The Case for Hardware Transactional Memory in Software Packet Processing

Martin Labrecque and J. Gregory Steffan
Department of Electrical and Computer Engineering, University of Toronto
{martinl,steffan}@eecg.toronto.edu

ABSTRACT

Software packet processing is becoming more important to enable differentiated and rapidly-evolving network services. With increasing numbers of programmable processor and accelerator cores per network node, it is a challenge to support sharing and synchronization across them in a way that is scalable and easy-to-program. In this paper, we focus on parallel/threaded applications that have irregular control-flow and frequently-updated shared state that must be synchronized across threads. However, conventional lock-based synchronization is both difficult to use and also often results in frequent conservative serialization of critical sections. Alternatively, we propose that *Transactional memory* (TM) is a good match to software packet processing: it both (i) can allow the system to optimistically exploit parallelism between the processing of packets whenever it is safe to do so, and (ii) is easy-to-use for a programmer. With the NetFPGA [1] platform and four network packet processing applications that are threaded and share memory, we evaluate hardware support for TM (HTM) using the reconfigurable FPGA fabric. Relative to NetThreads [2], our two-processor four-way-multithreaded system with conventional lock-based synchronization, we find that adding HTM achieves 6%, 54% and 57% increases in packet throughput for three of four packet processing applications studied, due to reduced conservative serialization.

Categories and Subject Descriptors

C.1.4 [**Processor architectures**]: Parallel Architectures; C.1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]: Multiple-instruction-stream, multiple-data-stream processors (MIMD); C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems

General Terms

Algorithms, Design, Performance

ANCS '10, 25-OCT-2010, San Diego CA, USA

Keywords

Transactional memory, packet processing

1. INTRODUCTION

Modern network appliance programmers are faced with larger and more complex systems-on-chip composed of multiple processor and acceleration cores that must synchronize and share data, while meeting the expectation that performance should scale with the number of compute threads [3, 4]. Programming is a particular challenge for stateful applications with irregular control-flow. We demonstrate in this paper that such applications are more suitable to a *run-to-completion* model of execution, where a single thread performs the complete processing of a packet from start to finish. Hence the programmer writes a single program that is executed on each packet. For performance, the system must be able to execute multiple instances of the program in parallel. However, the processing of different packets is not always independent, and the application must maintain some shared state.

While systems based on shared memory can ease the orchestration of sharing and communication between cores, they require the careful use of 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) will be serialized, thus losing the parallel advantage of such a system. Hence designers face two important challenges: (i) multiple processors need to share memory, communicate, and synchronize without serializing the execution, and (ii) writing parallel programs with manually inserted lock-based synchronization is error-prone and difficult to debug.

Transactional memory (TM) [5, 6] offers a potential solution to both challenges as it (i) can reduce false contention on critical sections, and (ii) offers an easier programming model for synchronization. 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. 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. To guarantee correctness, the underlying system dynamically monitors the memory access locations of each transaction (the *read set* and *write set*) and detects *conflicts* between them. While TM can be implemented purely in software (STM), a

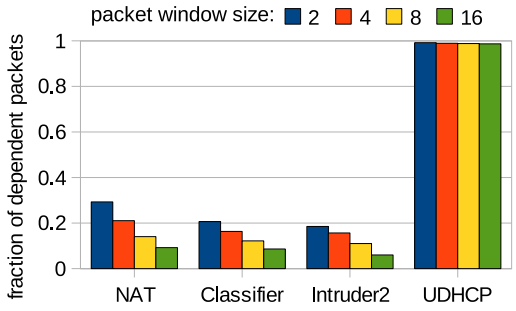


Figure 1: Average fraction of conflicting packet executions for windows of 2 to 16 consecutive packets.

hardware implementation (HTM) offers significantly lower performance overhead. The key question is: how amenable to optimistic transactional execution is packet processing on a multicore—i.e., on a platform with interconnected processor or accelerator cores that synchronize and share memory?

1.1 The Potential for Optimistic Parallelism

Although this depends on the application, for many applications only the processing of packets belonging to the same *flow* (i.e., a stream of packets with common application-specific attributes, such as addresses, protocols, or ports) results in accesses to the same shared state. In other words, there is often parallelism available in the processing of packets belonging to independent flows. Melvin et al. [7] show that for two NLANR packet traces the probability of having at least two packets from the same flow in a window of 100 consecutive preceding packets is approximately 20% and 40%. Verdú et al. [8] further show that the distance between packets from the same flow increases with the amount of traffic aggregation on a link, and therefore generally with the link bandwidth in the case of wide area networks.

Our position in this paper is that there is optimistic parallelism available in stateful, control-flow-intensive packet processing applications, and that HTM support (assuming an optimistic HTM) can exploit this parallelism and hence be both (i) better-performing, and (ii) easier-to-program, than lock-based critical sections. However, in this paper we offer evidence only for performance, but rely on prior work [9] to support the ease-of-programming claim. To demonstrate the potential for optimistic parallelism in our benchmark applications, we profiled them using our full-system simulator (both are described in more detail later). For now what we are interested in is how often the processing of packets has a conflict—i.e. for two threads each processing a packet, their write sets intersect or a write set intersects a read set. In Figure 1, we show the average fraction of packets for which their processing conflicts for varying windows of 2 to 16 packets. For three of our applications, `NAT`, `Classifier`, and `Intruder2`, the fraction of conflicting packet-executions varies from around 20% to less than 10% as the window considered increases from 2 to 16 packets, indicating two important things: first, that conventional synchronization for critical sections in these applications would be overly-conservative 80% to 90% of the time, and second that there is hence a strong potential for optimistic synchronization for these applications. For `UDHCP`, our profile indicates that nearly all packet-executions conflict. In reality `UDHCP` contains several critical sections, some that do nearly always conflict, but many others that do not conflict—hence the potential for optimistic parallelism exists even for `UDHCP`.

1.2 Contributions

In this paper we describe, implement, and evaluate four threaded, stateful, control-flow-intensive networking applications that share memory and synchronize, a real implementation of an HTM system called NetTM on the NetFPGA [1] platform, and compare with a two-processor, eight-threaded base system that implements only conventional lock-based synchronization (called NetThreads [2]). In this context we make the following contributions: (i) we motivate the need for software packet processing, i.e., a run-to-completion-on-shared-memory programming model for packet processing applications that are stateful and control-flow intensive; (ii) we demonstrate the benefits of exploiting optimistic parallelism in packet processing on NetTM; (iii) we show that NetTM outperforms flow-affinity scheduling, but that NetTM could be extended to exploit a flow affinity approach.

