Scaling Soft Processor Systems

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

Abstract

As FPGA-based systems including soft-processors become increasingly common we are motivated to better understand the best way to scale the performance of such systems. In this paper we explore the organization of processors and caches connected to a single off-chip memory channel, for workloads composed of many independent threads. In particular we design and evaluate real FPGA-based processor, multithreaded processor, and multiprocessor systems on EEMBC benchmarks—investigating different approaches to scaling caches, processors, and thread contexts to maximize throughput while minimizing area. Our main finding is that while a single multithreaded processor offers improved performance over a single-threaded processor, multiprocessors composed of single-threaded processors scale better than those composed of multithreaded processors.

1. Introduction

As embedded systems designers increasingly use FPGAs to implement single-chip designs, they are often motivated to use *soft processors*—processors built using an FPGA's programmable logic. Soft processors provide a familiar programming environment allowing non-hardware experts to target FPGAs. Soft processor environments allow FPGA designs to be more easily modified, and typically provide methods for bottleneck computations to be accelerated as necessary through additional custom hardware [1, 2]. One of the strengths of an FPGA-based design is the ability to connect to a number of memory channels, possibly of varying memory technologies; a typical performance goal for the construction of such a system is to fully-utilize a given memory channel. For example, in the field of packet processing the goal 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).

The goal of this paper is to explore architectural options for scaling the performance of soft systems, focussing on the organization of processors and caches connected to a single off-chip memory channel. For now we consider systems similar to packet processing where there are many independent tasks/threads to execute, and maximizing system throughput is the over-arching performance goal. In particular, we use a real FPGA-based soft system connected to DDR SDRAM to evaluate the performance scaling of soft uniprocessors, multithreaded processors, and multiprocessors. We focus solely on regular processors for now, leaving a study of the role of application-specific hardware accelerators for future work.

Scaling Soft Uniprocessors It is relatively straightforward to connect a soft uniprocessor to off-chip DDR memory through data and instruction caches. We analyze the resulting memory latency, the usage of FPGA resources, and the impact of scaling up the data cache size for such a system. Our analysis shows that off-chip memory latency is not as much of a challenge for FPGA-based processors as it is for ASIC processors, limiting the impact of large caches. As expected we find that increasing the data cache size shows diminishing returns, but surprisingly that a relatively small direct-mapped cache can capture most of the benefit of an infinite-sized cache. Hence we are motivated to pursue other methods of scaling the performance of soft systems.

Scaling Soft Multithreaded Processors In previous work we demonstrated that, for a system with only on-chip memory, a soft multithreaded processor can eliminate nearly all of the stall-cycles observed by a comparable single-threaded processor by executing an independent instruction in every stage of the processor pipeline [3]. In this paper we extend our multithreaded processors to interface with off-chip DDR memory through caches, which in contrast with uniprocessors presents some interesting challenges and design options. In particular we present an approach called instruction *replay* to handle cache misses without stalling other threads, and a mechanism for handling the resulting potential live-locks. We also evaluate several cache organization alternatives for soft multithreaded processors, namely shared, private, and partitioned caches, as well as support for different numbers of hardware contexts. We finally investigate issues related to sharing and conflicts between threads for soft multithreaded processors. In contrast with previous studies of systems with on-chip memory [3,4], with off-chip memory we find that single-threaded processors are generally more area-efficient than multithreaded processors.

This research is supported by grants from Altera Corporation and NSERC. Martin Labrecque and Peter Yiannacouras are partially supported by FCAR and NSERC scholarships respectively.

Scaling Soft Multiprocessors We also evaluate multiprocessors composed of uniprocessors or multithreaded processors. We find that, for a system of given area, multiprocessors composed of multithreaded processors provide a much larger number of thread contexts, but that uniprocessor-based multiprocessors provide the best overall throughput.

1.1. Related Work

There is a large body of prior work on memory systems, showing the benefits and trade-offs of different cache organizations for conventional systems; for FPGA-based systems, caches built in the FPGA fabric [5] are routinely utilized to improve the performance of systems with off-chip memory. The commercial off-the-shelf soft processors Nios-II [6] and Microblaze [7] both support optional direct-mapped instruction and data caches with configurable cache line sizes. Both processors allocate a cache line upon write misses (allocate-on-write) but the Microblaze uses a write-through policy while Nios-II uses write-back. Both vendors have extended their instruction set to accomplish tasks such as cache flushing, invalidating and bypassing. Our implementation is comparable to those designs (other than we do not allocate on writes) and we did not require ISA extensions: this renders our conclusions widely applicable.

While there exist several academic multiprocessor-on-FPGA implementations [8–10], there has been little work studying the impact of processor and cache architecture on the scalability of performance. Fort et al. [4] compared their soft multithreaded processor design to having multiple uniprocessors, using the Mibench benchmarks [11] and a system with on-chip instruction memory and off-chip shared data storage (with no data cache). They conclude that a single multithreaded soft processor can perform similarly to multiple uniprocessors in that environment. We further demonstrated that a soft multithreaded processor can be more area efficient than a single soft uniprocessor with on-chip memory [3].

The addition of off-chip memory and caches introduces variable-latency stalls to the processor pipeline. Handling such stalls in a multithreaded processor without stalling all threads is a challenge. Fort *et. al.* [4] use a FIFO queue of loads and stores, while Moussali *et. al.* [12] use an instruction scheduler to issue ready instructions from a pool of threads. In contrast with either of these approaches, our instruction *replay* approach requires little additional hardware support.

Commercial FPGA systems often have more than one off-chip memory channel [13–16], and in fact supporting multiple memory channels is one of the benefits of an FPGA implementation. Kulmala *et al.* [17] implement a distinct memory channel for instruction memory, and show that with small instruction caches, the scalability of the system throughput depends mainly on the data memory. For simplicity and because we expect workloads to often be parallel across memory channels, in our work so far we focus on maximizing performance given a single off-chip memory channel.

1.2. Contributions

This paper makes the following contributions: we describe and evaluate soft uniprocessors, multithreaded processors, and multiprocessors in a real system with off-chip DDR SDRAM, and evaluate them with EEMBC industrial benchmarks; we demonstrate the importance of evaluating with a real system and with off-chip memory, as our conclusions are different from previous work using only on-chip memory; we show that off-chip memory latency is not a significant challenge for soft systems; we present the technique of instruction *replay* to handle cache misses in soft multithreaded processors without stalling other threads; we show that, on a system of a given area with off-chip memory, single-threaded processors have a better instruction throughput than multithreaded processors, even for multiprocessors composed of either type of processor.

