NetThreads: Programming NetFPGA with Threaded Software

Martin Labrecque, J. Gregory Steffan
ECE Dept., University of Toronto
{martinl,steffan}@eecg.toronto.edu

Geoffrey Salmon, Monia Ghobadi,
Yashar Ganjali
CS Dept. University of Toronto
{geoff, monia, yganjali}@cs.toronto.edu

ABSTRACT

As FPGA-based systems including soft processors become increasingly common, we are motivated to better understand the architectural trade-offs and improve the efficiency of these systems. The traditional forwarding and routing are now well understood problems that can be accomplished at line speed by FPGAs but more complex applications are best described in a high-level software executing on a processor. In this paper, we evaluate stateful network applications with a custom multithreaded soft multiprocessor system-on-chip—as an improvement on previous work focusing on single-threaded off-the-shelf soft processors— to demonstrate the features of an efficient yet usable parallel processing system along with potential avenues to improve on its main bottlenecks.

1. INTRODUCTION

The NetFPGA development platform [1] allows networking researchers to create custom hardware designs affordably, and to test new theories, algorithms, and applications at line-speeds much closer to current state-of-the-art. The challenge is that many networking researchers are not necessarily trained in hardware design; and even for those that are, composing packet processing hardware in a *hardware-description language* is time consuming and error prone.

Improvements in logic density and achievable clock frequencies for FPGAs have dramatically increased the applicability of *soft processors*—processors composed of programmable logic on the FPGA. Despite the raw performance drawbacks, a soft processor has several advantages compared to creating custom logic in a hardware-description language: (i) it is easier to program (e.g., using `C`), (ii) it is portable to different FPGAs, (iii) it is flexible (i.e., can be customized), and (iv) it can be used to manage other components/accelerators in the design. However, and most importantly, soft processors are very well suited to packet processing applications that have irregular data access and control flow, and hence unpredictable processing times.

1.1 NetThreads

In this paper we present NetThreads, a NetFPGA-based soft multithreaded multiprocessor architecture. There are several features of our design that ease the implementation of high-performance, irregular packet processing applications. First, our CPU design is multithreaded, allowing a simple and area-efficient datapath to avoid stalls and tolerate memory and synchronization latencies. Second, our memory system is composed of several different memories (instruction, input, output, shared), allowing our design to tolerate the limited number of ports on FPGA block-RAMs while supporting a shared memory. Third, our architecture supports multiple processors, allowing the hardware design to scale up to the limits of parallelism within the application.

We evaluate NetThreads using several parallel packet-processing applications that use shared memory and synchronization, including UDHCP, packet classification and NAT. We measure several *packet processing* workloads on a 4-way multithreaded, 5-pipeline-stage, two-processor instantiation of our architecture implemented on NetFPGA, and also compare with a simulation of the system. We find that synchronization remains a significant performance bottleneck, inspiring future work to address this limitation.

2. MULTITHREADED SOFT PROCESSORS

Prior work [2–6] has demonstrated that supporting *multithreading* can be very effective for soft processors. In particular, by adding hardware support for multiple thread contexts (i.e., by having multiple program counters and logical register files) and issuing an instruction from a different thread every cycle in a round-robin manner, a soft processor can avoid pipeline bubbles without the need for hazard detection logic [2, 4]: a pipeline with N stages that supports $N-1$ threads can be fully utilized without hazard detection logic [4]. A multithreaded soft processor with an abundance of independent threads to execute is also compelling because it can tolerate memory and I/O latency [5], as well as the compute latency of custom hardware accelerators [6]. Such designs are particularly well-suited to FPGA-based processors because (i) hazard detection logic can often be on the critical path and can require significant area [7], and (ii) using the block RAMs provided in an FPGA to implement multiple logical register files is comparatively fast and area-efficient.

The applications that we implement require (i) synchronization between threads, resulting in synchronization latency (while waiting to acquire a lock) and (ii) *critical sections* (while holding a lock). To fulfil these requirements in a way that can scale to more than one processor, we implement locks with memory-mapped test-and-set registers.