2. SOFTWARE PACKET PROCESSING

Many packet processing applications must process packets at line rate, and to do so they must scale to make full use of a system composed of multiple processors and accelerators cores. Given the broad and varied use of packet processing, in this section we clarify the application-types and bandwidths for which software packet processing is suitable, and what challenges emerge when programming these applications in a multicore environment.

2.1 Application Types

We divide network processing applications into three categories:

1) Basic Common packet processing tasks performed in small office/home office network equipment include learning MAC addresses, switching and routing packets, and performing port forwarding, port and IP filtering, basic QoS, and VLAN tagging. These functions are typically limited to a predefined number of values (e.g. 10 port forwarding entries) such that they can be implemented in an ASIC switch controller chip, without the need for software programmability.

2) Byte-Manipulation A number of network applications, in particular cryptography and compression routines, apply a regular transformation to most of the bytes of a packet. Because these workloads often require several iterations of specialized bit-wise operations, they benefit from hardware accelerators such as the specialized engines present in network processors [10], modern processors (e.g. Intel AES instruction extensions), and off-the-shelf network cards; they also generally do not require the programmability of software.

3) Control-Flow Intensive Network packet processing is no longer limited solely to routing, with many applications that require deep packet inspection becoming

increasingly common. Some applications, such as storage virtualization and server load balancing, are variations on the theme of routing that reach deeper into the payload of the packets to perform content-based routing, access control, and bandwidth allocation. Other applications have entirely different computing needs such as the increasingly complex firewall and bandwidth management systems that must recognize applications, scan for known malicious patterns, and recognize new attacks among a sea of innocuous packets. Furthermore, with the increasing use of application protocols built on HTTP and XML, the distinction between payload and header processing is slowly disappearing. Hence in this paper we focus on such control-flow intensive applications.

2.2 The Need for Simpler Synchronization

The Internet has witnessed a transformation from static web pages to social networking and peer-to-peer data transfers. This transformation of user behavior patterns requires network connectivity providers to constantly and proactively adapt their services to adequately provision and secure their infrastructure. As networking requirements evolve constantly, many network equipment vendors opt for network processors to implement functions that are likely to change over time. Because many network protocols are now considered outdated, there is even a desire to have vendors open the hardware to accept user/researcher code [11]. However, once faced with board-specific reference code, programmers are often hesitant to edit it, in part due to the challenge of modifying the synchronization and parallelization mechanisms in those carefully tuned multicore programs.

With multicore processors becoming commodity, general purpose processors are closing the performance gap with network processors for network-edge applications [12] fueling an increased industry use of open-source software that can turn an off-the-shelf computer into a network device; examples include the Click Modular router [13], Snort [14], Quagga [15], and the XORP project [16]. Those applications are often coded as a single thread of computation with many global data structures that are unsynchronized—hence porting them to multicore is a substantial challenge when performance depends on constructing finer-grained synchronized sections. There is therefore a need for simpler synchronization mechanisms to support control-flow intensive programs.

2.3 Fast- vs Slow-Path Processing

Network equipment typically connects multiple network ports on links with speeds that span multiple orders of magnitude: 10Mbps to 10Gbps are common physical layer data rates. While local area networks can normally achieve a high link utilization, typical transfer speeds to and from the Internet are on the order of megabits per second [17] as determined by the network organization and utilization between the Internet service providers.

The amount of processing performed on each packet will directly affect the latency introduced on each packet and the maximum allowable sustained packet rate. The amount of buffering available on the network node will also help mitigate bursts of traffic and/or variability in the amount of processing. For current network appliances that process an aggregate multi-gigabit data stream across many ports, there is typically a division of the processing in *data plane* (a.k.a. fast path) and *control plane* (a.k.a. slow path) operations. The data plane takes care of forwarding packets at full speed based on rules defined by the control plane which only processes a fraction of the traffic (e.g. routing protocols). Data plane processing is therefore very regular from packet to packet and deterministic in the number of cycles per packet. Data plane operations are typically implemented in ASICs on linecards; control plane operations are typically implemented on a centralized supervisor card. The control plane, often software programmable and performing complex control-flow intensive tasks, still has to be provisioned to handle high data rates. For example, the Cisco SPP network processor in the CRS-1 router is designed to handle 40Gbps [18]. On smaller scale network equipment (eg., a commodity desktop-based router at the extreme end of the spectrum), the two planes are frequently implemented on a single printed circuit board either with ASICs or programmable network processors or a combination of both. In that case, the amount of computation per packet has a high variance, as the boundary between the fast and slow path is often blurred.

In this paper, we focus on complex packet processing tasks that are best-suited to a software implementation, since a complete hardware implementation would be impractical. Our benchmark applications therefore target the control plane, rather than the data plane of multi-gigabit machines.

3. BENCHMARK APPLICATIONS

Prior network processing benchmark suites [19, 20] are dominated by stateless kernels that emulate isolated packet processing routines and fall into the first two categories in Section 2.1 above. For those kernels that represent header processing workloads, the amount of instruction-level parallelism (ILP) can exceed several thousand instructions [20]. Because such tasks are best addressed by SIMD processors or custom ASICs, in this paper we instead focus on control-flow intensive applications where the average ILP is only five [21]. However, there is a lack of benchmark suites representing applications of that kind, i.e. that are threaded and synchronized.

We have developed the four control-flow intensive applications detailed in Table 1. Because our evaluation platform (Section 5) does not have an operating system, all the low-level protocol-handling is inlined directly into our programs. To implement time-stamps and time-outs, we use a hardware system clock. Table 1 also describes the nature of the parallelism in each application, and Table 2 reports statistics on the dynamic accesses per critical section for each application. Note that the critical sections comprise significant numbers of loads and stores with a high disparity between the average and maximum values, showing that our applications are stateful and irregular in terms of computations per packet. As indicated in Table 1, the `NAT` application maintains shared and per-flow statistics along with a dynamically-allocated address translation table, and the other benchmarks inspect deeply into the payload of packets. We next analyze the representative traits of each application and generalize them to other control-flow intensive network applications, particularly with respect to packet ordering, data parallelism, and synchronization.

Packet Ordering In a network device, there is typically no requirement to preserve the packet ordering across flows from the same or different senders: they are interpreted as unrelated. For a given flow, one source of synchronization is often to preserve packet ordering, which can mean: i) that packets must be processed in the order that they arrived; and/or ii) that packets must be sent out on the network in the order that they arrived. The first criteria is often relaxed because it is well known that packets can be reordered in a network [22], which means that the enforced order is optimistically the order in which the original sender created the packets. The second criteria can be managed at the output queues and does not usually affect the core of packet processing. For our benchmark applications in Table 1, while we could enforce ordering in software, we allow packets to be processed out-of-order because our application semantics allow it. Enforcing flow ordering in our system would require to support many more flows to stress our processors at maximum utilization and would therefore require significantly more packet buffering. In such a setting with an increased number of independent flows, we expect that a transactional memory system could extract more parallelism across the flows and its throughput would thus be further increased.