2. Experimental Framework

In this section we briefly describe our infrastructure for designing and measuring soft processors, our methodology for comparing soft processor designs, our compilation infrastructure, and the benchmark applications that we study.

Caches The Altera Stratix FPGA that we target provides three sizes of block-memory: M512 (512bits), M4K (4Kbits) and M-RAM (512Kbits). We use M512s to implement register files. In contrast with M-RAM blocks, M4K blocks can be configured to be read and written at the same time (using two ports), such that the read will return the previous value—hence, despite their smaller size, caches built with M4Ks typically out-perform those composed of M-RAMs, and we choose M4K-based caches for our processors.

Platform Our RTL is synthesized, mapped, placed, and routed by `Quartus 7.2` [18] using the default optimization settings.[1] The resulting soft processors are measured on the Transmogrifier platform [19], where we utilize one Altera Stratix FPGA `EP1S80F1508C6` device to (i) obtain the total number of execution cycles, and (ii) to generate a trace which is validated for correctness against the corresponding execution by an emulator (MINT [20]). Our memory controller connects a 64-bit-wide data bus to a 1Gbyte DDR SDRAM DIMM clocked at 133 MHz, and configured to transfer two 64-bit words (i.e., one cache line) on each memory access.

[1] We expect our conclusions to hold across different optimization settings, but for now we opt for the shorter synthesis times of the default configuration.

Table 1. EEMBC benchmark applications evaluated. ST stands for single-threaded and MT stands for multithreaded.

Category	Benchmark	Dyn. Instr. Counts (x10^6)	
		ST	MT
Automotive	A2TIME01	374	356
	AIFIRF01	33	31
	BASEFP01	555	638
	BITMNP01	114	97
	CACHEB01	16	15
	CANRDR01	38	35
	IDCTRN01	62	57
	IIRFLT01	88	84
	PUWMOD01	17	14
	RSPEED01	23	21
	TBLOOK01	149	140
Telecom	AUTCOR00DATA_2	814	733
	CONVEN00DATA_1	471	451
	FBITAL00DATA_2	2558	2480
	FFT00DATA_3	61	51
	VITERB00DATA_2	765	750
Networking	IP_PKTCHECKB4M	42	38
	IP_REASSEMBLY	385	324
	OSPFV2	49	33
	QOS	981	732

Measurement For Altera Stratix FPGAs, the basic logic element (LE) is a 4-input lookup table plus a flip-flop—hence we report the area of our soft processors in *equivalent LEs*, a number that additionally accounts for the consumed silicon area of any hardware blocks (e.g. multiplication or block-memory units). Even if a memory block is partially utilized by the design, the area of the whole block is nonetheless added to the total area required. For consistency, all our soft processors are clocked at 50 MHz and the DDR remains clocked at 133 MHz. The exact number of cycles for a given experiment is non-deterministic because of the phase relation between the two clock domains, a difficulty that is amplified when cache hit/miss behavior is affected. However, we have verified that the variations are not large enough to significantly impact our measurements.

Compilation and Benchmarks 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 [21] ISA; for example, for multithreaded processors we implement 3-operand multiplies (rather than MIPS Hi/Lo registers [3, 22]), and eliminate branch and load delay slots. Integer division is implemented in software. Table 1 shows the selected benchmarks from the EEMBC suite [23], avoiding benchmarks with significant file I/O that we do not yet support, along with the benchmarks dynamic instruction counts as impacted by different compiler settings. For systems with multiple threads and/or processors, we run multiple simultaneous copies of a given benchmark (i.e., similar to packet processing), measuring the time from the start of execution for the first copy until the end of the last copy.[2] In future work we hope to instead move towards implementing and analyzing fully-parallel applications with shared data and synchronization.

3. Integrating Uniprocessors with Off-Chip Memory

Past studies have shown that memory latency is not a concern for soft processors built with on-chip memory, since FPGA on-chip block-RAMs typically operate at higher speeds than the soft processors that use them [24, 25]. However, such systems can only support applications with limited data and instruction memory footprints. Soft-processor systems with caches and off-chip memory can support larger applications (such as the EEMBC benchmarks). In this section we analyze an implementation of such a system in detail, in particular examining memory latency as well as the breakdown of FPGA area devoted to parts of the system.

3.1. Uniprocessor Design

The specific uniprocessor design selected for our study was automatically generated by the SPREE processor generator [24, 25]. SPREE initially supported only on-chip memory, so we modified SPREE to export an external memory bus allowing the connection of a memory subsystem. We chose a 3-stage pipelined processor with full forwarding and a 1-bit branch history table for branch prediction as we found it to be the most area-efficient (good performance with low area). The instruction and data caches share access to the single-channel DDR controller. The caches have a 16-byte block size to match the word size of the DDR memory. The data cache implements a write-back, no-write-allocate write policy. The processor suffers a single pipeline stall on any branch misprediction or instance of a shift, multiply, load, or store instruction. Loads and stores can suffer varying additional stalls for cache misses, depending on the response of the memory subsystem.

The processor and caches are clocked together at 50 MHz while the DDR controller is clocked at 133 MHz. There are three main reasons for the reduced clock speed of the processor and caches: i) the original 3-stage pipelined processor with on-chip memory could only be clocked at 72 MHz on the slower speed grade Stratix FPGAs on the TM4; ii) adding the caches and bus handshaking further reduced the clock frequency to 64 MHz; and iii) to relax the timing constraints when crossing clock domains, we chose a 20 ns clock period which is a rational multiple of the 133 MHz (7.5 ns) DDR clock. In our evaluation in Section 6, we estimate the impact of higher processor clock frequencies that

[2] We verified that in most cases no thread gets significantly ahead of the others.

match the actual critical paths of the underlying circuits, and find that the results do not alter our conclusions.

3.2. Uniprocessor Area Breakdown

It is important to understand the relative area of components in a soft processor system. Our example uniprocessor system is comprised of the processor core (41.7%), 4KB direct-mapped L1 data cache (10.9%), 4KB direct-mapped L1 instruction cache (10.9%), and the rest of the system including memory controller, peripherals, and communication logic between the TM4 and Linux host (36.5%). Note that cache accounts for only a quarter of system area, despite the simplicity of the processor core. While this is quite different from conventional processors whose silicon area is typically dominated by cache, it is expected in FPGA technology since caches are composed mostly of SRAM storage and can be built by stitching together a few RAM blocks. Similarly, in contrast with "hard" systems, cache area is also dominated by area devoted to the memory controller and other peripherals.