2.1 Fast Critical Sections via Thread Scheduling with Static Hazard Detection

While a multithreaded processor provides an excellent opportunity to tolerate the resulting synchronization latency, the simple round-robin thread-issue schemes used previously fall short for two reasons: (i) issuing instructions from a thread that is blocked on synchronization (e.g., spin-loop instructions or a synchronization instruction that repeatedly fails) wastes pipeline resources; and (ii) a thread that currently owns a lock and is hence in a critical section only issues once every $N-1$ cycles (assuming support for $N-1$ thread contexts), exacerbating the synchronization bottleneck for the whole system. Hence we identified a method for *scheduling* threads that is more sophisticated than round-robin but does not significantly increase the complexity nor area of our soft multithreaded processor.

In our approach we *de-schedule* any thread that is awaiting a lock. In particular, any such thread will no longer have instructions issued until any lock is released in the system—at which point the thread may spin once attempting to acquire the lock and if unsuccessful it is blocked again.[1] Otherwise, for simplicity we would like to issue instructions from the unblocked threads in round-robin order.

To implement this method of scheduling we must first overcome two challenges. The first is relatively minor: to eliminate the need to track long latency instructions, our processors *replay* instructions that miss in the cache rather than stalling [5]. With non-round-robin thread scheduling, it is possible to have multiple instructions from the same thread in the pipeline at once—hence to replay an instruction, all of the instructions for that thread following the replayed instruction must be squashed to preserve the program order of instructions execution.

The second challenge is greater: to support any thread schedule other than round-robin means that there is a possibility that two instructions from the same thread might issue with an unsafe distance between them in the pipeline, potentially violating a data or control hazard. We solve this problem by performing *static hazard detection*: we identify hazards between instructions at compile time and encode hazard information into spare bits in the MIPS instruction encoding, decoding it when instructions are fetched into the instruction cache, and storing it by capitalizing on spare bits in the width of FPGA block-RAMs.

3. MULTIPROCESSOR ARCHITECTURE

Our base processor is a single-issue, in-order, 5-stage, 4-way multithreaded processor, shown to be the most area-efficient compared to a 3- and 7-stage pipeline in earlier work [5]. We eliminate the hardware multipliers from our processors, which are not heavily used by our applications. The processor is big-endian which avoids the need to perform network-to-host byte ordering transformations. To take advantage of the space available in the FPGA, we replicate our base processor core and interconnect the replicas to provide them with a coherent common view of the shared data.

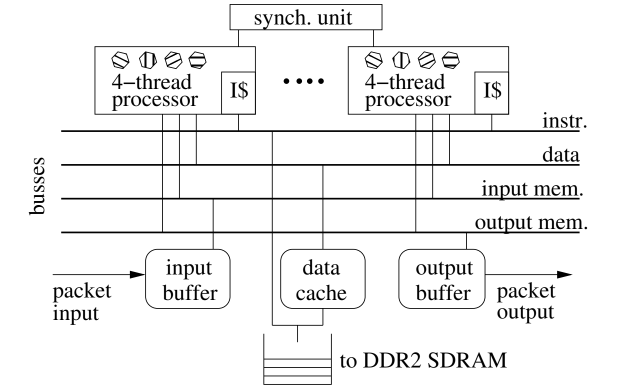

Figure 1: The architecture of a 2-processor soft packet multiprocessor.