Data Parallelism Packet processing typically implies tracking flows (or clients for `UDHCP`) in a database, commonly implemented as a hash-table or direct-mapped array. The size of the database is bounded by the size of the main memory—typically larger than what can be contained in any single data cache—and there is usually little or a very short-term reuse of incoming packets. Because a network device executes continuously, a mechanism for removing flows from the database after some elapsed time is also required. In *stateful* applications, i.e. applications where shared, persistent data structures are modified during the processing of most packets, there may be variables that do not relate directly to flows (e.g. a packet counter). Therefore, it is possible that the processing of packets from different flows access the same shared data and therefore the processing of those packets in parallel may conflict. Also, for certain applications, it may possible to extract intra-packet parallelism (e.g. parallelization of a loop), however those cases are rare because they are likely to leave some processors underutilized so we do not consider them further. Whenever shared data is accessed by concurrent threads, those accesses must be synchronized to prevent data corruption.

Synchronization To increase parallelism, implementing finer-grain synchronization is not always feasible since repeatedly entering and exiting critical sections will likely add significant overhead. For example, `NAT` and `Classifier` have a significant fraction of their code synchronized because there is an interaction between the hash table lock and the flow lock: a thread cannot release the lock on the hash table prior to acquiring a lock on a flow descriptor to ensure that the flow is not removed in the mean time. Mechanisms for allowing coarser-grained sections while preserving performance are therefore very desirable for packet processing.

4. EXPLOITING PARALLELISM

Because nearly all modern network processors are multi-core, the mapping of the application to those cores is often specific to the underlying processor architecture and is also a trade-off between performance and ease-of-programming. In this section we quantitatively justify our choice of the run-to-completion model for extracting parallelism from control-flow-intensive, stateful software packet processing applications. Run-to-completion is one of the three common execution models for exploiting parallelism, illustrated in Figure 2, that we describe next.

Table 2: Dynamic Accesses per Critical Section

Benchmark	Loads		Stores	
	mean	max	mean	max
`NAT`	114	739	42	98
`Classifier`	2433	67873	64	573
`Intruder2`	95	593	16	182
`UDHCP`	61	3504	11	36

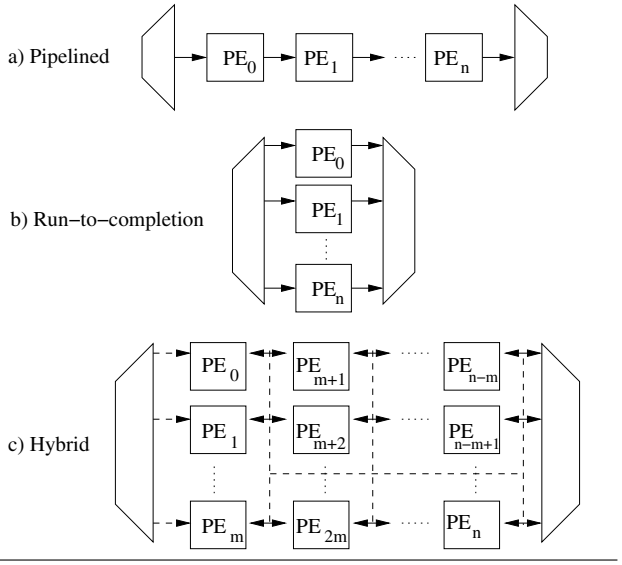


Figure 2: Parallelization models.

4.1 Pipelining

Pipelining is widely used as the underlying parallelization method [3, 12, 29, 30], most often to avoid the difficulty of managing locks: there is no need to synchronize if writes to shared data structures are done in a single pipeline stage[1], even if there are multiple readers. As shown in Figure 2(a), the pipeline model allows a sequence of operations to be performed on a packet by spreading the operations across cores. Programs are best suited for pipelining if they are composed of data-independent and self-contained kernels executing in a stable computation pattern with communication at the boundaries [31]. However, efficiently balancing the pipeline stages and the amount of communication between them to maximize multiprocessor utilization is complicated and often not possible for complex applications. Even if an acceptable result is obtained, this difficult process must be repeated when the application is updated or the code is ported to a different processor architecture. Furthermore, handling packets that require varying amounts of computation or breaking down frequent

[1]Assuming the writes can be flushed atomically.

Table 1: Benchmark Applications

	Description	Critical Sections	Input packet trace
`Classifier`	Performs a regular expression matching on TCP packets, collects statistics on the number of bytes transferred and monitors the packet rate for classified flows to exemplify network-based application recognition. In the absence of a match, the payloads of packets are reassembled and tested up to 500 bytes before a flow is marked as non-matching. As a use case, we configure the widely used PCRE matching library [23] (same library that the popular Snort [14] intrusion detection/prevention system uses) with the HTTP regular expression from the "Linux layer 7 packet classifier" [24].	Has long transactions when regular expressions are evaluated; exploits parallelism across flows stored in a global synchronized hash-table.	Publicly available packet trace from 2007 on a 150Mbps trans-Pacific link (the link was upgraded from 100Mbps to 150Mbps on June 1, 2007) [25]. HTTP server replies are added to all packets presumably coming from an HTTP server to trigger the classification.
`NAT`	Exemplifies network address translation by rewriting packets from one network as if originating from one machine, and appropriately rewriting the packets flowing in the other direction. As an extension, `NAT` collects flow statistics and monitors packet rates.	Exhibits short transactions that encompass most of the processing; exploits parallelism across flows stored in a global synchronized hash-table.	Same packet trace as `Classifier`.
`Intruder2`	Network intrusion detection [26] modified for packetized input and re-written to have array-based reassembly buffers to avoid the overhead of queues, lists and maps that also reduced the effectiveness of signatures due to the large amount of `malloc()/free()` calls [27].	Has two synchronization phases: first a per-flow lock is acquired and released to allow processing each packet individually, then most of the computation is performed on reassembled messages before the per-flow variables are modified again under synchronization.	256 flows sending random messages of at most 128 bytes, broken randomly in at most 4 fragments, containing 10% of 'known attacks'. The fragments are shuffled with a sliding window of 16 and encapsulated in IP packets.
`UDHCP`	Derived from the widely-used open-source DHCP server. As in the original code, leases are stored in a linearly traversed array and IP addresses are leased after a ping request for them expires, to ensure that they are unused.	Periodic polling on databases for time expired records results in many read-dominated transactions as seen in Table 2. Has high contention on shared lease and awaiting-for-ping array data structures.	Packet trace modeling the expected DHCP message distribution of a network of 20000 hosts [28].

accesses to large stateful data structures (such as routing tables) to ensure lock-free operation is impractical in a pipeline with many stages.