3.3. Uniprocessor Memory Latency

Our system has a processor clock of 50MHz, a memory system clock of 133MHz, and a load-miss latency of only 8-cycles. If we assume a faster processor implementation with a clock matching that of memory (133MHz), then the load latency would be 21 cycles, which can be broken-down as follows. The processor uses a 3-cycle handshaking scheme to communicate the memory request to the DDR controller. Pipelining within the DDR controller and the row and column access latencies are responsible for a 9-cycle delay before the data is available at the pins of the FPGA.[3] Conversion from the 64-bit dual-data-rate signal to a 128-bit wide single-edge signal requires 2 cycles, followed by 3 cycles for phase realignment to the DDR controller's 133 MHz clock. Crossing back into the processor's clock domain with some handshaking then consumes an additional 4 cycles. Furthermore, our current memory controller implementation has room for improvement such as by: (i) setting the column access latency to 2 instead of 3; (ii) tracking open DRAM pages and saving unnecessary row access latency; (iii) fusing the single edge conversion, phase realignment, and clock crossing which together amount to a single clock crossing.

Our first key observation is therefore that off-chip memory latency for FPGA-based soft processors is not as significant as it is for ASICs and other "hard" processors, because the clock frequency of typical soft processors is much slower. While this may reduce the effect of the memory latency hiding offered by multithreading, it also reduces the penalties arising from cache collisions suggesting that we may be able to more easily scale the number of hardware threads.

Although our hardware platform is two generations old, we expect our results to hold in a Stratix III FPGA system with DDR3 SDRAM. Synthesis results of our uniprocessor on a Stratix III yields a clock frequency of 151 MHz. Conservatively, the DDR3 bus clock would be 400 MHz, thus preserving the memory-to-processor clock ratio of our current system. However the column access latency would have increased from 3 to 5-6 for the DDR3-based system. The extra 2-3 clock cycles would result in only 1 cycle of increased memory latency from the perspective of the processor. We believe that our conclusions hold even in the face of the extra cycle or so of memory latency.

Summary The small load miss latency observed in our soft processor (8 cycles) is at least 20 times slower than what is typically encountered in modern desktop ASIC processors [26], encouraging us to investigate scaling performance through larger numbers of threads—as we do in the next section—rather than pursuing better latency-tolerance techniques.

4. Scaling Uniprocessor Caches

Designers of soft systems are frequently concerned with trade-offs between area and performance. In particular, the question is how to best utilize additional area to improve performance. Possible non-trivial solutions include schemes for prefetching, or customizing instructions or the memory system to match the needs of a particular application. In this section we evaluate the impact of the more straightforward solution of increasing soft uniprocessor data cache size.

To demonstrate the impact on performance of increasing data cache size, in Figure 1 the *measured* line shows the geometric-mean speedup across our EEMBC benchmarks for varying direct-mapped data-cache sizes, relative to a 256B data cache. Compared to the 4KB data cache, a 256KB data cache provides only a 9.8% additional speedup at the cost of a 64-fold increase in area devoted to cache.

To model the impact of a given cache, we use

$$Speedup = CPI_{perfect} + (f_{ld} M_{ld} P_{ld}) + (f_{st} M_{st} P_{st}) \quad (1)$$

where $CPI_{perfect}$ is the cycles-per-instruction measured with a perfect memory system, and for loads and stores f is the frequency of memory references, M is the miss rate, and P is the miss penalty. Using the CPI values measured previously for processors with only on-chip memory [24] as an estimate, the frequency of memory references and miss rates reported by our instruction simulator, and miss penalties reported by the Quartus II SignalTap Logic Analyzer software, we plot the "modelled speedup" line shown in Fig-

[3] Note the controller uses a closed-page policy so that every request opens a DRAM row and then closes it.

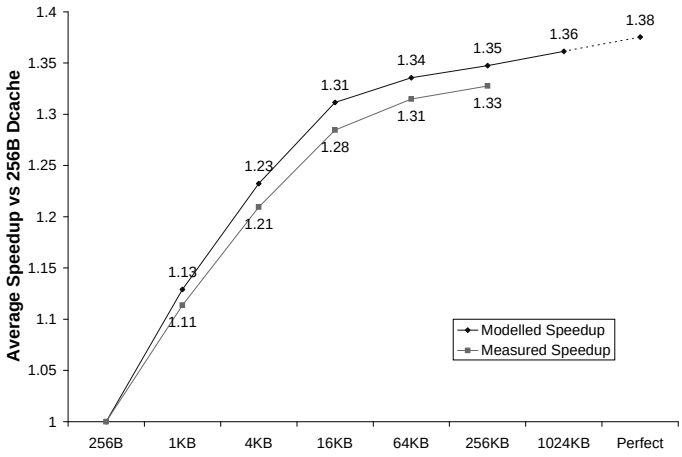


Fig. 1. Geometric mean speedup across our EEMBC benchmarks for varying direct-mapped data-cache sizes, relative to a 256B data cache. Speedup is both *measured* in real hardware and *modeled* according to Equation 1, both with a 64KB instruction-cache. For the modeled speedup, the *perfect* point shows the impact of an infinite-size data cache.

ure 1. We observe that the modelled speedup tracks the measured speedup very closely, with the modelled speedup being slightly larger since it models neither instruction misses nor bus contention. According to this model a perfect data cache improves performance only 11.6% over the 4KB data cache. Caches with increased associativity would be ineffective at reclaiming this fraction because they would increase the cache access latency.

The diminishing returns seen in the larger cache sizes and the idealized cache motivate the exploration of other means of translating additional area into improved performance other than increasing cache performance. In this work we focus on exploiting additional threads, in particular by considering multithreaded processors in the next section, and multiprocessors composed of either single-threaded or multithreaded processors in the subsequent section.

5. Integrating Multithreaded Processors with Off-Chip Memory

As prior work has shown [3, 4], multithreaded soft processors can hide processor stalls while saving area, resulting in more area-efficient soft systems than those composed of uniprocessors or multiprocessors. Our multithreaded soft processors support fine-grained multithreading, where an instruction for a different thread context is fetched each clock cycle in a round-robin fashion. Such a processor requires the register file and program counter to be logically replicated per thread context. However, since consecutive instructions in the pipeline are from independent threads, we eliminate the need for data hazard detection logic and forwarding lines—assuming that there are at least $N-1$ threads for an N-stage pipelined processor [3,4]. Our multithreaded processors have a 5-stage pipeline that never stalls: this pipeline depth was found to be the most area-efficient for multithreading in earlier work [3]. In this section we describe the challenges in connecting a multithreaded soft processor to off-chip memory through caches, and our respective solutions.

