Application-Specific Customization of Soft Processor Microarchitecture

Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
{yiannac,steffan,jayar}@eecg.utoronto.ca

ABSTRACT

A key advantage of *soft* processors (processors built on an FPGA programmable fabric) over hard processors is that they can be customized to suit an application program's specific software. This notion has been exploited in the past principally through the use of application-specific instructions. While commercial soft processors are now widely deployed, they are available in only a few microarchitectural variations. In this work we explore the advantage of tuning the processor's microarchitecture to specific software applications, and show that there are significant advantages in doing so.

Using an infrastructure for automatically generating soft processors that span the area/speed design space (while remaining competitive with Altera's Nios II variations), we explore the impact of tuning several aspects of microarchitecture including: (i) hardware vs software multiplication support; (ii) shifter implementation; and (iii) pipeline depth, organization, and forwarding. We find that the processor design that is fastest overall (on average across our embedded benchmark applications) is often also the fastest design for an individual application. However, in terms of area efficiency (i.e., performance-per-area), we demonstrate that a tuned microarchitecture can offer up to 30% improvement for three of the benchmarks and on average 11.4% improvement over the fastest-on-average design. We also show that our benchmark applications use only 50% of the available instructions on average, and that a processor customized to support only that subset of the ISA for a specific application can on average offer 25% savings in both area and energy. Finally, when both techniques for customization are combined we obtain an average improvement in performance-per-area of 25%.

1. INTRODUCTION

Instruction set processors are an integral part of all modern digital systems, including systems now fully implemented with FPGAs. While some vendors have included fixed, *hard* processors on their FPGA die, there has been significant uptake [15,16] of *soft* processors [3,4] which are constructed in the programmable fabric itself. While soft processors cannot match the performance, area, or power consumption of a hard processor, their key advantage is the flexibility they present in allowing different numbers of processors and specialization to the application through special instructions.

Specialization can also occur by selecting a different microarchitecture for a specific application, and that is the focus of this paper. We suggest that this kind of specialization can be applied in two different contexts:

1. When an embedded processor is principally running a *single* application for which both performance and area are important factors—if a particular microarchitecture can be shown to be advantageous over what would otherwise be a single, general-purpose microarchitecture, there is a win. Moreover, this win is "free" in a reconfigurable FPGA environment except for the exploration time spent finding these customizations at compile time.

2. When an embedded processor executes many applications on a reconfigurable FPGA, there may be value in reducing the area consumed by the processor, either to make room for something else that is dynamically required or to reduce its power consumption, or possibly to increase the performance of the processor on a per-program basis. Of course, these circumstances have to be such that the reconfiguration itself does not obviate the gain in benefit.

In this paper we show that the impact of varying several microarchitectural axes varies widely across different embedded applications. We then demonstrate that a soft processor microarchitecture that has been tuned to a specific application can be significantly more area-efficient than the best-on-average processor, and that typical embedded applications use less than half of the available Instruction Set Architecture (ISA), despite the use of an unmodified version of `gcc` as a compiler. By removing the hardware associated only with these unused instructions (what we call *ISA subsetting*) we can obtain significant area and power savings. Finally, we show that microarchitectural tuning and ISA subsetting are complementary.

This paper is organized as follows: in the following subsection we review related work. Section 2 briefly describes our system for automatically generating soft processors with different micro-architecture. Section 3 describes our method of measuring the area, speed and power consumption of the generated processors. Section 4 presents the impact of specialization for these metrics, and Section 5 concludes.

1.1 Related Work

Commercial customizable processors for inclusion in ASICs are available from Tensilica [7] and ARC [1] which allow designers to tune the processor with additional hardware instructions to better match their application requirements. Altera Nios [4] and Xilinx Microblaze [3] market soft processors that also allow customized instructions or hardware, and are typically available in only a few microarchitectural variants.

The CUSTARD [9] customizable threaded soft processor is an FPGA implementation of a parameterizable core supporting the following options: different number of hardware threads and types, custom instructions, branch delay slot, load delay slot, forwarding, and register file size. While the available architectural axes seem interesting the results show large overheads in the processor design: clock speed varied only between 20 and 30 MHz on the 0.15 micron XC2V2000, and the single-threaded base processor consumed 1800 slices while the commercial Microblaze typically consumes less than 1000 slices on the same device.