Because a number of network processors implement the pipeline model [33] with the promise of extracting parallelism while being lock-free, we must justify our choice of a different model (run-to-completion). For this purpose, we use benchmarks from NetBench [32][2], and our stateful benchmarks running on a single thread. As Netbench's applications require a number of system and library calls, they cannot be ported easily to our NetTM embedded target (Section 5), so we instead record execution traces using the PIN tool [34]. We only monitor the processing for each packet and ignore the packet and console I/O routines.

As seen in Figure 3(a), our four applications span the spectrum of latency variability (i.e. jitter) per packet that is represented by the NetBench benchmarks. `Route` and `ipchain` have a completely deterministic behavior (no variability), while table lookup `tl` and the regular expression matching `Classifier` have the most variation across packets. Considering that for those applications the amount of computation can be more than doubled depending on the packet, we conclude that they are less amenable to pipelining. Even if re-circulating a packet from the end to the beginning of a pipeline were effective at mitigating this huge variation in latency [33], we would also have to effectively divide the packet processing into pipeline stages.

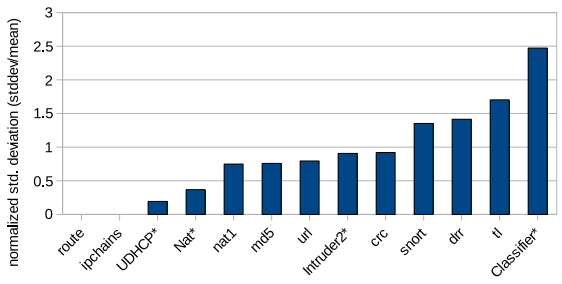
To emulate pipelining, we employ a previously-proposed graph-clustering method that greedily clusters instructions with the highest control and data flow affinity [35] to eliminate cyclic dependences and minimize communication between pipeline stages. Since NetTM has 8 threads, we cluster the instructions into 8 pipeline stages based on the profile information[3]. In all benchmarks in Figure 3(b), clustering the code into pipeline stages leads to significant load imbalance. `url` has the largest pipeline imbalance (i.e. the rate of the pipeline is 7.9 times slower than the average rate of all the stages) because of the clustering of the Boyer-Moore string search function in a single pipeline stage. Even `route` which has a deterministic execution (Figure 3(a)) has load imbalance because of the clustering of the checksum routine in a single longer-latency pipeline stage and `ipchains` has similar problems. While hardware accelerators could be used to accelerate checksum operations, a programmer cannot rely on them to balance the latency of arbitrary code in stages. To get a better load balance, a programmer would replicate the slowest stages and move to the hybrid or run-to-completion model, and add synchronization around stateful data structures.

4.2 Run-to-Completion and Hybrid

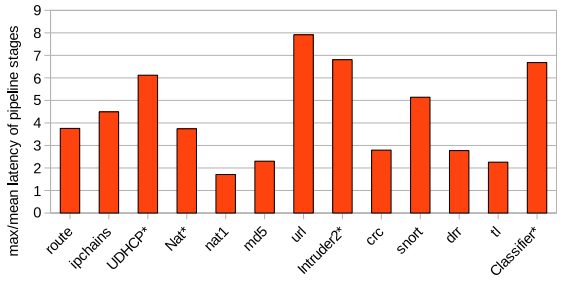
The model in Figure 2(b) refers to the conventional single-threaded method of writing a program where different processors process packets from beginning-to-end by executing the same program. Different paths in the program will exercise different parts of the application on different threads, which do not execute in lock-step. The programming is therefore intuitive but typically requires the addition of locks to protect shared data structures and of coherence mechanisms when shared data can be

[2]Except `dh` which is not packet based nor `ssl` because of its inlined console output.

[3]Control-flow and data dependences are based on the profiled basic-blocks.



(a) Normalized variability in packet processing latency with a single thread (stddev/mean latency).



(b) Imbalance in pipeline stages with an emulated pipeline of 8 threads (max stage latency/mean).

Figure 3: Single-threaded computational variability and load imbalance in an emulated pipeline for NetBench [32] benchmarks and our benchmarks (marked with a *).

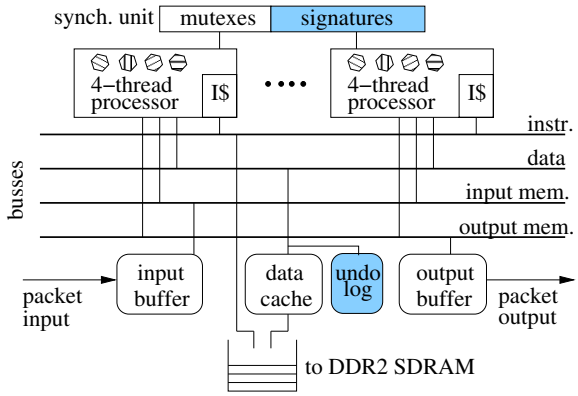


Figure 4: The NetThreads architecture, currently with two processors. NetTM supports TM by extending NetThreads, mainly with signatures and an undo-log.

Table 3: On-chip memories.

Memory	Description
Input buffer	Receives packets on one port and services processor requests on the other port, read-only, logically divided into ten fixed-sized packet slots.
Output buffer	Sends packets to the NetFPGA MAC controllers on one port, connected to the processors via its second port.
Data cache	Connected to the processors on one port, 32-bit line-sized data transfers with the DDR2 SDRAM controller (similar to previous work [38]) on the other port.
All three	16KB, single-cycle random access, arbitrated across processors, 32 bits bus.

held in multiple locations. For network processors with no special communication channel between the cores for pipeline operations, such as the 100Gbps-rated 160-threads Cisco QuantumFlow Processor [36], run-to-completion is the natural programming model.

A hybrid of pipelining and run-to-completion approaches is also possible, as shown in Figure 2(c), where each pipeline stage is replicated. While packets can flow across different pipelines, the assumption is that a specialized engine at the input would dispatch a packet to a given pipeline, which would function generally independently. Enforcing this independence to minimize lock contention across pipelines is actually application specific and can lead to important load-imbalance. The number of processors assigned to each pipeline must also be sized according to the number of network interfaces to provide a uniform response time. A variation on the hybrid model consists of using the processors as run-to-completion but delegating atomic operations to specialized processors [37]. That model also removes the need for locks (assuming point-to-point lock-free communication channels) but poses the same problems in terms of ease of programming and load imbalance as the pipeline model.