5.1. Reducing Cache Conflicts

The workload we assume for this study is comprised of multiple copies of a single task (i.e., similar to packet processing), hence instructions and an instruction cache are easily shared between threads without conflicts. However, since the data caches we study are direct-mapped, when all the threads access the same location in their respective data sections, these locations will all map to the same cache entry, resulting in pathologically bad cache behavior. As a simple remedy to this problem we pad the data sections for each thread such that they are staggered evenly across the data cache, in particular by inserting multiples of padding equal to the cache size divided by the number of thread contexts sharing the cache. However, doing so makes it more complicated to share instruction memory between threads: since data can be addressed relative to the global pointer [21], we introduce a short thread-specific initialization routine that adjusts the global pointer by the padding amount; there can also be static pointers and offsets in the program, that we must adjust to reflect the padding. We find that applying this padding increases the throughput of our base multithreaded processor by 24%, hence we apply this optimization for all of our experiments.

5.2. Tolerating Miss Latency via Replay

When connected to an off-chip memory through caches, a multithreaded processor will ideally not stall other threads when a given thread suffers a multiple-cycle cache miss. In prior work, Fort *et. al.* [4] use a FIFO queue of loads and stores, while Moussali *et. al.* [12] use an instruction scheduler to issue ready instructions from a pool of threads. For both instruction and data cache misses we implement a simpler method requiring little additional hardware that we call instruction *replay*. The basic idea is as follows: whenever a memory reference instruction suffers a cache miss, that instruction *fails*—i.e., the program counter for that thread is not incremented. Hence that instruction will execute again (i.e., replay) when it is that thread context's turn again, and the cache miss is serviced while the instruction is replaying. Other threads continue to make progress, while the thread that suffered the miss fails and replays until the memory reference is a cache hit. However, since our processors can handle only a single outstanding memory reference, if a second thread suffers a cache miss it will itself fail and replay

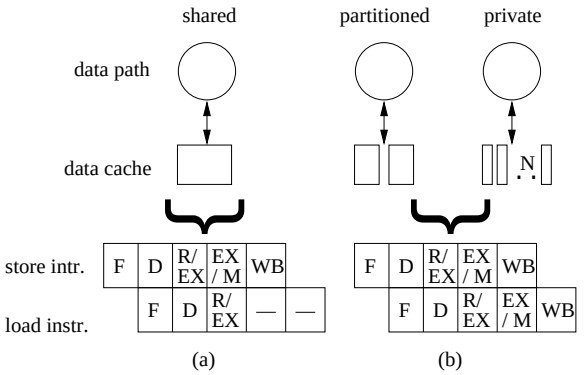


Fig. 2. Cache organizations and the corresponding impact on the execution of a write hit from one thread followed by a load from a consecutive thread: (a) a shared data cache, for which the load is aborted and later replays; (b) partitioned and private caches, for which the load succeeds.

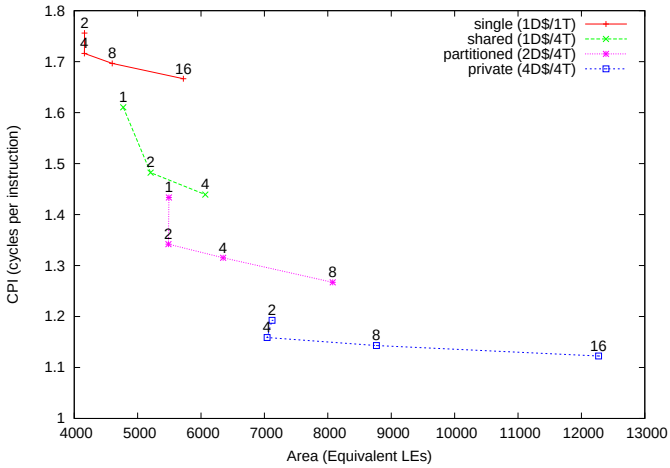


Fig. 3. CPI versus area for the *single* threaded processor and for the multithreaded processors with *shared*, *partitioned*, and *private* caches. 1D$/4T means there is one data cache and 4 threads total. Each point is labeled with the total cache capacity in KB available per thread.

until its miss is serviced.

To safely implement the instruction replay technique we must consider how cache misses from different threads might interfere. First, it is possible that one thread can load a cache block into the cache, and then another thread replaces that block before the original thread is able to use it. Such interference between two threads can potentially lead to live-lock. However, we do not have to provide a solution to this problem in our processors because misses are serviced in order and the miss latency is guaranteed to be greater than the latency of a full round-robin of thread contexts—hence a memory reference suffering a miss is guaranteed to succeed before the cache line is replaced. However, a second possibility is one that we must handle: the case of a memory reference that suffers a data cache miss, for which the corresponding instruction cache block is replaced before the memory reference instruction can replay. This subtle pathology can indeed result in live-lock in our processors, so we prevent it by saving a copy of the last successfully fetched instruction for each thread context.

5.3. Cache Organization

Each thread has its own data section, hence despite our padding efforts a shared data cache can still result in conflicts. A simple solution to this problem is to increase the size of the shared data cache to accommodate the aggregate data set of the multiple threads, although this reduces the area-saving benefits of the multithreaded design. Furthermore, since our caches are composed of FPGA memory blocks which have only two ports (one connected to the processor, one connected to the DRAM channel), writes take two cycles: one cycle to lookup and compare with the tag, and another cycle to perform the write (on a hit). As illustrated in Figure 2(a), this can lead to further contention between consecutive threads in a multithreaded processor that share a cache: if a second consecutive thread attempts a memory reference directly after a write-hit, we apply our failure/replay technique for the second thread, rather than stall that thread and subsequent threads.

Rather than increasing the size of the shared cache, we consider two alternatives. The first is to have *private caches* such that each thread context in the multithreaded processor accesses a dedicated data cache. The second, if the number of threads is even, is to have *partitioned caches* such that non-consecutive threads share a data cache—for example, if there are four threads, threads 1 and 3 would share a cache and threads 2 and 4 would share a second cache. As shown in Figure 2(b), both of these organizations eliminate port contention between consecutive threads, and reduce (partitioned) or eliminate (private) cache block conflicts between threads.

6. Scaling Multithreaded Processor Caches

In this section we compare single-threaded and multithreaded soft processors, and study the impact of cache organization and thread count on multithreaded processor performance and area efficiency.