As shown in Figure 1, the memory system is composed of a private instruction cache for each processor, and three data memories that are shared by all processors; this design is sensitive to the two-port limitation of block RAMs available on FPGAs. The first memory is an input buffer that receives packets on one port and services processor requests on the other port via a 32-bit bus, arbitrated across processors. The second is an output memory buffer that sends packets to the NetFPGA output-queues on one port, and is connected to the processors via a second 32-bit arbitrated bus on the second port. Both input and output memories are 16KB, allow single-cycle random access and are controlled through memory-mapped registers; the input memory is read-only and is logically divided into ten fixed-sized packet slots. The third is a shared memory managed as a cache, connected to the processors via a third arbitrated 32-bit bus on one port, and to a DDR2 SDRAM controller on the other port. For simplicity, the shared cache performs 32-bit line-sized data transfers with the DDR2 SDRAM controller (similar to previous work [8]), which is clocked at 200MHz. The SDRAM controller services a merged load/store queue of 16 entries in-order; since this queue is shared by all processors it serves as a single point of serialization and memory consistency, hence threads need only block on pending loads but not stores. Finally, each processor has a dedicated connection to a synchronization unit that implements 16 mutexes.

Soft processors are configurable and can be extended with accelerators as required, and those accelerators can be clocked at a separate frequency. To put the performance of the soft processors in perspective, handling a 10^9 bps stream (with an inter-frame gap of 12 bytes) with 2 processors running at 125 MHz implies a maximum of 152 cycles per packet for minimally-sized 64B packets; and 3060 cycles per packet for maximally-sized 1518B packets. Since our multi-

[1] Note that a more sophisticated approach that we leave for future work would only unblock threads that are waiting on the particular lock that was released.

processor architecture is bus-based, in its current form it will not easily scale to a large number of processors. However, as we demonstrate later in Section 6, 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.

4. OUR NETFPGA PROGRAMMING ENVIRONMENT

This section describes our NetFPGA programming environment including how software is compiled, our NetFPGA configuration, and how we do timing, validation, and measurement.

Compilation: Our compiler infrastructure is based on modified versions of `gcc` 4.0.2, `Binutils` 2.16, and `Newlib` 1.14.0 that target variations of the 32-bit MIPS I [9] ISA. We modify MIPS to support 3-operand multiplies (rather than MIPS Hi/Lo registers [4, 7]), and eliminate branch and load delay slots. Integer division and multiplication are both implemented in software. To minimize cache line conflicts in our direct-mapped data cache, we align the top of the stack of each software thread to map to equally-spaced blocks in the data cache.

NetFPGA Configuration: Our processor designs are inserted inside the NetFPGA 2.1 `Verilog` infrastructure [1], between a module arbitrating the input from the four 1GigE Media Access Controllers (MACs) and a CPU DMA port and a module managing output queues in off-chip SRAM. We added to this base framework a memory controller configured through the Xilinx Memory Interface Generator to access the 64 Mbytes of on-board DDR2 SDRAM. The system is synthesized, mapped, placed, and routed under high effort to meet timing constraints by Xilinx ISE 10.1.03 and targets a Virtex II Pro 50 (speed grade 7ns).

Timing: Our processors run at the clock frequency of the Ethernet MACs (125MHz) because there are no free PLLs (a.k.a. Xilinx DCMs) after merging-in the NetFPGA support components. Due to these stringent timing requirements, and despite some available area on the FPGA, (i) the private instruction caches and the shared data write-back cache are both limited to a maximum of 16KB, and (ii) we are also limited to a maximum of two processors. These limitations are not inherent in our architecture, and would be relaxed in a system with more PLLs and a more modern FPGA.

Validation: At runtime in debug mode and in RTL simulation (using `Modelsim` 6.3c [10]) the processors generate an execution trace that has been validated for correctness against the corresponding execution by a simulator built on MINT [11]. We also extended the simulator to model packet I/O and validated it for timing accuracy against the RTL simulation. The simulator is also able to process packets outgoing or incoming from network interfaces, virtual network (tap) devices and packet traces.

API: The memory mapped clock and packet I/O registers are accessible through a simple non-intrusive application programming interface (the API has less than twenty calls), that is easy to build upon. We have developed a number of test applications providing a wealth of routines such as bitmap operations, checksum routines, hashtable and read-write locks.