Nearly all modern multicore processors are logically organized in a grid to which the programmer can map an execution model of his choice. The major *architectural* factors that would push a programmer away from the intuitive run-to-completion model are: (i) a high cache coherence penalty across private data caches, or (ii) a reduced instruction and data cache storage compared to a task decomposition in pipeline(s). Most network processors do not implement cache coherence (except commodity multicore machines) and for our experiments, our data cache is shared and therefore has roughly the same hit rate with or without task pipelining. Because the synchronization around shared data structures in our stateful applications makes it impractical to extract parallelism otherwise (e.g., with a pipeline of balanced execution stages), we adopt the *run-to-completion/pool-of-threads* model, where each thread performs the processing of a packet from beginning-to-end, and where all threads essentially execute the same program code. Consequently our work can be interpreted as an evaluation of a run-to-completion or of an hybrid model where we focus on a single replicated pipeline stage.

5. NETTHREADS ON NETFPGA

In this section we briefly describe the NetThreads [2] multithreaded multiprocessor system that allows us to program the NetFPGA [1] platform in software using shared memory and conventional lock-based synchronization.

NetThreads: Processor Architecture In this work our starting point architecture for comparison is our NetThreads multithreaded multicore architecture that supports only lock-based critical sections, shown in Figure 4. Each processor has a single-issue, in-order, 5-stage pipeline that issues instructions from four hardware threads in round-robin to hide pipeline hazards and cache miss latency [39].

Each processor also has a 16KB private instruction cache and support for thread scheduling [40][4]: when a thread cannot immediately acquire a lock, its slot in the round-robin order can be used by other threads until an unlock operation occurs—this leads to better pipeline utilization by minimizing the execution of instructions that implement spin-waiting. To implement lock-based synchronization, NetThreads provides a synchronization unit containing 16 hardware mutexes; in our ISA, each lock/unlock operation specifies a unique identifier, indicating one of these 16 mutexes. In our current instantiation of NetThreads, 16 mutexes is the maximum number that we can support while meeting the 125MHz target clock speed. However, through simulation we found that supporting an unlimited number of mutexes would improve the performance of our applications by less than 2%, except for Classifier which would improve by 12%.

NetThreads: Memory System As described in Table 3, our multiprocessor architecture is bus-based and designed to match the two-port limitation of block RAMs available on FPGAs. Similarly to other network processors [10, 36], our packet input/output and queuing in the input and output buffers are hardware managed. In addition to the input buffer in Figure 4, the NetFPGA framework can buffer incoming packets for up to 6100 bytes (4 maximally sized packets) but the small overall input storage, while consistent with recent findings that network buffers should be small [41], is very challenging for irregular applications with high computational variance and conservatively caps the maximum steady-state packet rate sustainable via packets dropped at the input of the system. In NetThreads, off-chip memory is accessed via an SDRAM controller that 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 (as opposed to the increased complexity of having private data caches). Because of the port limitation of block RAMs in an FPGA, a shared cache is more efficient than coherent caches for a small number of processors. In its current form, our shared memory architecture will not easily scale to a large number of processors—however, as we demonstrate later in Section 7, our applications are mostly limited by synchronization and critical sections and not contention on the shared buses. In other words, the synchronization inherent in the applications is the primary roadblock to scalability, and our focus in this paper.

Targeting NetFPGA Table 4 describes the details of our implementation of NetThreads on the NetFPGA platform, including compilation, timing, validation, and measurement. Due to stringent timing requirements (there are no free PLLs after merging-in the NetFPGA support components), and despite some available area on the FPGA, we are limited to caches of 16KB each and a maximum of two processors. NetThreads is constrained by, but meets the 125 MHz clock rate of the Ethernet MACs. These limitations are not inherent in our architecture, and would be relaxed in a system with more PLLs and a more modern FPGA.

[4]This thread scheduling [40] is found in the NetThreads-RE released version of NetThreads [2].

Table 4: Implementation Details.

Aspect	Description
Compilation	Modified `gcc` 4.0.2, `Binutils` 2.16, and `Newlib` 1.14.0
Instruction set	32-bit MIPS-I ISA without delay slots [42], with software mul and div
Platform	NetFPGA 2.1 [1] with 4 x 1GigE Media Access Controllers (MACs)
FPGA	Virtex II Pro 50 speed grade 7ns
Synthesis	Xilinx ISE 10.1.03, high effort to meet timing constraints
Off-chip memory	64 Mbytes 200MHz DDR2 SDRAM, Xilinx MIG controller
Processor clock	125MHz, same as Ethernet MACs
Validation	Execution trace generated in RTL simulation and online in debug mode, compared against cycle-accurate simulator
Measuring host	Linux 2.6.18 Dell PowerEdge 2950 with two quad-core 2GHz Xeon processors
Packet source	Modified `Tcpreplay` 3.4.0 sending packet traces from a Broadcom NetXtreme II GigE NIC to an input port of the NetFPGA
Packet sink	NetXtreme GigE NIC connected to another NetFPGA port used for output

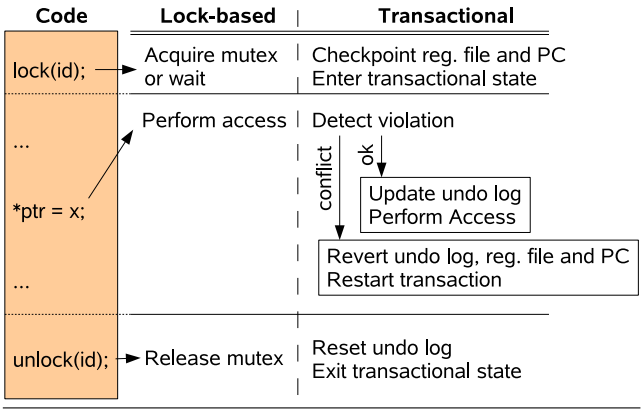


Figure 5: Comparison of the actions taken by the processor if the synchronization is lock-based or transactional.

6. NETTM: NETTHREADS + HTM

To support HTM, as shown in Figure 4, the main additions over our NetThreads implementation are the signature table, the undo-log, and (not shown) support for transactions in the register file and thread scheduler. In this section we describe how these hardware features impact the semantics of synchronization in the code and the execution in NetTM.

6.1 Programming NetTM

In this section, we examine how TM can seamlessly co-exist with lock-based synchronization.

Specifying Transactions TM semantics imply that any transaction will appear to have executed atomically with respect to any other transaction. Like most TM systems, for NetTM a transactional critical section is specified by denoting the start and end of a transaction, using the same instruction API as the lock-based synchronization for NetThreads—i.e., `lock` can mean "start transaction" and `unlock` 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 as shown in Figure 5.