The PEAS-III project [11] focuses on ISA design and hardware software co-design, and proposes a system which generates a synthesizable RTL description of a processor from a clock-based micro-operation description of each instruction. Although PEAS-III enables a broad range of exploration, it requires changes to the description of many instructions to produce a small structural change to the architecture. PEAS-III was used [11] to conduct a synthesis-driven exploration which explored changing the multiply/divide unit to sequential (34-cycles), and then adding a MAC (multiply-accumulate) instruction. The results compared the area and clock frequency reported by the synthesis tool.

Mishra and Kejariwal augmented the EXPRESSION Architecture Description Language (ADL) to include RTL generation enabling synthesis-driven architectural exploration [14]. The generated processors were shown to have significant overhead versus an already poorly designed academic processor meant only for simulation not synthesis. A small exploration was then conducted [14] for an FFT benchmark where the number of functional units were increased from 1 to 4, the number of stages in the multiplier unit were increased from 1 to 4, and `sin`/`cos` instructions were added to the instruction set. These customizations seemed promising but the analysis did not consider the entire processor (measurements were only made for the execution stage). Moreover, this exploration was performed for traditional hard processors, with no reference to FPGA-based processors.

2. GENERATING SOFT PROCESSORS WITH SPREE

Our long-term research agenda is to be able to automatically navigate the soft processor design space, and to make intelligent application-specific architectural trade-offs based on a full understanding of soft processor architecture. The work presented in this paper builds on the *Soft Processor Rapid Exploration Environment* (SPREE) [19], a system we developed to allow the fast and easy generation of a large number of soft processor designs. In particular, SPREE takes as input a high-level, text-based description of the target ISA and datapath, and generates an RTL description of a working soft processor (SPREE is described in more detail later in Section 2). SPREE was initially used to explore the microarchitecture of soft processors [19], and allowed us to study the impact on area, performance, and energy of a given microarchitectural feature or trade-off across a set of benchmark applications. For example, we are able to vary the shifter implementation, hardware versus software support for multiplication, and the pipeline depth and forwarding capabilities. In this work we explore microarchitectural "wins" on a per-application basis by customizing the processor to the given application.

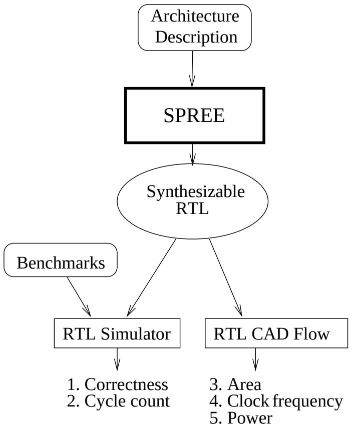

Figure 1: Overview of the SPREE system.

Figure 1 shows an overview of the SPREE system. Taking a high-level description of an architecture as input, SPREE automatically generates synthesizable RTL (in Verilog). We then simulate the RTL description on benchmark applications to both ensure correctness of the processor, and to measure the total number of cycles required to execute each application. The RTL is also processed by CAD tools which accurately measure the area, clock frequency, and power of the generated soft processor. The following discussion describes how SPREE generates a soft processor in more detail—complete descriptions of SPREE are available in previous publications [17–19].

2.1 Input: The Architecture Description

The input to SPREE is a description of the desired processor, composed of textual descriptions of both the target ISA and the processor datapath. Each instruction in the ISA is described as a directed graph of generic operations (GENOPs), such as `ADD`, `XOR`, `PCWRITE`, `LOADBYTE`, and `REGREAD`. The graph indicates the flow of data from one GENOP to another required by that instruction. SPREE provides a library of basic components (e.g., a register file, adder, sign-extender, instruction fetch unit, forwarding line, and more). A processor datapath is described by the user as an interconnection of these basic components. As we describe below, SPREE ensures that the described datapath is capable of implementing the target ISA.

