FAST CRITICAL SECTIONS VIA THREAD SCHEDULING FOR FPGA-BASED MULTITHREADED PROCESSORS

Martin Labrecque and J. Gregory Steffan

Department of Electrical and Computer Engineering, University of Toronto
email: {martinl,steffan}@eecg.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. Previous work has demonstrated that support for multithreading in soft processors can tolerate pipeline and I/O latencies as well as improve overall system throughput—however earlier work assumes an abundance of completely independent threads to execute. In this work we show that for real workloads, in particular packet processing applications, there is a large fraction of processor cycles wasted while awaiting the synchronization of shared data structures, limiting the benefits of a multithreaded design. We address this challenge by proposing a method of scheduling threads in hardware that allows the multithreaded pipeline to be more fully utilized without significant costs in area or frequency. We evaluate our technique relative to conventional multithreading using both simulation and a real implementation on a *NetFPGA* board, evaluating three deep-packet inspection applications that are threaded, synchronize, and share data structures, and show that overall packet throughput can be increased by 63%, 31%, and 41% for our three applications.

1. INTRODUCTION

Improving logic density and maximum clock rates of FPGAs have led to an increasing number of FPGA-based system-on-chip designs, which in turn increasingly contain one or more *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: it is easier to program (e.g., using `C`), portable to different FPGAs, flexible (i.e., can be customized), and can be used to control or communicate with other components/custom accelerators in the design.

Prior work [1–5] 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 stalls and pipeline bubbles without the need for hazard detection logic [1, 3]: a pipeline with N stages that supports $N-1$ threads can be fully utilized without hazard detection logic [3]. 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 [6], and (ii) using the block RAMs provided in an FPGA to implement multiple logical register files is comparatively fast and area-efficient.

A multithreaded soft processor with an abundance of independent threads to execute is also compelling because it can tolerate memory and I/O latency [4], as well as the compute latency of custom hardware accelerators [5]. In prior work it is generally assumed that such an abundance of completely independent threads is available—modeled as a collection of independent benchmark kernels [1–5]. However, in real systems, threads will likely share memory and communicate, requiring (i) synchronization between threads, resulting in synchronization latency (while waiting to acquire a lock) and (ii) *critical sections* (while holding a lock).

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.

1.1. Fast Critical Sections via Thread Scheduling with Static Hazard Detection

In this work we demonstrate that implementing a thread-scheduling policy that is more sophisticated than the round-robin approach can significantly improve performance with only a small area and frequency cost. The main goal of our proposed scheduler is to de-prioritize any thread that is awaiting a lock such that it cannot issue further instructions until the thread has an opportunity to acquire the lock. While this solution makes intuitive sense, its implementation results in a significant challenge: assuming a pipeline of N stages that supports $N-1$ thread contexts, if one or more threads is blocked awaiting synchronization then there

are no longer enough threads to naturally utilize the pipeline; to do so, multiple instructions from a single thread must be in flight, which once again requires a method of hazard detection across instructions within that same thread.

We consider several methods for re-introducing hazard detection. First, we could simply add hazard detection logic to the pipeline—but this would increase area and reduce clock frequency, and would also lead to stalls and bubbles in the pipeline. Second, we could consult hazard detection logic to find a hazard-free instruction to issue from any ready thread—but this more complex approach requires the addition of an extra pipeline stage, and we demonstrate that it does not perform well. A third solution, which we advocate in this paper, is to perform *static hazard detection* by identifying hazards at compile time and encoding hazard information into the instructions themselves. This approach capitalizes on unused bits in block RAM words on FPGAs[1] to store these *hazard bits*, allowing us to fully benefit from more sophisticated thread scheduling.

Evaluation with a Realistic System: We evaluate our thread scheduler with static hazard detection using realistic parallel workloads that have shared memory, synchronization, and critical sections. In particular, we measure both simulation and real execution of several *packet processing* workloads on a 4-way multithreaded, 5-stage, two-processor system implemented on the NetFPGA development platform [7]. We find that, with simple round-robin thread scheduling, over 18% of all potential issue cycles are wasted while blocked on synchronization. In contrast, our scheduler with static hazard detection spends almost none of the issue cycles blocked on synchronization, and exhibits a 38% improvement in overall CPI and a 45% improvement in total packet throughput relative to the default round-robin scheme.