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, in our evaluation, 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.

Composing Locks and Transactions It is desirable for locks and transactions in our system to be composable, meaning that they may be nested within each other. For example, to atomically transfer a record between two linked lists, the programmer might nest existing atomic delete and insert operations within some outer critical section. NetTM supports composition as follows. *Lock within lock* is straightforward and supported. *Lock within transaction* is not supported, since code within a lock-based critical section should never be undone, and we do not support making transactions irrevocable [43]. *Transaction within lock* is supported, although the transaction must be fully nested within the lock/unlock, and will not be executed atomically—meaning that the transaction start/end are essentially ignored, under the assumption that the enclosing lock properly protects any shared data. *Transaction within transaction* is supported, and again the start/end of the inner transaction are ignored.

Improving Performance via Feedback TM allows the programmer to quickly achieve a correct threaded application. Performance can then be improved by reducing transaction aborts, using feedback that pin-points specific memory accesses and data structures that caused the conflicts. While this feedback could potentially be gathered directly in hardware, for now we use our full-system simulator of NetTM to do so. For example, we identified memory management functions (`malloc()` and `free()`) as a frequent source of aborts, and instead employed a light-weight per-thread memory allocator that is not contended on synchronization. The baseline code of our benchmarks is thus optimized by hand with cycle-accurate simulation feedback and attempts to make synchronization as fined-grained as possible to best represent the performance of the locks-only implementations. Data structures were privatized, except for cases where the data should be global (e.g. global hash tables). Privatizing those last data structures implies a different organization of the work to minimize lock contention across the threads, an approach that we also evaluate against transactional memory in Section 7.1.

6.2 Implementing NetTM

A key aspect of any HTM architecture is the required ability to segregate transactional modifications from regular memory. For this purpose, NetTM implements *eager* version management [5]—i.e. writes modify main memory directly and are not buffered. To support rollback for aborts, a backup copy of each modified memory location must be saved in an *undo-log*. Eager version management simplifies NetTM, since the alternative of *lazy* version management would require mechanisms to manage multiple versions of transactionally-modified memory locations. The choice of eager version management drives most of the design of the NetTM architecture, as summarized by the following four key mechanisms.

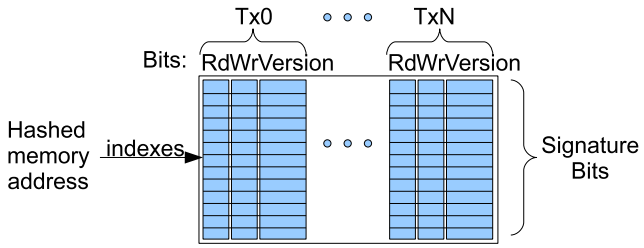


Figure 6: Signature table used to mark speculatively read and/or written for `N` hardware contexts. Version bits are used to distinguish between consecutive transactions on the same hardware context: a mismatch between this version and the context's transaction number indicates that the entry for that context is invalid.

1) Online Conflict Detection Conflicts must be detected before a write is performed, to avoid adding transactional data to the undo-log. To detect conflicts without introducing undue stalls in the system, we must be able to do so in a single cycle. This requirement led us to implement conflict detection via *signatures*, which are bit-vectors that track the memory locations accessed by a transaction via hash indexing [27], with each transaction owning two signatures to track its read and write sets.

2) Application-Specific Signatures To alleviate the tension between the limited on-chip storage of FPGAs and the large number of bits required to prevent commonly-accessed memory locations from mapping to the same signature bit, we construct trie-based application-specific hash functions [27] that map the most contentious profiled memory locations to different signature bits. Performance was previously shown to be highly dependent on the size of the signature [27], so we tune signatures to match the maximum size that fits our FPGA timing constraints for each benchmark individually. Figure 6 shows how the signatures are stored in block RAMs and how the relevant read and write bits of all transactions can be accessed in a single cycle. 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 block RAM to do so. Instead we add to the signature table 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 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

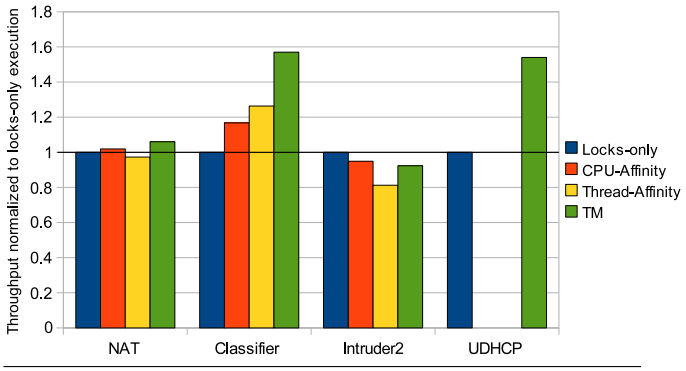


Figure 7: Throughput improvement relative to locks-only (NetThreads) for flow-affinity scheduling and TM (NetTM). In its current form, `UDHCP` is unable to exploit flow-affinity scheduling.

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.

3) Decoupling from the Data Cache Because signatures track transactional accesses, optimistic modifications are allowed to spill from the data cache to next level of memory (in this case off-chip), which is essential to support our benchmarks that exhibit large write sets (Table 2). In the event of a conflict, all optimistic modifications are un-done by flushing appropriately the undo-log (for both on-chip and off-chip locations).

4) Fast Commits Commits in NetTM have a zero-cycle overhead: main memory is eagerly modified, hence to commit we need only discard the corresponding undo-log by simply resetting its write pointer. For our applications, transactions commit in the common case: when measured at maximum saturation rate in our HTM system simulator, 67%, 78%, 97% and 79% of the transactions commit without aborting for `NAT`, `Classifier`, `Intruder2`, and `UDHCP` respectively.

6.3 Managing Transaction Conflicts

When a conflict between two transactions is detected, there are many options for how long to stall or when to restart after aborting [44]. Stalling approaches require frequent re-validation of the read and write sets of the stalled transaction, and can lead to live-lock (if a stalled transaction conflicts with a repeatedly retrying transaction), hence for simplicity we unconditionally abort and restart any transaction that causes a conflict. To prevent restarts from impeding unaborted transactions, we delay restarts until a commit occurs in the system.

7. RESULTS

In this section we evaluate the performance of NetTM relative to NetThreads—in other words we measure the value-added of TM support over support for only conventional lock-based synchronization, for a multithreaded multicore. Figure 7 shows the resulting throughput of the different synchronization alternatives, normalized to that of the conventional-locks-only approach (i.e., NetThreads). We report the maximum sustainable packet rate for a given application 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.

The remainder of this section analyzes these results in detail. We begin by evaluating the important alternative of flow-affinity scheduling for NetThreads.