2.2 Generating a Soft Processor

From the above inputs, SPREE generates a complete Verilog RTL model of the desired processor in three phases: (i) datapath verification, (ii) datapath instantiation, and

(iii) control generation. In the datapath verification phase, SPREE compares the submitted ISA description and datapath description, ensuring that the datapath is functionally capable of executing all of the instructions in the ISA description. The datapath instantiation phase automatically generates multiplexers for sinks with multiple sources and eliminates any components that are not required by the ISA. Finally, the control generation phase implements the control logic necessary to correctly operate the datapath, and emits the Verilog descriptions of the complete processor design.

2.3 Limitations

There are several limitations to the scope of soft processor microarchitectures that we study in this paper. For now we consider simple, in-order issue processors that use only on-chip memory and hence have no cache. Since the relative speeds of memory and logic on a typical FPGA are much closer than for a hard processor chip, we are less motivated to explore a memory hierarchy for soft processors. The largest FPGA devices have more than one megabyte of on chip memory which is adequate for the applications that we study in this paper—however, in the future we do plan to broaden our application base to those requiring off-chip RAM and caches. We do not yet include support for dynamic branch prediction, exceptions, or operating systems. Finally, in this paper we do not add new instructions to the ISA (we restrict ourselves to a subset of MIPS-I) nor have we modified the compiler, with the exception of evaluating software versus hardware support for multiplication due to the large impact of this aspect on cycle time and area.

3. EXPERIMENTAL FRAMEWORK

Having described the SPREE system in the previous section, we now describe our framework for measuring and comparing the soft processors it produces. We present methods for employing FPGA CAD tools, a methodology for measuring and comparing soft processors (including variations of a commercial soft processor), and the benchmark applications that we use to do so.

3.1 FPGAs, CAD, and Soft Processors

While SPREE itself emits Verilog which is synthesizable to any target FPGA architecture, we have selected Altera's Stratix [12] device for performing our FPGA-based exploration. The library of processor components thus targets Stratix I FPGAs. We use Quartus II v4.2 CAD software for synthesis, technology mapping, placement and routing. We synthesize all designs to a Stratix EP1S40F780C5 device (a middle-sized device in the family, with the fastest speed grade) and extract and compare area, clock frequency, and power measurements as reported by Quartus.

We have taken the following measures to counteract variation caused by the non-determinism of CAD tool output: (i) we have coded our designs structurally to avoid the creation of inefficient logic from behavioral synthesis; (ii) we have experimented with optimization settings and ensured that our conclusions do not depend on them, and (iii) for the area and clock frequency of each soft processor design we determine the arithmetic mean across 10 seeds (different initial placements before placement and routing) so that we are 95% confident that our final reported value is within 2% of the true mean.

3.2 Metrics for Measuring Soft Processors

To measure area, performance, and energy, we must decide on an appropriate set of specific metrics. For an FPGA, one typically measures area by counting the number of resources used. In Stratix, the main resource is the *Logic Element* (LE), where each LE is composed of a 4-input *lookup table* (LUT) and a flip flop. Other resources, such as the hardware multiplier block, and memory blocks can be converted into an equivalent number of LEs based on the relative areas of each in silicon.[1] Hence we report area in terms of *equivalent LEs*.

To measure performance, we report the wall-clock-time for execution of a collection of benchmark applications, since reporting clock frequency or instructions-per-cycle (IPC) alone can be misleading. To be precise, we multiply the clock period (determined by the Quartus timing analyzer after routing) with the cycles-per-instruction (CPI) for each benchmark to attain the wall-clock-time execution for each benchmark. When comparing across benchmarks we normalize the benchmark lengths and measure instruction throughput in Millions of Instructions Per Second (MIPS).

To measure energy, we use Quartus' Power Play tool which produces a power measurement based on the switching activities of post-placed-and-routed nodes—determined by simulating benchmark applications on a post-placed-and-routed netlist of a processor in the Modelsim industrial RTL simulator. We subtract out static power, and we also subtract the power of the I/O pins: I/O pin power dominates, and is more dependent on how the processor interfaces to off-chip resources than its microarchitecture. For each benchmark we compare the energy-per-instruction, which is independent of the length of each benchmark.