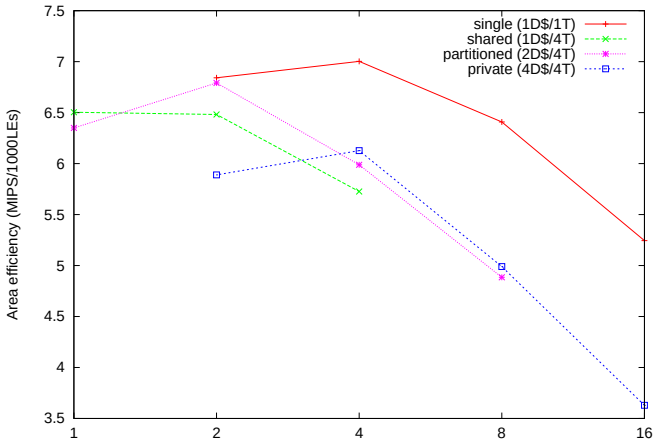
In Figure 3 we plot performance versus area for the single threaded processor and the three possible cache organizations for multithreaded processors (shared, partitioned, and private), and for each we vary the sizes of their caches. For performance we plot cycles-per-instruction (*CPI*), which is computed as the total number of cycles divided by the total

number of instructions executed; we use this metric as opposed to simply execution time because the single-threaded and multithreaded processors execute different numbers of threads, and because the compilation of benchmarks for the single-threaded and multithreaded processors differ (as shown in Table 1). CPI is essentially the inverse of throughput for the system, and this is plotted versus the area in equivalent LEs for each processor design—hence the most desirable designs minimize both area and CPI.

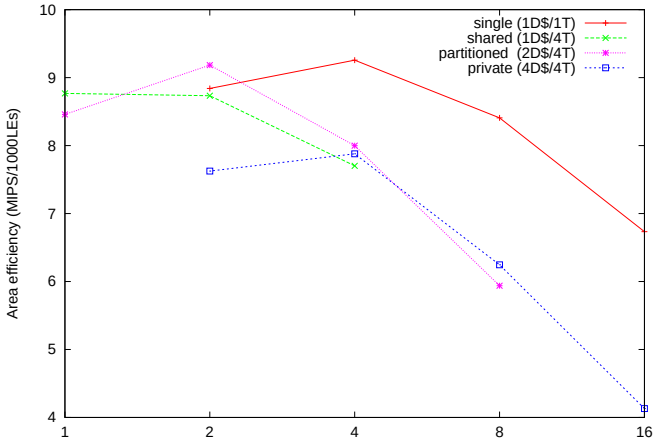
We first observe that the single-threaded and different multithreaded processor designs with various cache sizes allow us to span a broad range of the performance/area space, giving a system designer interested in supporting only a small number of threads the ability to scale performance by investing more resources. The single-threaded processor is the smallest but provides the worst CPI, and this is improved only slightly when the cache size is doubled (from 2KB to 4KB). Of the multithreaded processors, the shared, partitioned, and private cache designs provide increasing improvements in CPI at the cost of corresponding increases in area. The shared designs outperform the single-threaded processor because of the reduced stalls enjoyed by the multithreaded architecture. The partitioned designs outperform the shared designs as they eliminate replays due to contention. The private cache designs provide the best performance as they eliminate replays due to both conflicts and contention, but for these designs performance improves very slowly as cache size increases.

There are several instances where increasing available cache appears to cost no additional area: similar behavior is seen for the single-threaded processor moving from 2KB to 4KB of cache and for the partitioned multithreaded processor moving from 1KB to 2KB of cache per thread. This is because the smaller designs partially utilize M4K memories, while in the larger designs they are more fully utilized—hence the increase appears to be free since we account for the entire area of an M4K regardless of whether it is fully utilized. For the private-cache multithreaded designs, moving from 2KB to 4KB of cache per thread actually saves a small amount of area, for similar reasons plus additional savings in LEs due to fortuitous mapping behavior.

To better understand the trade-off between performance and area for different designs, it is instructive to plot their area efficiency as shown in Figure 4(a). We measure area efficiency as millions of instructions per second (MIPS) per 1000 LEs. The single-threaded processors are the most area-efficient, in contrast with previous work comparing similar processors with on-chip memory and without caches [3]. The partitioned design with 2KB of cache per thread is nearly as area-efficient as the corresponding single-threaded processor. The shared-cache designs with 1KB and 2KB of cache per thread are the next most area-efficient, with the private-cache designs being the least area-efficient. These



(a) 50MHz processor clock frequency.



(b) Maximum allowable processor clock frequency.

Fig. 4. Area efficiency versus total cache capacity per thread for the single-threaded processor and the multithreaded processors, reported using (a) the implemented clock frequency of 50MHz, and (b) the maximum allowable clock frequency per processor design.

results tell us to expect the single-threaded and partitioned-cache multithreaded designs to scale well, as they provide the best performance per area invested.

Due to limitations of the Transmogrifier platform, all of our processors are actually clocked at 50MHz, while their maximum possible frequencies (i.e., f_{max}) are on average 65MHz. To investigate whether systems measured using the true maximum possible frequencies for the processors would lead to different conclusions, we estimate this scenario in Figure 4(b). We observe that the relative trends are very similar, with the exception of the single-threaded processor with 2KB of cache for which the area efficiency drops below that of the corresponding partitioned design to close to that of the shared cache design.

Impact of Increasing Thread Contexts Our multithreaded processors evaluated so far have all implemented a minimal number of threads contexts: four thread contexts for five pipeline stages. To justify this choice, we evaluated multithreaded processors with larger numbers of threads for the different cache designs and for varying amounts of available cache per thread. For shared and partitioned designs we found that increasing the number of thread contexts (i) increases the CPI, due to increased contention and conflicts, and (ii) increases area, due to hardware support for the additional contexts. Since the private cache designs eliminate all contention and conflicts, there is a slight CPI improvement as area increases significantly with additional thread contexts. These results confirmed that the four-thread multithreaded designs are the most desirable.

Summary We have observed that multithreaded processors provide significant throughput improvements over single-threaded processors when port contention is removed between the threads (i.e., with partitioned and private data cache organizations), even when the multithreaded processor has less total data cache capacity. The most area-efficient multithreaded processors are larger than the most area-efficient single threaded processors of same cache capacity because (i) the multithreaded pipeline is longer (five stages rather than three), and (ii) the additional hardware overheads of supporting multithreading for memory references (partitioning, checks for conflicts, support for replay and deadlock avoidance). Finally, we demonstrated that supporting the minimum number of thread contexts (four thread contexts for five pipeline stages) in a multithreaded processor minimizes both contention and area.