1.2. Related work

The closest work on thread scheduling for soft processors that we are aware of is by Moussali *et. al.* [5] who use a table of pre-determined worst-case instruction latencies to avoid pipeline stalls. Our technique can handle the same cases while additionally prioritizing critical threads and handling unpredictable latencies. In the ASIC world, thread scheduling is an essential part of multithreading with synchronized threads [8]. The IXP [9] family of network processors use non-preemptive thread scheduling where threads exclusively occupy the pipeline until they voluntarily deschedule themselves when awaiting an event. Other examples of in-order multithreaded processors include the Niagara [10] and the MIPS 34K [11] processors where instructions from each thread wait to be issued in a dedicated pipeline stage. While thread scheduling and hazard detection are well studied in general (operating systems provide thread management primitives [12] and EPIC architectures, such as IA-64 [13], bundle independent instructions to maximize instruction-level parallelism), our goal is to achieve thread scheduling efficiently in the presence of synchronization at the fine grain required to tolerate pipeline hazards.

[1]Extra bits in block RAMs are available across FPGA vendors: block RAMs of almost all granularities are configured in widths that are multiples of nine bits, while processors normally have busses multiples of eight bits wide.

1.3. Contributions

This paper makes the following contributions: (i) we demonstrate that thread scheduling is crucial for multithreaded soft processors executing synchronized workloads; (ii) we propose methods of thread scheduling and hazard detection that significantly improve pipeline utilization without reducing clock frequency; (iii) we evaluate our approach on both simulated and real packet-processing systems and demonstrate improvements in critical section latency, packet latency variability, and overall packet throughput.

2. PACKET PROCESSING

Network packet processing is no longer limited solely to routing, with many applications that require deep packet inspection becoming increasingly common and desired: examples include encryption / decryption, compression / decompression, content-based routing, traffic shaping, intrusion/virus detection, and more. The goal of such a packet processing systems is often to process packets at line rate, scaling up a system composed of processors and accelerators to make full use of available bandwidth to and from a given packet-buffer (i.e., memory channel). FPGAs are increasingly used in these systems [14–16] due to several advantages that they provide: (i) ease of design and fast time-to-market; (ii) the ability to connect to a number of memory channels and network interfaces, possibly of varying technologies; (iii) the ability to fully exploit parallelism and custom accelerators; and (iv) the ability to field-upgrade the hardware design.

To measure packet throughput, we need to define the processing performed on each packet. In contrast with prior work [17–19] 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.

2.1. Benchmark Applications

To take full advantage of the software programmability of our processors, our focus is on control-flow intensive ap-

plications 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).

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 [20]. As in the original code, leases are stored in a linearly traversed array and IP addresses are ping'ed 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 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 [21] with the HTTP regular expression from the "Linux layer 7 packet classifier" [22] and exercise our system with a publicly available packet trace [23] 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.

3. MULTITHREADED SOFT 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 [4]. 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.

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

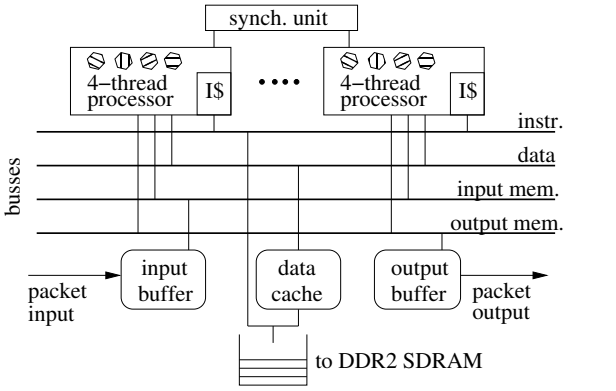


Fig. 1. The architecture of a 2-processor soft packet multiprocessor.

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 [24]), 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.

Since our multiprocessor 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. IMPLEMENTING THREAD SCHEDULING

A multithreaded processor has the advantage of being able to fully utilize the processor pipeline by issuing instructions from different threads in a simple round-robin manner to avoid stalls and hazards. However, for real workloads with shared data and synchronization, one or more threads may often spin awaiting a lock, and issuing instructions from such threads is hence a waste of pipeline resources. Also, a thread which holds a lock (i.e., is in a critical section)

can potentially be the most important, since other threads are likely waiting for that lock; ideally we would allocate a greater share of pipeline resources to such threads. Hence in this section we consider methods for *scheduling* threads that are more sophisticated than round-robin but do not significantly increase the complexity nor area of our soft multithreaded processor.