3.3 Comparing with Altera Nios II Variations

To ensure that our generated designs are indeed interesting and do not suffer from prohibitive overheads, we have selected Altera's Nios II family of processors for comparison. Nios II has three mostly-unparameterized variations: `Nios II/e`, a small unpipelined 6-CPI processor with serial shifter and software multiplication; `Nios II/s`, a 5-stage pipeline with multiplier-based barrel shifter, hardware multiplication, and instruction cache; and `Nios II/f`, a large 6-stage pipeline with dynamic branch prediction, instruction and data caches, and optional hardware divider.

We have taken several measures to ensure that comparison against the Nios II variations is as fair as possible. We have generated each of the Nios processors with memory systems identical to those of our designs: two 64KB blocks of RAM are used for separate instruction and data memories. We do not include caches in our measurements, though some logic required to support the caches will inevitably count towards the Nios II areas. The Nios II instruction set is very similar to the MIPS-I ISA with some minor modifications in favor of Nios (for example, the Nios ISA has no tricky branch delay slots)—hence Nios II and our generated processors are very similar in terms of ISA. Nios II supports exceptions and OS instructions, which are so far ignored by SPREE. Finally, like Nios II, we also use `gcc` as our compiler, though we did not modify any machine specific parameters nor alter the instruction scheduling. Despite these differences, we believe that comparisons between Nios II and our generated pro-

[1] The relative area of these blocks was provided by Altera [8].

Table 1: Benchmark applications evaluated.

Source	Benchmark	Modified	Dyn. Instr. Counts
MiBench [10]	BITCNTS	di	26,175
	CRC32	d	109,414
	QSORT*	d	42,754
	SHA	d	34,394
	STRINGSEARCH	d	88,937
	FFT*	di	242,339
	DIJKSTRA*	d	214,408
	PATRICIA	di	84,028
XiRisc [6]	BUBBLE_SORT	-	1,824
	CRC	-	14,353
	DES	-	1,516
	FFT*	-	1,901
	FIR*	-	822
	QUANT*	-	2,342
	IQUANT*	-	1,896
	TURBO	-	195,914
	VLC	-	17,860
Freescale [2]	DHRY*	i	47,564
RATES [5]	GOL	di	129,750
	DCT*	di	269,953

* Contains multiply
d Reduced data input set
i Reduced number of iterations

cessors are relatively fair, and that we can be confident that our architectural conclusions are sound.

3.4 Benchmark Applications

We measure the performance of our soft processors using 20 embedded benchmark applications from four sources (as summarized in Table 1). Some applications operate solely on integers, and others on floating point values (although for now we use only software floating point emulation); some are compute intensive, while others are control intensive. Table 1 also indicates any changes we have made to the application to support measurement, including reducing the size of the input data set to fit in on-chip memory (`d`), and decreasing the number of iterations executed in the main loop to reduce simulation times (`i`). Additionally, all file and other I/O were removed since we do not yet support an operating system.

4. THE IMPACT OF CUSTOMIZING SOFT PROCESSORS

In this section we use the SPREE system to measure the impact of customizing soft processors to meet the needs of individual applications. We first validate our generated soft processors by comparing them to the Nios II variations, then we investigate the opportunities for microarchitectural customization afforded by each. Next we demonstrate the impact of tuning the microarchitecture for a given application, for example by selecting between hardware and software support for multiplication, the type of shifter implementation, the number of pipeline stages, and options for multi-cycle paths in a pipeline. We also explore the impact of ISA subsetting which eliminates hardware support for architectural features not used by the application, and how ISA subsetting and microarchitectural tuning can both be combined.

4.1 Comparison with Nios II Variations

Figure 2 illustrates our comparison of SPREE generated processors to the commercial Altera Nios II variations in the performance-area space. SPREE's generated processors span the design space between Nios II variations, while allowing more fine-grained microarchitectural customization. Processors between 600 and 1600 LEs can be generated with up to 9x difference in performance. The figure also shows that SPREE processors remain competitive with the commercial Nios II. In fact, one of our generated processors is both smaller and faster than the `Nios II/s`—hence we examine that processor in greater detail.