7.1 Flow-Affinity Scheduling for NetThreads

A simple way to avoid lock contention is to schedule packets from the same flow (i.e., that are likely to contend) to the same hardware context (thread context or CPU). Such a scheduling strategy could potentially lower or eliminate the possibility of lock contention, although by forcing critical sections to be executed serially and threads to wait on each other. We implement flow-affinity scheduling by modifying the source code of our applications such that, after receiving a packet, a thread can either process the packet directly or enqueue (in software with a lock) the packet for processing by other threads. The global shared data structures must be fractionable, replicated or partitioned (in this case) such that this packet scheduling results in fewer data dependences. In Figure 7 we evaluate two forms of flow-affinity, where packets are either mapped to a specific one of the two CPUs (`CPU-Affinity`), or to a specific one of the eight thread contexts available across both CPUs (`Thread-Affinity`).

Flow-affinity is determined for `NAT` and `Classifier` by hashing the IP header fields, and for `Intruder2` by extracting the flow identifier from the packet payload. We cannot evaluate flow-affinity for `UDHCP` because we did not find a clear identifier for flows that would result in parallel packet processing, since UDHCP has many inter-related critical sections (as shown in Figure 1). To implement a reduced lock contention for the flow-affinity approaches we (i) replicated shared data structures when necessary, in particular hash-tables in `NAT` and `Classifier` and (ii) modified the synchronization code such that each CPU operates on a separate subset of the mutexes for `CPU-Affinity`, and uses no mutexes for `Thread-Affinity`.

Figure 7 shows that flow-affinity scheduling only improves `Classifier`. `NAT` shows a slight improvement for CPU-based affinity scheduling and otherwise `NAT` and `Intruder2` suffer slowdowns due to load-imbalance: the downside of flow-affinity scheduling is that it reduces flexibility in mapping packets to threads, and hence can result in load-imbalance. The slowdown due to load-imbalance is less pronounced for `CPU-Affinity` because the packet workload is still shared among the threads of each CPU.

7.2 TM via NetTM

Figure 7 shows that NetTM can achieve throughput improvements of 6%, 57% and 54% for `NAT`, `Classifier` and `UDHCP` relative to NetThreads, by reducing false synchronization and exploiting the optimistic parallelism available across critical sections. TM is particularly useful in enforcing synchronization only when necessary around the global hash table in `NAT` and `Classifier`. `Classifier` has the most speedup because its regular expression matching depends on the data structures stored in the hash table, so parallelizing the hash table operations allows for coarse-grained parallelism. `UDHCP` also experiences a significant

speedup because the operations on the global linked list of IP lease records are coarse grained and can often be at least partially overlapped. Figure 7 also shows that overall TM outperforms the best performing flow-affinity approach by 4% for `NAT` and 31% for `Classifier`, while requiring no special code modification nor program analysis to determine how to privatize global data structures and to dispatch packets to those partitions as in the flow affinity approach explained in Section 7.1.

Despite having a high average commit rate, `Intruder2` has a throughput reduction of 8%: optimistic parallelism is difficult to extract in this case because `Intruder2` has short transactions (Table 2) and an otherwise high CPU utilization such that any transaction abort directly hinders performance. The cost of failed speculation includes the instructions speculatively executed but aborted, the atomic flushing of the undo-log and the reduced parallelism inside a multithreaded core experienced while aborted threads are de-scheduled for a transaction restart. The case of `Intruder2` therefore demonstrates that TM isn't necessarily the best option for every region of code or application. One advantage of NetTM is that the programmer is free to revert to using locks since NetTM integrates support for both transactions and locks.

While there are many possible further code transformations possible to improve TM performance (e.g. minimizing undo-log size or privatizing variables), our experience with our full-system simulator shows that the main potential source of further improvement resides in addressing peaks of serialization that trigger packet drops that thus limit the maximum system throughput. From an architectural standpoint, we believe that the most fruitful avenues for speedups are in addressing those bursts of reduced parallelism using application-specific (i) contention management to throttle a thread to avoid repeated conflicts, and (ii) more expressive signatures that only report conflicts on true data dependences.

8. RELATED WORK

There is a large body of work studying the design space of programming models for network devices. Mudigonda et al. [45] show that the minimum set of hardware features for a network processor to be simple to program and efficient are data-caching to exploit data locality, and multithreading to exploit parallelism and to hide long cache-miss latencies. Our NetThreads system implements both features, but the programmer is left to properly synchronize shared data access between the threads with locks. The need and the effects of inserting synchronization has long been understood for packet processing on multicores [46]. The Aspen project [47] presents a survey of many projects that focus on expressing parallelism by decomposing an application into a pipeline of modules, often linked together with new programming language idioms. Proposing similar automated pipeline decompositions to extract parallelism, compiler efforts [29, 31] either avoid replicating stateful pipeline stages or use locks to ensure correctness. Alternatively, it is possible to mitigate locks by distributing packets to different pipeline stages based on affinity approaches [37, 48] like the one that we implement in this paper. The MultiLayer project [37] reports performance gains with such affinity scheduling because of increased instruction and data locality. However, modern network processors have considerably more memory available than in that study: e.g. the NFP-32xx [49] processors have 20 times more local memory and 160 times more instruction storage. This makes instruction and data locality arguably less important than efficiently managing the parallelism across many cores. While speculative execution was previously proposed for packet processing [6, 7], to our knowledge this paper is the first evaluation with a real system and stateful and control-flow intensive applications. While we do not quantitatively measure the increased ease of programmability of transactional memory for packet processing, we rely on prior efforts to do so in other application domains, e.g. Rossbach et al. [9] demonstrated that over 70% of a pool of 237 student programmers made errors with fine-grained locking on a multiplayer game programming assignment, while less than 10% made errors with transactions.

9. CONCLUSIONS

In this paper we have shown that many complex packet processing applications written in a high-level language, especially those that are stateful, are unsuitable to pipelining. Flow-affinity scheduling can mitigate lock contention for certain applications, assuming the code has such affinity and the programmer is able and willing to replicate and provide scheduling for global data structures as necessary. However, we have demonstrated that transactional memory (TM) provides the best overall performance in most cases, by exploiting the parallelism available in the processing of packets from independent flows, while allowing the most flexible packet scheduling and hence load balancing. Our NetTM implementation 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. For multithreaded applications with shared data and synchronization, we demonstrated that NetTM can improve throughput by 6%, 55%, and 57% over our NetThreads locks-only system, although TM is inappropriate for one application due to short transactions that frequently conflict.

10. REFERENCES

[1] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, "NetFPGA - an open platform for gigabit-rate network switching and routing," in *MSE*, 2007.