The most sophisticated possibility would be to give priority to any thread that holds a critical lock, and otherwise to schedule a thread having an instruction that has no hazards with current instructions in the pipeline. However, this method is more complex than it sounds due to the possibility of nested critical sections: since a thread may hold multiple locks simultaneously, and more than one thread may hold different locks, scheduling such threads with priorities is very difficult and could even result in deadlock. A correct implementation of this aggressive scheduling would likely also be slow and expensive.

Instead of giving important threads priority, in our approach we propose to only *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.[2] 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 [4]. 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 with an implementation of *static hazard detection*.

4.1. Static Hazard Detection

With the ability to issue from any thread not waiting for a lock, the thread scheduler must ensure that dependences between the instructions from the same thread are enforced, either from the branch target calculation to the fetch stage, or from the register writeback to the register read. The easiest way to avoid such hazards is to support *forwarding lines*. By supporting forwarding paths between the writeback and register-read stages of our pipeline, we can limit the maximum hazard distance between instructions from the same thread to two stages.

[2] A more sophisticated approach that we leave for future work would only unblock threads waiting on the particular lock that was released.

Program Disassembly	Hazard distance	Min. issue cycle
addi r1,r1,r4	0	0
addi r2,r2,r5	1	1
or r1,r1,r8	0	3
or r2,r2,r9	0	4

Fig. 2. Example insertion of hazard distance values. Arrows indicate register dependences, implying that the corresponding instructions must be issued with at least two pipeline stages between them. A hazard distance of one encoded into the second instruction commands the processor to ensure that the third instruction does not issue until cycle 3, and hence the fourth instruction cannot issue until cycle 4.

Our scheduling technique consists of determining hazards at compile-time and inserting *hazard distances* as part of the instruction stream. Because our instructions are fetched from off-chip DDR2 memory into our instruction cache, it is impractical to have instructions wider than 32 bits. We therefore compress instructions to accommodate the hazard distance bits in the program executable, and decompress them as they are loaded into the instruction cache. We capitalize on the unused capacity of block RAMs, which have a width multiple of 9 bits—to support 32-bit instructions requires four 9-bit block RAMs, hence there are 4 spare bits for this purpose.

To represent instructions in off-chip memory in fewer than 32 bits, we compress them according to the three MIPS instruction types [25]: for the `R`-type, we merge the function bit field into the opcode field and discard the original function bit field; for the `J`-type instructions, we truncate the target bit field to use fewer than 26 bits; and for the `I`-type, we replace the immediate values by their index in a lookup table that we insert into our FPGA design. To size this lookup table, we found that there are usually more than 1024 unique 16-bit immediates to track, but that 2048 entries is sufficient to accommodate the union of the immediates of all our benchmarks. Therefore, the instruction decompression in the processor incurs a cost of some logic and 2 additional block RAMs[3], but not on the critical path of the processor pipeline. After compression, we can easily reclaim 4 bits per instruction: 2 bits are used to encode the maximum hazard distance, and 2 bits are used to identify lock request and release operations. Our compiler automatically sets these bits accordingly when it recognizes memory-mapped accesses for the locks.

An example of code with hazard distances is shown in Figure 2: the compiler must account for the distances in-

[3] For ease of use, the immediate lookup table could be initialized as part of the loaded program executable, instead of currently, the FPGA bit file.