Measurement: We drive our design, for the packet echo experiment, with a generator that sends copies of the same preallocated packet through `Libnet` 1.4 and otherwise with a modified `Tcpreplay` 3.4.0 that sends packet traces from a Linux 2.6.18 Dell PowerEdge 2950 system, configured with two quad-core 2GHz Xeon processors and a Broadcom NetXtreme II GigE NIC connecting to a port of the NetFPGA used for input and a NetXtreme GigE NIC connecting to another NetFPGA port used for output. To simplify the analysis of throughput measurements, we allow packets to be processed out-of-order so long as the correctness of the application is preserved. We characterize the throughput of the system as being the maximum sustainable input packet rate. We derive this rate by finding, through a bisection search, the smallest fixed packet inter-arrival time where the system does not drop any packet when monitored for five seconds—a duration empirically found long enough to predict the absence of future packet drops at that input rate.

5. APPLICATIONS

In contrast with prior evaluations of packet-processing multiprocessor designs [12–14] we focus on *stateful* applications—i.e., applications in which shared, persistent data structures are modified during the processing of most packets. When the application is composed of parallel threads, accesses to such shared data structures must be synchronized. These dependences make it difficult to pipeline the code into balanced stages of execution to extract parallelism. Alternatively, 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.

To take full advantage of the software programmability of our processors, our focus is on control-flow intensive applications performing deep packet inspection (i.e., deeper than the IP header). Network processing software is normally closely-integrated with operating system networking constructs; because our system does not have an operating system, we instead inline all low-level protocol-handling directly into our programs. To implement time-stamps and time-outs we require the hardware to implement a device that can act as the system clock. We have implemented the following packet processing applications, as detailed in Table 1 (Section 6.1), along with a precise traffic generator tool evaluated in another paper [15].

`UDHCP` is derived from the widely-used open-source DHCP server. The server processes a packet trace modeling the expected DHCP message distribution of a network of 20000 hosts [16]. As in the original code, leases are stored in a linearly traversed array and IP addresses are pinged before being leased, to ensure that they are unused.

`Classifier` performs a regular expression matching on TCP packets, collects statistics on the number of bytes transfered

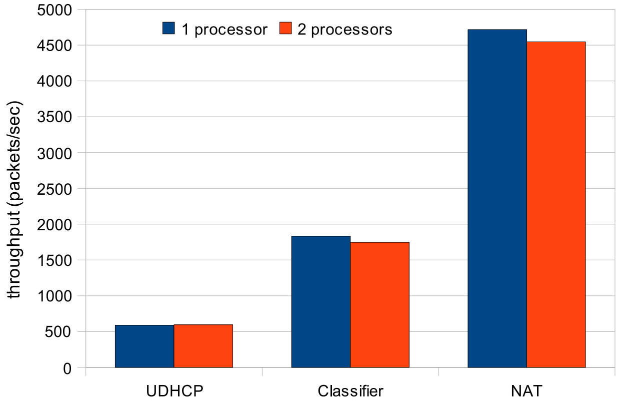

Figure 2: Throughput (in packets per second) measured on the NetFPGA with either 1 or 2 CPUs.

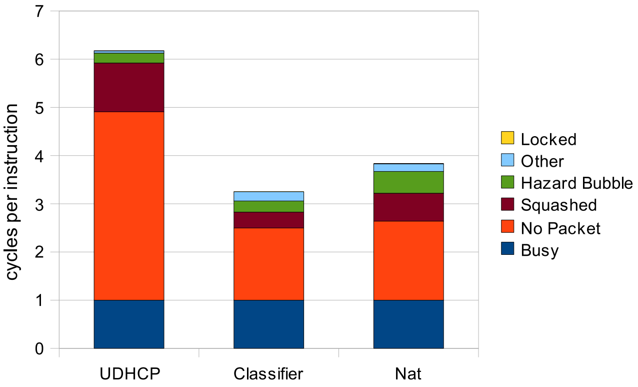

Figure 3: Breakdown of how cycles are spent for each instruction (on average) in simulation.