[2] M. Labrecque, J. G. Steffan, G. Salmon, M. Ghobadi, and Y. Ganjali, "NetThreads: Programming NetFPGA with threaded software," in *NetFPGA Developers Workshop*, 2009. [Online]. Available: http://www.netfpga.org/foswiki/bin/view/NetFPGA/OneGig/NetThreads

[3] J. Wang, H. Cheng, B. Hua, and X. Tang, "Practice of parallelizing network applications on multi-core architectures," in *ICS*, 2009.

[4] S. Passas, K. Magoutis, and A. Bilas, "Towards 100 gbit/s ethernet: multicore-based parallel communication protocol design," in *ICS*, 2009.

[5] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling hardware transactional memory from caches," in *HPCA*, 2007.

[6] C. Kachris and C. Kulkarni, "Configurable transactional memory," in *FCCM*, 2007.

[7] S. Melvin and Y. Patt, "Handling of packet dependencies: a critical issue for highly parallel network processors," in *CASES*, 2002.

[8] J. Verdú, J. Garcia, M. Nemirovsky, and M. Valero, "Analysis of traffic traces for stateful applications," in *NP-3*, 2004.

[9] C. J. Rossbach, O. S. Hofmann, and E. Witchel, "Is transactional programming actually easier?" in *PPoPP*, 2010.

[10] *IXP2850 Network Processor: Hardware Reference Manual*, Intel Corporation, July 2004.

[11] A. Feldmann, "Internet clean-slate design: what and why?" *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 3, pp. 59–64, 2007.

[12] Intel Corporation, "Packet processing with Intel multi-core processors," 2008.

[13] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The Click modular router," *SIGOPS Oper. Syst. Rev.*, vol. 33, no. 5, pp. 217–231, 1999.

[14] M. Roesch, "Snort - lightweight intrusion detection for networks," in *LISA*, 1999.

[15] "Quagga routing software suite." [Online]. Available: http://www.quagga.net

[16] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov, "Designing extensible IP router software," in *NSDI*, 2005.

[17] Communications Workers of America, "Report on internet speeds in all 50 states," August 2009.

[18] *Cisco carrier routing system*, Cisco Systems, 2004.

[19] T. Wolf and M. Franklin, "CommBench - a telecommunications benchmark for network processors," in *ISPASS*, 2000.

[20] B. K. Lee and L. K. John, "NpBench: A benchmark suite for control plane and data plane applications for network processors," in *ICCD*, 2003.

[21] D. W. Wall, "Limits of instruction-level parallelism," in *ASPLOS*, 1991.

[22] Y. Wang, G. Lu, , and X. Li, "A study of Internet packet reordering," in *ICOIN*, 2004.

[23] "PCRE - Perl compatible regular expressions." [Online]. Available: http://www.pcre.org

[24] "Application layer packet classifier for Linux." [Online]. Available: http://l7-filter.sourceforge.net

[25] Cooperative Association for Internet Data Analysis, "A day in the life of the Internet," WIDE-TRANSIT link, Jan. 2007.

[26] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IISWC*, 2008.

[27] M. Labrecque, M. Jeffrey, and J. G. Steffan, "Application-specific signatures for transactional memory in soft processors," in *ARC*, 2010.

[28] B. Bahlmann, "DHCP network traffic analysis," *Birds-Eye.Net*, June 2005.

[29] J. Dai, B. Huang, L. Li, and L. Harrison, "Automatically partitioning packet processing applications for pipelined architectures," *SIGPLAN Not.*, 2005.

[30] N. Weng and T. Wolf, "Pipelining vs. multiprocessors – choosing the right network processor system topology," in *ANCHOR*, 2004.

[31] W. Thies, "Language and compiler support for stream programs," Ph.D. dissertation, 2009.

[32] G. Memik and W. H. Mangione-Smith, "Evaluating network processors using NetBench," *ACM Trans. Embed. Comput. Syst.*, vol. 5, no. 2, pp. 453–471, 2006.

[33] G. Lidington, "Programming a data flow processor," http://www.xelerated.com, 2003.

[34] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "Pin: a binary instrumentation tool for computer architecture research and education," in *WCAE*, 2004.

[35] R. Ramaswamy, N. Weng, and T. Wolf, "Application analysis and resource mapping for heterogeneous network processor architectures," in *NP-3*, 2004.

[36] Cisco Systems, "The Cisco QuantumFlow processor: Cisco's next generation network processor," 2008.

[37] J. Verdú, M. Nemirovsky, and M. Valero, "Multilayer processing - an execution model for parallel stateful packet processing," in *ANCS*, 2008.

[38] R. Teodorescu and J. Torrellas, "Prototyping architectural support for program rollback using FPGAs," in *FCCM*, 2005.

[39] M. Labrecque, P. Yiannacouras, and J. G. Steffan, "Scaling soft processor systems," in *FCCM*, 2008.

[40] M. Labrecque and J. G. Steffan, "Fast critical sections via thread scheduling for FPGA-based multithreaded processors," in *FPL*, 2009.

[41] N. Beheshti, Y. Ganjali, M. Ghobadi, N. McKeown, and G. Salmon, "Experimental study of router buffer sizing," in *IMC*, 2008.

[42] M. Labrecque, P. Yiannacouras, and J. G. Steffan, "Custom code generation for soft processors," *SIGARCH Comput. Archit. News*, vol. 35, no. 3, pp. 9–19, 2007.

[43] J. E. Gottschlich, D. Connors, D. Winkler, J. G. Siek, and M. Vachharajani, "An intentional library approach to lock-aware transactional memory," University of Colorado at Boulder, Tech. Rep. CU-CS-1048-08, October 2008.

[44] M. Ansari, C. Kotselidis, M. Luján, C. Kirkham, and I. Watson, "On the performance of contention managers for complex transactional memory benchmarks," in *ISPDC*, 2009.

[45] J. Mudigonda, H. M. Vin, and R. Yavatkar, "Overcoming the memory wall in packet processing: hammers or ladders?" in *ANCS*, 2005.

[46] M. Björkman and P. Gunningberg, "Locking effects in multiprocessor implementations of protocols," in *SIGCOMM*, 1993.

[47] G. Upadhyaya, V. S. Pai, and S. P. Midkiff, "Expressing and exploiting concurrency in networked applications with Aspen," in *PPoPP*, 2007.

[48] T. L. Riché, J. Mudigonda, and H. M. Vin, "Experimental evaluation of load balancers in packet processing systems," in *Workshop on Building Block Engine Architectures for Computers and Networks*, 2004.

[49] Netronome Systems, Inc., "NFP-3200 network flow processor," http://www.netronome.com, 2010.