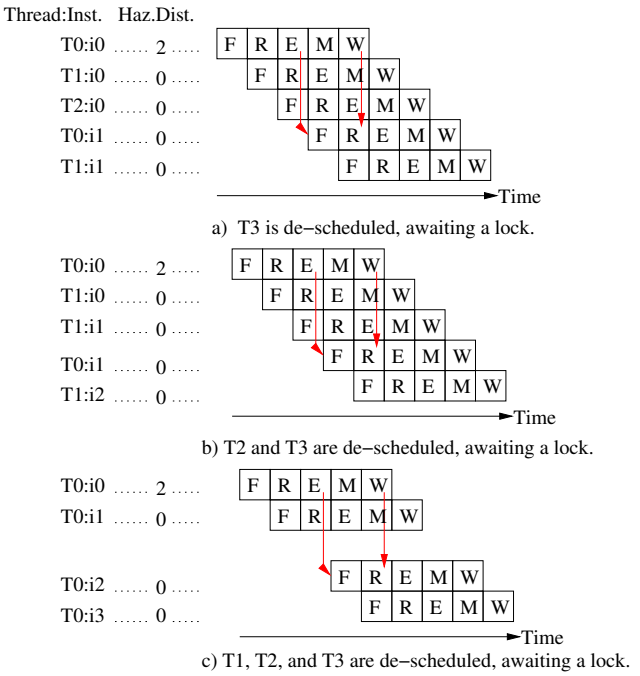


Fig. 3. Examples using hazard distance to schedule threads. The pipeline stages are: `F` for fetch, `R` for register, `E` for execute, M for memory, and `W` for write-back. The arrows indicate potential branch target or register dependences.

serted between the previous instructions to avoid inserting superfluous hazard distances. It first evaluates hazards with no regard to control flow: if a branch is not-taken, the hazards will be enforced by the hazard distance bits; if the branch is taken, the pipeline will be flushed from instructions on the mispredicted path and hazards will be automatically avoided. If we extended our processors to be able to predict taken branches, we would have to recode the hazard distance bits to account for both execution paths. As a performance optimization, we insert a hazard distance of 2 for unconditional jumps to prevent the processor from fetching on the wrong path, as it takes 2 cycles to compute the branch target (shown in Figure 3). In our measurements, we found best not to insert any hazard distance on conditional branches; an improvement would consist of using profile feedback to selectively insert hazard distances on mostly taken branches. With more timing margin in our FPGA design, we could explore other possible refinements in the thread scheduler to make it aware of load misses and potentially instantiate lock priorities.

At runtime upon instruction fetch, the hazard distances are loaded into counters that inform the scheduler about hazards between instructions in unblocked threads as illustrated in Figure 3. When no hazard-free instruction is available for issue (Figure 3c), the scheduler inserts a pipeline bubble. In our processors with thread scheduling, when two memory instructions follow each other and the first one misses, we found that the memory miss signal does not have enough timing slack to disable the second memory access while meeting our frequency requirements. Our solution is to take advantage of our hazard distance bits to make sure that consecutive memory instructions from the same thread are spaced apart by at least one instruction.

Note that in off-the-shelf soft processors, the generic hazard detection circuitry identifies hazards at runtime (potentially with a negative impact on the processor frequency) and inserts bubbles in the pipeline as necessary. To avoid such bubbles in multithreaded processors, ASIC implementations [10,11] normally add an additional pipeline stage for the thread scheduler to select hazard-free instructions. We evaluate the performance of this approach along with ours in the next section.

5. EVALUATION INFRASTRUCTURE

This section describes our evaluation infrastructure, including compilation, our evaluation platform, 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 [26] ISA. We modify MIPS to support 3-operand multiplies (rather than MIPS Hi/Lo registers [3,6]), 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.

Platform: Our processor designs are inserted inside the NetFPGA 2.1 `Verilog` infrastructure [7] that manages four 1GigE Media Access Controllers (MACs) and 4.5 Mbytes of on-board SRAM to buffer packets before they are sent on the network. 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 (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.

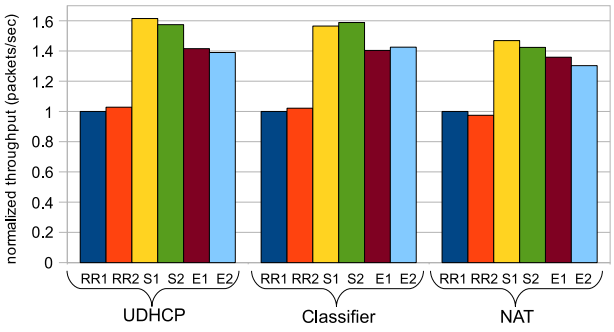
Validation: At runtime in debug mode and in RTL simulation (using `Modelsim` 6.3c the processors generate an execution trace that has been validated for correctness against the corresponding execution by a simulator built on MINT [27]. We validated the simulator for timing accuracy against the RTL simulation.

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. We characterize the throughput of the system as being the maximum sustainable input packet rate obtained 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. While we could enforce ordering in software, we allow packets to be processed out-of-order because our application semantics allow it.