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 [17] with the HTTP regular expression from the "Linux layer 7 packet classifier" [18] and exercise our system with a publicly available packet trace [19] with HTTP server replies 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. Packets originate from the same packet trace as `Classifier`, and like `Classifier`, flow records are kept in a synchronized hash table.

6. EXPERIMENTAL RESULTS

We begin by evaluating the raw performance that our system is capable of, when performing minimal packet processing for tasks that are completely independent (i.e., unsynchronized). We estimate this upper-bound by implementing a simple packet echo application that retransmits to a different network port each packet received. With minimum-sized packets of 64B, the echo program executes 300±10 dynamic instructions per packet (essentially to copy data from the input buffer to the output buffer as shown in Figure 1), and a single round-robin CPU can echo 124 thousand packets/sec (i.e., 0.07 Gbps). With 1518B packets, the maximum packet size allowable by Ethernet, each echo task requires 1300±10 dynamic instructions per packet. With two CPUs and 64B packets, or either one or two CPUs and 1518B packets, our PC-based packet generator cannot generate packets fast enough to saturate our system (i.e., cannot cause packets to be dropped). This amounts to more than 58 thousand packets/sec (>0.7 Gbps). Hence the scalability of our system will ultimately be limited by the amount of computation per packet/task and the amount of parallelism across tasks, rather than the packet input/output capabilities of our system.

Figure 2 shows the maximum packet throughput of our (real) hardware system with thread scheduling. We find that our applications do not benefit significantly from the addition of a second CPU due to increased lock and bus contention and cache conflicts: the second CPU either slightly improves or degrades performance, motivating us to determine the performance-limiting factors.

6.1 Identifying the Bottlenecks

To reduce the number of designs that we would pursue in real hardware, and to gain greater insight into the bottlenecks of our system, we developed a simulation infrastructure. While verified for timing accuracy, our simulator cannot reproduce the exact order of events that occurs in hardware, hence there is some discrepancy in the reported throughput. For example, `Classifier` has an abundance of control paths and events that are sensitive to ordering such as routines for allocating memory, hash table access, and assignment of mutexes to flow records. We depend on the simulator only for an approximation of the relative performance and behavior of applications on variations of our system.

To obtain a deeper understanding of the bottlenecks of our system, we use our simulator to obtain a breakdown of how cycles are spent for each instruction, as shown in Figure 3. In the breakdown, a given cycle can be spent executing an instruction (`busy`), awaiting a new packet to process (`no packet`), awaiting a lock owned by another thread (`locked`), squashed due to a mispredicted branch or a preceding instruction having a memory miss (`squashed`), awaiting a pipeline hazard (`hazard bubble`), or aborted for another reason (`other`, memory misses or bus contention). Figure 3 shows that our thread scheduling is effective at tolerating almost all cycles spent spinning for locks. The fraction of time spent waiting for packets (`no packet`) is significant and is a result of reducing the worst-case processing latency of a small fraction of packets. The fraction of cycles spent on squashed instructions (`squashed`) is significant with our thread scheduling scheme: if one instruction must replay, we must also squash and replay any instruction from that thread that has already issued. The fraction of cycles spent on bubbles (`hazard bubble`) is significant: this indicates that the CPU is frequently executing instructions from only

Benchmark	Dyn. Instr. ×1000 /packet	Dyn. Sync. Instr. %/packet	Sync. Uniq. Addr. /packet	
			Reads	Writes
UDHCP	34.9±36.4	90±105	5000±6300	150±60
Classifier	12.5±35.0	94±100	150±260	110±200
NAT	6.0±7.1	97±118	420±570	60±60

Table 1: Application statistics (mean±standard-deviation): dynamic instructions per packet, dynamic synchronized instructions per packet (i.e., in a critical section) and number of unique synchronized memory read and write accesses.