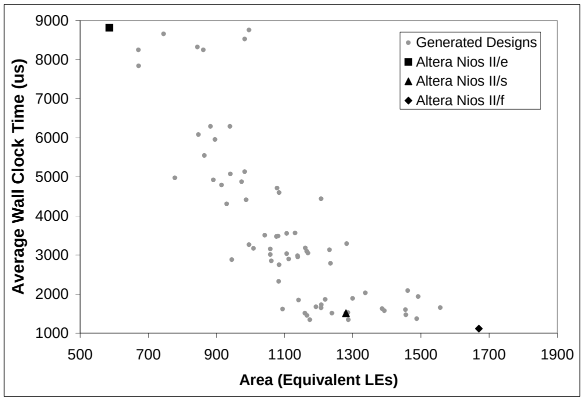

Figure 2: Comparison of our generated designs vs the three Altera Nios II variations.

The processor of interest is an 80MHz 3-stage pipelined processor, which is 9% smaller and 11% faster in wall-clock-time than the `Nios II/s`, suggesting that the extra area used to deepen `Nios II/s`'s pipeline succeeded in increasing the frequency, but brought overall wall-clock-time performance down. The generated processor has full inter-stage forwarding support and hence no data hazards, and suffers no branching penalty because of the branch delay slot instruction in MIPS. The CPI of this processor is 1.36 whereas the CPIs of `Nios II/s` and `Nios II/f` are 2.36 and 1.97 respectively. However, this large gap in CPI is countered by a large gap in clock frequency: `Nios II/s` and `Nios II/f` achieve clock speeds of 120 MHz and 135 MHz respectively, while the generated processor has a clock of only 80MHz. These results demonstrate the importance of evaluating wall-clock-time over clock frequency or CPI alone, and that faster frequency is not always better.

4.2 Tuning Soft Processor Microarchitecture

In this paper we explore the application-specific tuning of four microarchitectural axes: (i) optional hardware multiplication support; (ii) choice of shifter implementation; (iii) pipeline depth; and (iv) cycle latency of multi-cycle paths. Each of these axes provides an application-specific tradeoff in area, performance, and energy that we exploit. We explore the trade-offs for each axis independently, and then we explore the benefits of tuning all axes to the needs of individual applications at once. Finally, we evaluate the best application-tuned processor for each benchmark application versus the best overall general-purpose processor.

4.2.1 Hardware vs Software Multiplication

Hardware support for multiplication instructions can provide enormous performance improvements for multiplication-intensive applications—however, this extra hardware significantly increases the area and energy consumption of the pro-

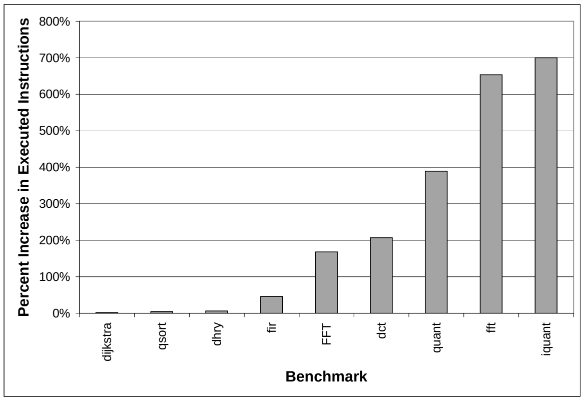

Figure 3: The increase in total instructions executed when changing from hardware to software multiplication support.

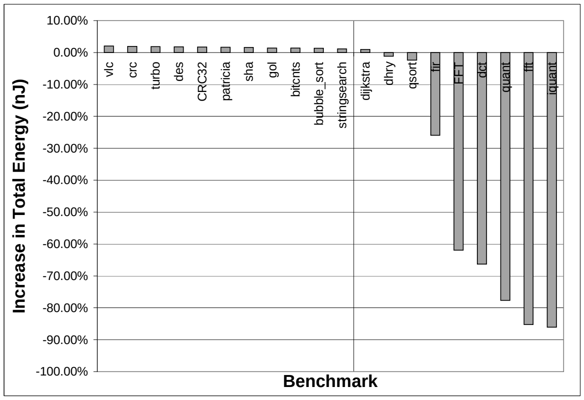

Figure 4: Increase in total energy due to support for hardware multiplication. Benchmarks to the left of the vertical divider are those without multiply instructions (note there is still energy overhead).