7. Scaling Multiprocessors

For systems with larger numbers of threads available, another alternative for scaling performance is to instantiate multiple soft processors. In this section we explore the design space of soft multiprocessors, with the goals of maximizing (i) performance, (ii) utilization of the memory channel, and (iii) utilization of the resources of the underlying FPGA. To support multiple processors we augment our DRAM controller with an arbiter that serializes requests by queuing up to one request per processor; note that this simple interconnect architecture does not impact the clock frequencies of our processors.

In Figure 5 we plot CPI versus area for multiprocessors composed of single-threaded or multithreaded processors; we replicate the processor designs that were found to be the most area-efficient according to Figure 4(a). For each multiprocessor design, each design point has double the number of processors as the previous, with the exception of the largest (rightmost) for which we plot the largest design supported by the FPGA—in this case the design that has exhausted the M4K block memories.

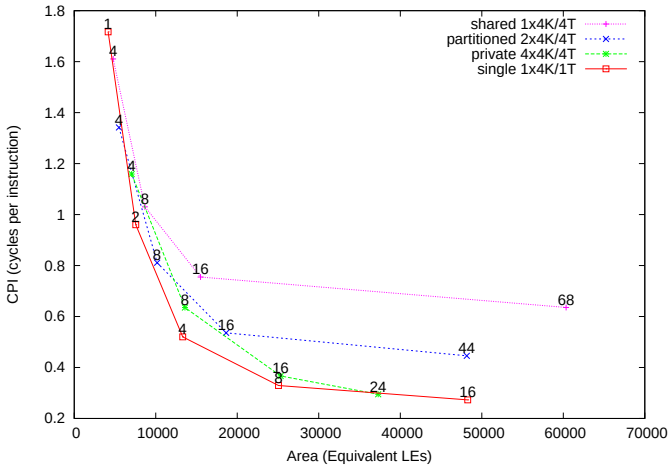


Fig. 5. CPI versus area for multiprocessors composed of single-threaded and multithreaded processors. 1x4K/4T means that each processor has one 4KB data cache and 4 thread contexts, and each point is labeled with the total number of thread contexts supported.

Our first and most surprising observation is that the Pareto frontier (the set of designs that minimize CPI and area) is mostly comprised of single-threaded multiprocessor designs, many of which out-perform multithreaded designs that support more than twice the number of thread contexts. For example, the 16-processor single-threaded multiprocessor has a lower CPI than the 44-thread-context partitioned-cache multithreaded multiprocessor of the same area. We will pursue further insight into this result later in this section. For the largest designs, the private-cache multithreaded multiprocessor provides the second-best overall CPI with 24 thread contexts (over 6 processors), while the partitioned and shared-cache multithreaded multiprocessor designs perform significantly worse at a greater area cost.

Sensitivity to Cache Size Since the shared and partitioned-cache multithreaded designs have less cache per thread than the corresponding private-cache multithreaded or single-threaded designs, in Figure 6 we investigate the impact of doubling the size of the data caches (from 4KB to 8KB per cache) for those two designs. We observe that this improves CPI significantly for the shared-cache designs and more modestly for the partitioned cache design, despite the fact that the 4KB designs were the most area-efficient (according to Figure 4(a)). Hence we evaluate the 8KB-cache shared and partitioned-cache designs in subsequent experiments.

Per-Thread Efficiency In this section we try to gain an understanding of how close to optimally each of our architectures performs—i.e., how close to a system that experiences no stalls. The optimal CPI is 1 for our single-threaded pro-

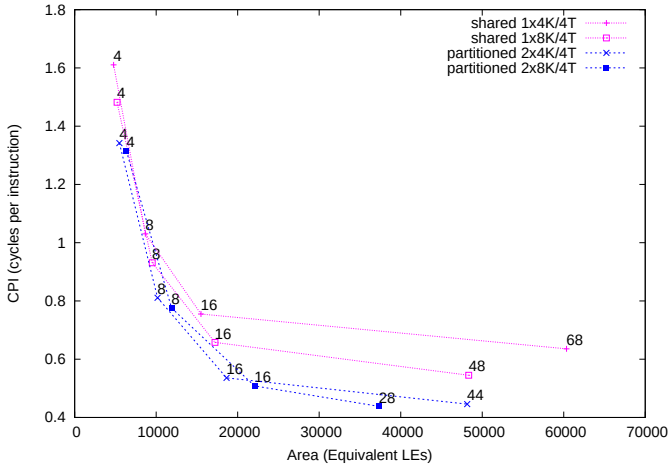
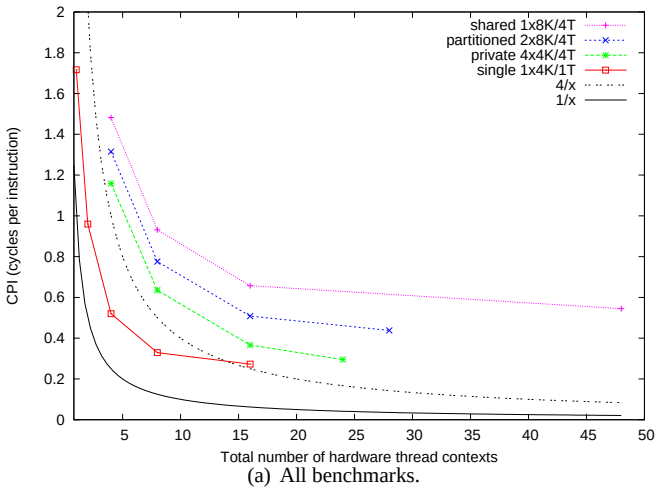


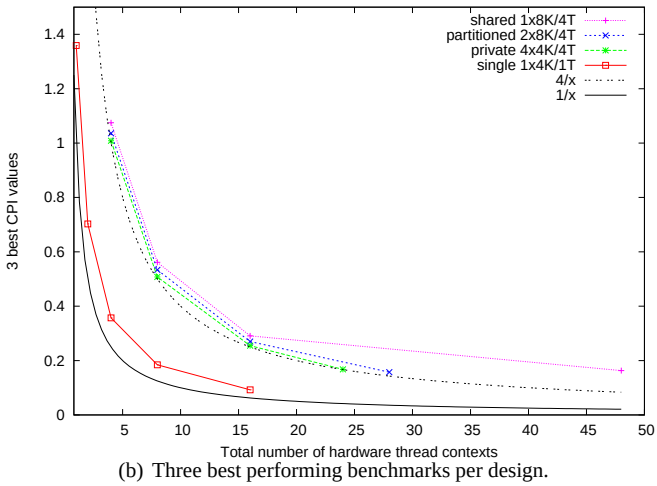
Fig. 6. CPI versus area for the shared and partitioned designs as we increase the size of caches from 4KB to 8KB each. Each point is labeled with the total number of thread contexts supported.