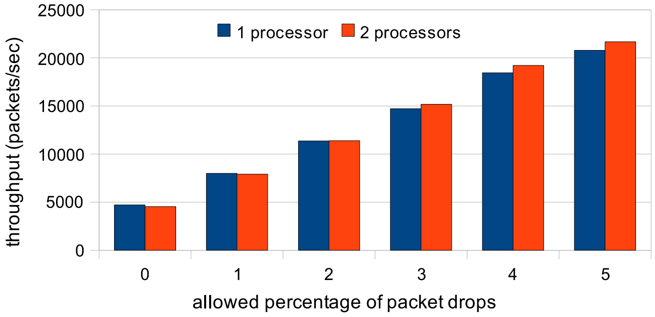

Figure 4: Throughput in packets per second for NAT as we increase the tolerance for dropping packets from 0 to 5%, with either 1 or 2 CPUs.

one thread, with the other threads blocked awaiting locks.

In Table 1, we measure several properties of the computation done per packet in our system. First, we observe that task size (measured in dynamic instructions per second) has an extremely large variance (the standard deviation is larger than the mean itself for all three applications). This high variance is partly due to our applications being best-effort unpipelined C code implementations, rather than finely hand-tuned in assembly code as packet processing applications often are. We also note that the applications spend over 90% of the packet processing time either awaiting synchronization or within critical sections (dynamic synchronized instructions), which limits the amount of parallelism and the overall scalability of any implementation, and in particular explains why our two CPU implementation provides little additional benefit over a single CPU. These results motivate future work to reduce the impact of synchronization, as discussed in Section 8.

Our results so far have focused on measuring throughput when zero packet drops are tolerated (over a five second measurement). However, we would expect performance to improve significantly for measurements when packet drops are tolerated. In Figure 4, we plot throughput for NAT as we increase the tolerance for dropping packets from 0 to 5%, and find that this results in dramatic performance improvements for both fixed round-robin and our more flexible thread scheduling—confirming our hypothesis that task-size variance is undermining performance.

6.2 FPGA resource utilization

Our two-CPU full system hardware implementation consumes 165 block RAMs (out of 232; i.e., 71% of the total capacity). The design occupies 15,671 slices (66% of the total capacity) and more specifically, 23158 4-input LUTs when optimized with high-effort for speed. Considering only a single CPU, the synthesis results give an upper bound frequency of 129MHz.

7. CONCLUSIONS

In most cases, network processing is inherently parallel between packet flows. We presented techniques to improve upon commercial off-the-shelf soft processors and take advantage of the parallelism in stateful parallel applications with shared data and synchronization. We implemented a multithreaded multiprocessor and presented a compilation and simulation framework that makes the system easy to use for an average programmer. We observed that synchronization was a bottleneck in our benchmark applications and plan to pursue work in that direction.

8. FUTURE WORK

In this section, we present two avenues to improve on our architecture implementation to alleviate some of its bottlenecks.

Custom Accelerators Because soft processors do not have the high operating frequency of ASIC processors, it is useful for some applications to summarize a block of instructions into a single custom instruction [20]. The processor interprets that new instruction as a call to a custom logic block (potentially written in a hardware description language or obtained through behavioral synthesis). We envision that this added hardware would be treated like another processor on chip, with access to the shared memory busses and able to synchronize with other processors. Because of the bit-level parallelism of FPGAs, custom instruction can provide significant speedup to some code sections [21, 22].

Transactional Execution When multiple threads/processors collaborate to perform the same application, synchronization must often be inserted to keep shared data coherent. With multiple packets serviced at the same time and multiple packet flows tracked inside a processor, the shared data accessed by all threads is not necessarily the same, and can sometimes be exclusively read by some threads. In those cases, critical sections may be overly conservative by preventively reducing the number of threads allowed in a critical section. Reader and writer locks may not be applicable, or useful, depending on the implementation. To alleviate the problem, a possibility is to allow a potentially unsafe number of threads in a critical section, detect coherence violations if any, abort violated threads and restart them in an earlier checkpointed state. If the number of violations is small, the parallelism, and the throughput, of the application can be greatly increased [23].

NetThreads is available online [24].

9. 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 *Proc. of MSE ’07*, June 3-4 2007.