cessor. This trade-off is therefore very application specific, hence we evaluate processors both with and without hardware multiplication support; those without emulate multiplication in software using the `C` subroutine provided by `gcc`, at the cost of each multiplication requiring a much larger number of instructions to complete. Previous experiments [19] (across many processors) demonstrated that supporting multiplication in hardware costs approximately 220 LEs, which is significant compared to the size of processors typically generated using SPREE. If the multiplier is also used to perform shifting (as described in Metzgen [13]) then the multiplier effectively costs only 100 LEs. These costs are relatively consistent across architectures, and also have no dependence on the application.

The performance benefit of hardware support for multiplication depends heavily on the frequency of multiply instructions in the dynamic execution of a given application. As we see in Figure 3 for the applications that contain multiplication instructions, the impact on the total number of instructions executed when changing from hardware to software multiplication support varies widely across applications: for example, IQUANT executes many multiplies while DIJKSTRA executes very few.

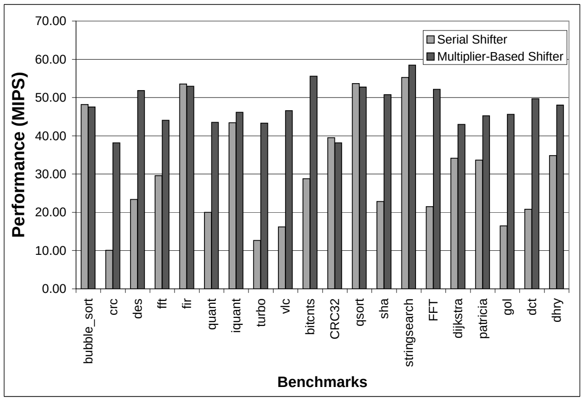

Figure 5: Performance of a processor with different shifter implementations.

Figure 4 shows the increase in energy consumed for processors when changing from software to hardware multiplication support. Note that for applications which do not use multiplication, energy consumption is increased by nearly 2% because the multiplication logic can toggle even when the multiplier is not being used.[2] Applications that execute many multiplications save immensely on energy, in some cases 80% since the single hardware multiply instruction is much more energy-efficient than executing the potentially long subroutine.

4.2.2 Tuning the Shifter

The choice of shifter implementation provides another potential avenue for application-tuning. In this work, we consider three shifter implementations: (i) a serial shift register; (ii) a multiplier-based barrel shifter using the hard multiplier, as proposed by Metzgen [13], and (iii) a LUT-based barrel shifter, implemented as tree of multiplexers. We explored the trade-offs between these shifter implementations previously [19]. We found that compared to the serial shifter, the multiplier-based shifter is 65 LEs larger and the LUT-based shifter is 250 LEs larger. We also found that on average the LUT-based shifter provides a negligible performance benefit over the multiplier-based shifter—hence we do not further consider the LUT-based shifter here.

Figure 5 shows the performance of a 3-stage pipelined processor with either a serial shifter or multiplier-based shifter. For many benchmarks, the multiplier-based shifter provides a significant performance advantage, while for others the benefit is only slight. Similar to multiplication, the performance impact of more aggressive hardware support for shifting depends on the frequency of shift operations executed by the application. However, for shifters the resulting performance is additionally dependent on the actual shift amounts in the application: when shifting by one, a serial shifter and barrel shifter have similar performance; as the shift amount increases, so does the performance advantage of the barrel shifter. When the barrel shifter does not benefit the performance of a given application, the serial shifter is the preferred choice since it consumes less area.

There are also application-specific tradeoffs related to the energy consumption of different shifter implementations. When

[2] In the future, we will consider adding support to SPREE that guards against such unnecessary toggling.

