishes itself in the type of TM that we implement and in the way that we perform conflict detection. To track transactional speculative state, prior FPGA implementations [1,3,8] use (i) extra bits per line in a private cache per thread or in a shared cache, and (ii) *lazy version management* (i.e., regular memory is modified only upon commit), and (iii) *lazy conflict detection* (i.e., validation is only performed at commit time). These approaches are not a good match for product-oriented FPGA-based systems because of the significant cache storage overhead required. Rather than using off-the-shelf soft cores requiring the programmer to explicitly mark each transactional access in the code as in other work [1,7,8], in NetTM we integrate TM with each soft processor pipeline and automatically and seamlessly handle loads and stores within transactions or lock-based critical sections appropriately.

7. CONCLUSIONS

We have shown that NetTM, an eager TM with application-specific signatures and contention management, matches the strengths and limitations of an FPGA, and can be integrated into a multithreaded processor pipeline without impacting clock frequency by adding a pipeline stage. NetTM makes synchronization (i) easier, by allowing more coarse-grained critical sections and eliminating deadlock errors, and (ii) faster, by exploiting the optimistic parallelism available in many concurrent critical sections, especially for software packet processing. For multithreaded applications that share and synchronize, we demonstrated that NetTM can improve throughput by 6%, 55%, and 57% over a locks-only system, but that TM is inappropriate for some applications where the cost of occasional aborts negates the wait times reclaimed.

8. REFERENCES
[1] S. Grinberg and S. Weiss. Investigation of transactional memory using FPGAs. In *EEEI*, Nov. 2006.
[2] 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. *SIGARCH Comput. Archit. News*, 32(2):102, 2004.
[3] C. Kachris and C. Kulkarni. Configurable transactional memory. In *FCCM*, 2007.
[4] M. Labrecque, M. Jeffrey, and J. G. Steffan. Application-specific signatures for transactional memory in soft processors. In *ARC*, 2010.
[5] M. Labrecque and J. G. Steffan. The case for hardware transactional memory in software packet processing. In *ANCS*, 2010.
[6] J. W. Lockwood, N. McKeown, G. Watson, et al. NetFPGA - an open platform for gigabit-rate network switching and routing. In *MSE*, 2007.
[7] R. Teodorescu and J. Torrellas. Prototyping architectural support for program rollback using FPGAs. *FCCM*, 2005.
[8] S. Wee, J. Casper, N. Njoroge, et al. A practical FPGA-based framework for novel CMP research. In *FPGA*, 2007.
[9] L. Yen, J. Bobba, M. R. Marty, et al. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA*, 2007.