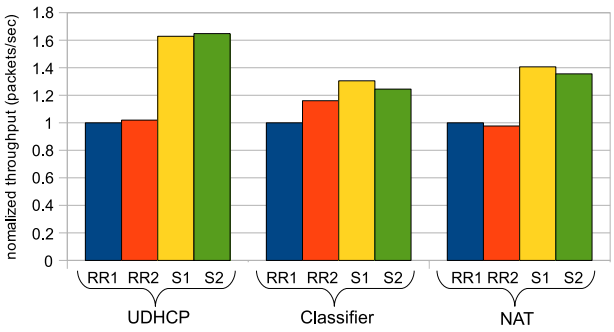
6. EXPERIMENTAL RESULTS

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 packet echo application that simply copies packets from an input port to an output port. With minimum-sized packets of 64B, the echo program executes 300±10 dynamic instructions per packet, 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.

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



(a) Simulation results.



(b) Hardware results.

Fig. 4. Throughput (in packets per second) normalized to that of a single round-robin CPU. Each design has either round-robin scheduling (`RR`), our proposed scheduling (`S`), or scheduling via an extra pipeline stage (`E`), and has either `1` or `2` CPUs.

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.

Figure 4(a) shows the maximum packet throughput of our simulated system, normalized to that of a single round-robin CPU; these results estimate speedups for our scheduling on a single CPU (`S1`) of 61%, 57% and 47% for `UDHCP`, `Classifier` and `NAT` respectively.[4] We also used the simulator to estimate the performance of an extra pipeline stage for scheduling (`E1` and `E2`, as described in Section 4.1), but find that our technique dominates in every case: the cost of extra squashed instructions for memory misses and mispredicted branches for the longer pipeline overwhelms any scheduling benefit—hence we do not pursue that design in hardware.

Figure 4(b) shows the maximum packet throughput of our (real) hardware system, normalized to that of a single round-robin CPU. We see that with a single CPU our scheduling technique (`S1`) significantly out-performs round-

[4]We will report benchmark statistics in this order from this point on.

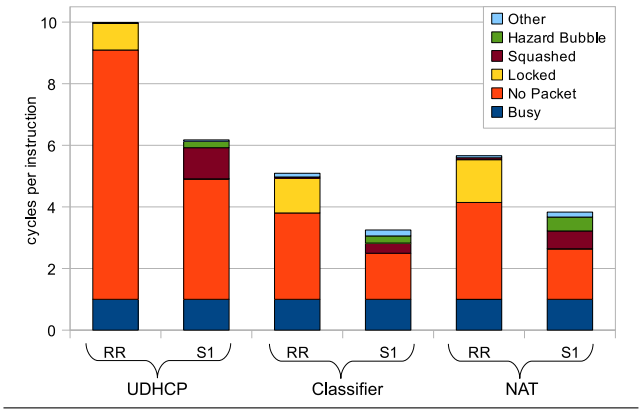


Fig. 5. Average cycles breakdown for each instruction at the respective maximum packet rates from Figure 4(a).

robin scheduling (`RR1`) by 63%, 31%, and 41% across the three applications. However, we also find that our applications do not benefit significantly from the addition of a second CPU due to increased lock and bus contention, and reduced cache locality: for `Classifier` two round-robin CPUs (`RR2`) is 16% better, but otherwise the second CPU either very slightly improves or degrades performance, regardless of the scheduling used. We also observe that our simulator (Figure 4(a)) indeed captures the correct relative behaviour of the applications and our system.

Comparing two-CPU full system hardware designs, the round-robin implementation consumes 163 block RAMs (out of 232, i.e., 70% of the total capacity) compared to 165 blocks (71%) with scheduling: two additional blocks are used to hold the lookup table for instruction decoding (as explained in Section 4.1). The designs occupy respectively 15,891 and 15,963 slices (both 67% of the total capacity) when optimized with high-effort for speed. Considering only a single CPU, the synthesis results give an upper bound frequency of 136MHz for the round-robin CPU and 129MHz for scheduling. Hence the overall overhead costs of our proposed scheduling technique are low, with a measured area increase of 0.5% and an estimated frequency decrease of 5%.

6.1. Identifying the Bottlenecks

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 5. 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 5 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 reduced by 52%, 47%, and 48%, a result of reducing the worst-case processing latency of packets: our simulator reports that the task latency standard deviation decreases by 34%, 33%, and 32%. The fraction of cycles spent on squashed instructions (`squashed`) becomes significant with our proposed scheduling: recall that if one instruction must replay that we must also squash and replay any instruction from that thread that has already issued. The fraction of cycles spent on bubbles (`hazard bubble`) also becomes significant: this indicates that the CPU is frequently executing instructions from only one thread, with the other threads blocked awaiting locks.