cessor, and hence $1/X$ for a multiprocessor composed of X single-threaded processors. For one of our multithreaded processors the optimal CPI is also 1, but since there are four thread contexts per processor, the optimal CPI for X multithreaded processors is $4/X$. In Figure 7(a) we plot CPI versus total number of thread contexts for our single and multithreaded designs, as well as the two ideal curves (as averaged across all of our benchmarks). As expected, for a given number of threads the single-threaded processors exhibit better CPI than the corresponding multithreaded designs. However, it is interesting to note that the private-cache multithreaded designs perform closer to optimally than the single-threaded designs. For example, with 16 threads (the largest design), the single-threaded multiprocessor has a CPI that is more than 4x greater than optimal, but regardless this design provides the lowest CPI across any design. This graph also illustrates that private-cache designs outperform partitioned-cache designs which in turn outperform shared-cache designs.

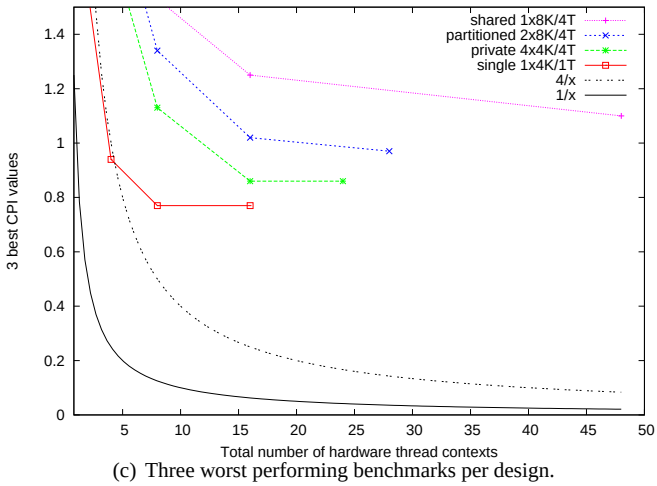
Potential for Customization A major advantage of soft systems is the ability to customize the hardware to match the requirements of the application—hence we are motivated to investigate whether the multithreaded designs might dominate the single-threaded design for certain applications. However, we find that this is not the case. To summarize, in Figures 7(b) and 7(c) we plot CPI versus total number of thread contexts for the three best performing and three worst performing benchmarks per design, respectively. For neither extremity do the multithreaded designs outperform the single-threaded designs. Looking at Figure 7(b), we see that for the best performing benchmarks the private-cache mul-



(a) All benchmarks.



(b) Three best performing benchmarks per design.



(c) Three worst performing benchmarks per design.

Fig. 7. CPI versus total thread contexts across all benchmarks (a), and the three best (b) and worst (c) performing benchmarks per design.

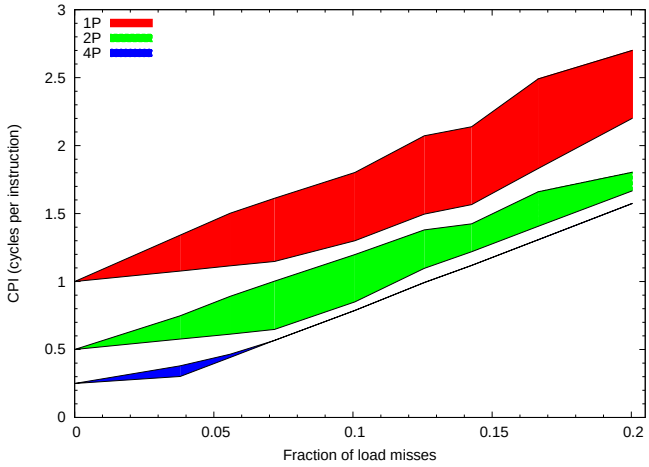
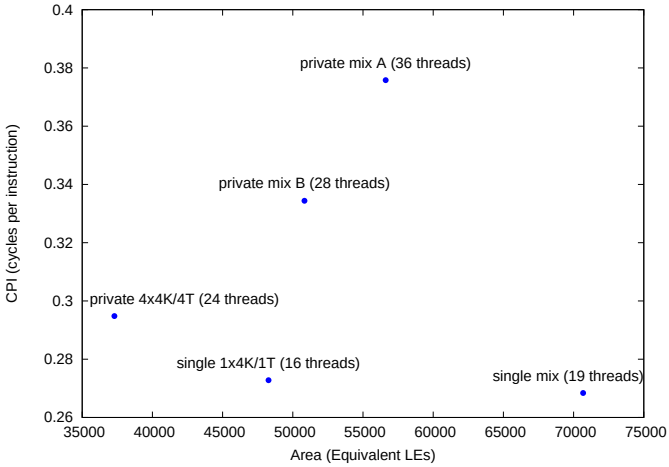


Fig. 8. CPI versus fraction of load misses. For each surface the upper edge plots the single-threaded multiprocessor designs (4KB data cache per processor) and the lower edge plots the private-cache multithreaded multiprocessor designs (4KB data cache per thread context).



Composition of Heterogeneous Multiprocessors			
	Private mix A	Private mix B	Single mix
M4K-based	mt_private x 5	mt_private x 5	st x 15
M-RAM-based	mt_shared x 4	mt_partitioned x 2	st x 4
Total threads	36	28	19

Fig. 9. CPI versus area for our two best-performing maximal designs (the 16-thread-context single-threaded design and the 24-thread-context private-cache multithreaded design), and for those designs extended with processors with M-RAM-based caches.

tithreaded designs perform nearly optimally. In the worst cases (Figure 7(c)), the single-threaded designs maintain their dominance, despite the 16-processor design performing slightly worse than the 8-processor design.