[2] B. Fort, D. Capalija, Z. G. Vranesic, and S. D. Brown, “A multithreaded soft processor for SoPC area reduction,” in *Proc. of FCCM ’06*, 2006, pp. 131–142.

[3] R. Dimond, O. Mencer, and W. Luk, “Application-specific customisation of multi-threaded soft processors,” *IEE Proceedings—Computers and Digital Techniques*, vol. 153, no. 3, pp. 173– 180, May 2006.

[4] M. Labrecque and J. G. Steffan, “Improving pipelined soft processors with multithreading,” in *Proc. of FPL ’07*, August 2007, pp. 210–215.

[5] M. Labrecque, P. Yiannacouras, and J. G. Steffan, “Scaling soft processor systems,” in *Proc. of FCCM ’08*, April 2008, pp. 195–205.

[6] R. Moussali, N. Ghanem, and M. Saghir, “Microarchitectural enhancements for configurable multi-threaded soft processors,” in *Proc. of FPL ’07*, Aug. 2007, pp. 782–785.

[7] M. Labrecque, P. Yiannacouras, and J. G. Steffan, “Custom code generation for soft processors,” in *Proc. of RAAW ’06*, Florida, US, December 2006.

[8] R. Teodorescu and J. Torrellas, “Prototyping architectural support for program rollback using FPGAs,” in *Proc. of FCCM ’05*, April 2005, pp. 23–32.

[9] S. A. Przybylski, T. R. Gross, J. L. Hennessy, N. P. Jouppi, and C. Rowen, “Organization and VLSI implementation of MIPS,” Stanford University, CA, USA, Tech. Rep., 1984.

[10] Mentor Graphics Corp., “Modelsim SE,” http://www.model.com, Mentor Graphics, 2004.

[11] J. Veenstra and R. Fowler, “MINT: a front end for efficient simulation of shared-memory multiprocessors,” in *Proc. of MASCOTS ’94*, NC, USA, January 1994, pp. 201–207.

[12] T. Wolf and M. Franklin, “CommBench - a telecommunications benchmark for network processors,” in *Proc. of ISPASS*, Austin, TX, April 2000, pp. 154–162.

[13] G. Memik, W. H. Mangione-Smith, and W. Hu, “NetBench: A benchmarking suite for network processors,” in *Proc. of ICCAD ’01*, November 2001.

[14] B. K. Lee and L. K. John, “NpBench: A benchmark suite for control plane and data plane applications for network processors,” in *Proc. of ICCD ’03*, October 2003.

[15] G. Salmon, M. Ghobadi, Y. Ganjali, M. Labrecque, and J. G. Steffan, “NetFPGA-based precise traffic generation,” in *Proc. of NetFPGA Developers Workshop’09*, 2009.

[16] B. Bahlmann, “DHCP network traffic analysis,” *Birds-Eye.Net*, June 2005.

[17] “PCRE - Perl compatible regular expressions,” [Online]. Available: http://www.pcre.org.

[18] “Application layer packet classifier for linux,” [Online]. Available: http://l7-filter.sourceforge.net.

[19] Cooperative Association for Internet Data Analysis, “A day in the life of the internet,” WIDE-TRANSIT link, January 2007.

[20] H.-P. Rosinger, “Connecting customized IP to the MicroBlaze soft processor using the Fast Simplex Link (FSL) channel,” XAPP529, 2004.

[21] R. Lysecky and F. Vahid, “A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning,” in *Proc. of DATE ’05*, 2005, pp. 18–23.

[22] C. Kachris and S. Vassiliadis, “Analysis of a reconfigurable network processor,” in *Proc. of IPDPS*. Los Alamitos, CA, USA: IEEE Computer Society, 2006, p. 173.

[23] C. Kachris and C. Kulkarni, “Configurable transactional memory,” in *Proc. of FCCM ’07*, April 2007, pp. 65–72.

[24] “NetThreads - project homepage,” [Online]. Available: http://netfpga.org/netfpgawiki/index.php/Projects:NetThreads.