Benchmark	Dyn. Instr. ×1000/pkt	Dyn. Sync. Instr. %/pkt	Sync. Uniq. Addr. /pkt 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) per packet: dynamic instructions, dynamic synchronized instructions (i.e., in a critical section) and number of unique synchronized memory read and write accesses.

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 note also 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.

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 6, 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.

7. CONCLUSIONS

In this paper, we have shown that previously studied multithreaded soft processors with fixed round-robin thread in-

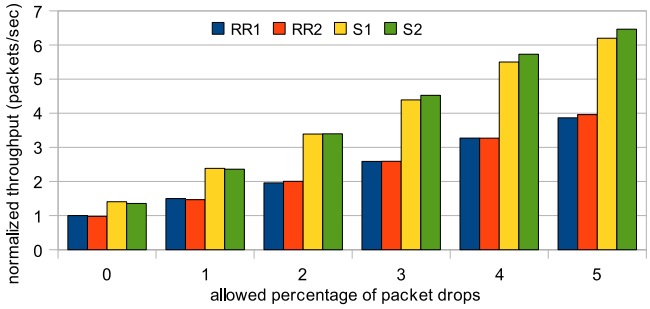


Fig. 6. Throughput in packets per second for `NAT` as we increase the tolerance for dropping packets from 0 to 5%. Each design has either round-robin scheduling (`RR`) or our proposed scheduling (`S`) and has either `1` or `2` CPUs.

terleaving can spend a significant amount of cycles spinning for locks when all the threads contribute to the same application and have synchronization around data dependences. We presented a technique to implement a more advanced thread scheduling that has minimal area and frequency overhead, because it capitalizes on features of the FPGA fabric. Our scheme builds on static hazard detection and performs better than the scheme used in ASIC processors with hazard detection logic because it avoids the need for an additional pipeline stage. Our improved handling of critical sections with thread scheduling improves the instruction throughput which results in reduced processing latency average and variability. Using a real FPGA-based network interface, we measured packet throughput improvements of 63%, 31% and 41% for our three applications.

8. REFERENCES

[1] 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.

[2] 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.

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

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

[5] 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.

[6] M. Labrecque, P. Yiannacouras, and J. G. Steffan, "Custom code generation for soft processors," in *Proc. of RAAW '06*, December 2006.

[7] 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.

[8] T. Ungerer, B. Robič, and J. Šilc, "A survey of processors with explicit multithreading," *ACM Computing Surveys (CSUR)*, vol. 35, no. 1, March 2003.

[9] *IXP2800 Network Processor: Hardware Reference Manual*, Intel Corporation, July 2005.

[10] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded Sparc processor," *Micro, IEEE*, vol. 25, no. 2, pp. 21–29, March-April 2005.

[11] K. D. Kissell, "MIPS MT: A multithreaded RISC architecture for embedded real-time processing," in *Proc. of HiPEAC '08*, Sweden, January 2008, pp. 9–21.

[12] F. Mueller, "A library implementation of POSIX threads under UNIX," in *Proc. of USENIX*, 1993, pp. 29–41.

[13] *Intel®Itanium®Architecture Software Developer's Manual*, Intel Corporation, January 2006.

[14] Y. J. Kaushik Ravindran, Nadathur Satish and K. Keutzer, "An FPGA-based soft multiprocessor system for IPv4 packet forwarding," in *Proc. of FPL '05*, August 2005, pp. 487–492.

[15] B. Moyer, "Packet subsystem on a chip," *Xcell Journal*, pp. 10–13, 2006.

[16] C. Kachris and S. Vassiliadis, "Analysis of a reconfigurable network processor," *Proc. of IPDPS*, vol. 0, p. 173, 2006.

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

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

[19] 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.

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

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

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

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

[24] R. Teodorescu and J. Torrellas, "Prototyping architectural support for program rollback using FPGAs," *Proc. of FCCM '05*, pp. 23–32, April 2005.

[25] T. Bonny and J. Henkel, "Instruction re-encoding facilitating dense embedded code," in *Proc. of DATE*, 2008, pp. 770–775.

[26] 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.

[27] 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.