Understanding the Single-Threaded Advantage To clarify the advantage of the single-threaded multiprocessor designs, we use a synthetic benchmark that allows us to vary the density of load misses. In particular, this benchmark consists of a thousand-instruction loop comprised of loads and no-ops, and the loads are designed to always miss in the data cache. This benchmark allows us to isolate the impact of load misses since they can be the only cause of stalls in our processors. In Fig 8 we compare the CPI for single-threaded designs with the private-cache multithreaded designs as the fraction of load misses increases. In particular, for a certain number of processors we plot a surface such that the top edge of the surface is the single-threaded design and the bottom edge is the corresponding private-cache multithreaded design. Hence the three surfaces show this comparison for designs composed of 1, 2, and 4 processors. Looking at the surface for a single processor, as the fraction of load misses increases the multithreaded processor (bottom edge) has a somewhat consistent CPI advantage (of about 0.5 at 10% misses) over the single-threaded processor (top edge). However, this advantage narrows as the number of processors increases, and for four processors the multithreaded designs have only a negligible advantage over the single-threaded designs—and the multithreaded processors have greater area requirements than their single-threaded counterparts.

Exploiting Heterogeneity In contrast with ASIC designs, FPGAs provide limited numbers of certain resources, for example block memories. This leads to an interesting difference when targeting an FPGA as opposed to an ASIC: replicating only the same design will eventually exhaust a certain resource while under-utilizing others. For example, our largest multiprocessor design (the 68-thread shared-cache multithreaded design) uses 99% of the M4K block memories but only 43% of the available LEs. Hence we are motivated to exploit heterogeneity in our multiprocessor design to more fully utilize all of the resources of the FPGA—in this case we consider adding processors with M-RAM based caches despite the fact that individually they are less efficient than their M4K-based counterparts. In particular, we extend our two best-performing multiprocessor designs with processors having M-RAM-based caches as shown in Figure 9. In this case, extending the multithreaded processor with further processors does not improve CPI but only increases total area. For the single-threaded case the heterogeneous design improves CPI slightly but at a significant cost in area—however, this technique does allow us to go beyond the previously maximal design.

Summary Multithreaded and single-threaded processors both have an ideal CPI equal to one, although one of our multithreaded processors requires four threads to achieve that

CPI; hence for an equal total number of threads, single-threaded processors have the most favorable lower bound on CPI. However, even when threads are abundant, multithreaded processors are larger and suffer more frequent cache misses due to cache contention than single-threaded processors. Overall, when scaling-up multiprocessors based on multithreaded and single-threaded processors, we find that those based on multithreaded processors exhaust all memory resources on the FPGA or saturate the memory bus before they can outperform those based on single-threaded processors.

8. Conclusions

In this paper we explored architectural options for scaling the performance of soft systems, focussing on the organization of processors and caches connected to a single off-chip memory channel, for workloads composed of many independent threads. Our investigation of real FPGA-based processor, multithreaded processor, and multiprocessor systems has led to a number of interesting conclusions. First, we found that off-chip memory latency is not a significant challenge for FPGA-based systems, and that a small direct-mapped cache is sufficient to capture much of the benefit of an ideal cache. We showed that multithreaded designs help span the throughput/area design space, and that private-cache based multithreaded processors offer the best performance. Looking at multiprocessors we found that designs based on single-threaded processors perform the best for a given total area, followed closely by private-cache multithreaded multiprocessor designs. We demonstrated that as the number of processors increases, multithreaded processors lose their latency-hiding advantage over single-threaded processors, as both designs become bottlenecked on the memory channel. Finally, we showed the importance of exploiting heterogeneity in FPGA-based multiprocessors to fully utilize a diversity of FPGA resources when scaling-up a design.

9. References

[1] Altera Corp., "Nios II C2H compiler user guide v7.2," 2007.

[2] Xilinx Inc., "AccelDSP synthesis tool v9.2.01," 2007.

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

[4] B. Fort, D. Capalija, Z. G. Vranesic, and S. D. Brown, "A multithreaded soft processor for SoPC area reduction," in *Proc. of FCCM '06*, DC, USA, 2006, pp. 131–142.

[5] P. Yiannacouras and J. Rose, "A parameterized automatic cache generator for FPGAs," in *Proc. of FPT'03*, December 2003, pp. 324– 327.

[6] Altera Corp., "Nios II Processor Reference v7.2," 2007.

[7] Xilinx Inc., "MicroBlaze Processor Reference v8.0," 2007.

[8] A. Tumeo, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto, "A design kit for a fully working shared memory multiprocessor on FPGA," in *Proc. of GLSVLSI'07*. New York, NY, USA: ACM, 2007, pp. 219–222.

[9] M. Saldaña, L. Shannon, and P. Chow, "The routability of multiprocessor network topologies in FPGAs," in *Proc. of FPGA'06*. New York, NY, USA: ACM, 2006, pp. 232–232.

[10] Y. Jin, N. Satish, K. Ravindran, and K. Keutzer, "An automated exploration framework for FPGA-based soft multiprocessor systems," in *Proc. of CODES+ISSS '05*. New York, NY, USA: ACM, 2005, pp. 273–278.

[11] M. G. et al., " MiBench: A free, commercially representative embedded benchmark suite," in *Proc. of WWC '01*, 2001.

[12] R. Moussali, N. Ghanem, and M. A. R. Saghir, "Supporting multithreading in configurable soft processor cores," in *Proc. of CASES'07*, NY, USA, 2007, pp. 155–159.

[13] 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*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 160–161.

[14] SGI, "SGI RASC RC100 Blade," 2008.

[15] Cray Inc., "Cray XD1TM supercomputer release 1.3," 2005.

[16] XtremeData Inc., "XD2000F FPGA coprocessor for AMD socket F," 2007.

[17] A. Kulmala, E. Salminen, and T. D. Hamalainen, "Instruction memory architecture evaluation on multiprocessor FPGA MPEG-4 encoder," in *Proc. of DDECS'07*, April 2007.

[18] Altera Corp., "Quartus II," San Jose, CA, USA, Altera.

[19] J. Fender, J. Rose, and D. Galloway, "The Transmogrifier-4: an FPGA-based hardware development system with multi-gigabyte memory capacity and high host and memory bandwidth," in *Proc. of FPT'05*, december 2005, pp. 301– 302.

[20] 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.

[21] 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.

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

[23] "Embedded Microprocessor Benchmark Consortium (EEMBC)," http://www.eembc.org.

[24] P. Yiannacouras, J. Rose, and J. G. Steffan, "The microarchitecture of FPGA-based soft processors," in *Proc. of CASES'05*, September 2005, pp. 202–212.

[25] P. Yiannacouras, J. G. Steffan, and J. Rose, "Application-specific customization of soft processor microarchitecture," in *Proc. of FPGA'06*, Monterey, CA, 2006, pp. 201–210.

[26] D. Besedin, "Platform benchmarking with RightMark memory analyzer," http://www.digit-life.com, 2004.