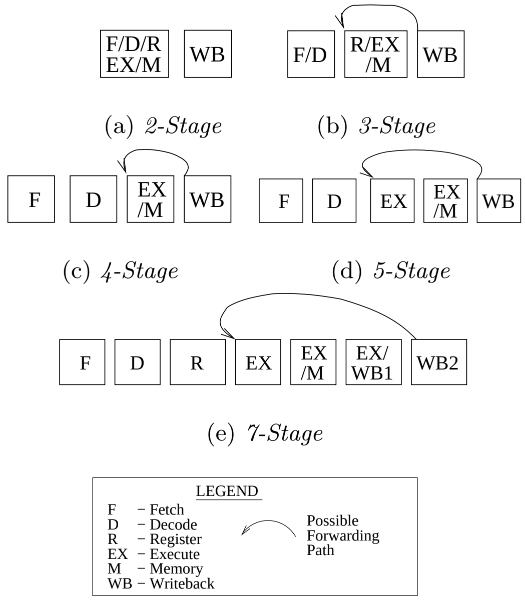

Figure 6: Processor pipeline organizations studied. Arrows indicate possible for warding lines.

an application does not use shift operations, the energy consumption of a processor that uses a serial shifter is slightly better than one with a multiplier-based shifter. For example, BUBBLE_SORT and FIR, which do not use shift operations, experience an energy savings of over 7% when a serial shifter is used. However, for a benchmark such as CRC that uses many shifts, the serial shifter can increase energy consumption 160% compared to the multiplier-based shifter.

4.2.3 Tuning Pipeline Depth

The number of pipeline stages can greatly influence the size and complexity of any processor. We have used SPREE to generate pipelines between 2 and 7-stages deep[3]. In previous work [19] we observed that deepening the pipeline would increase area substantially but irregularly: some stages could be added cheaply, while others were more expensive. Using as a baseline a processor with both a hardware multiplier and multiplier-based shifter, in this work we explore pipeline depth by implementing the pipelines shown in Figure 6. In all pipelines, data hazards are resolved through interlocking, and branches are statically predicted to be *not taken*.

Figure 7 shows the performance impact of varying pipeline depth for four applications which are representative of several trends that we observed. The figure shows that the 2-stage pipeline performs poorly compared to the rest: the synchronous RAMs in Stratix must be read in a single stage of the pipeline for this design, hence it suffers a stall cycle. The 7-stage pipeline also has a disadvantage: branch delay slot instructions are much more difficult to support in such a deep pipeline, increasing the complexity of the control logic for this design. In contrast, the trends for the 3, 4, and 5-stage pipelines vary widely by application. DES experiences up to 17% improved performance as the pipeline depth increases from 3 to 5 stages, while for STRINGSEARCH performance degrades by 18%. SHA maintains consistent performance across the pipelines, which is a typical trend for many applications. For DHRY, performance decreases by only 2% and then increases by 11%. Pipeline depth is therefore another application-specific tradeoff, due to the fact that some applications suffer more than others from branch penalties, and data hazards of varying distances.

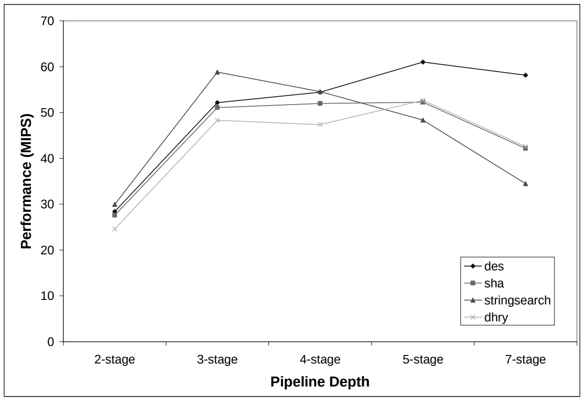

Figure 7: Performance impact of varying pipeline depth for select benchmarks.

Although not shown, we found that the variance in trends between applications for different forwarding paths is insignificant. We found that they can provide area/performance/energy tradeoffs in general, but none that differ significantly on a per-application basis—e.g., forwarding is a "win" for all applications, even when considering performance-per-area. We also found that energy-per-instruction consistently decreases as pipeline depth increases, and this also has no application-specific dependence. We attribute this decrease to the impact of pipeline registers on glitching: the propagation of a glitch stops at a pipeline register, hence the additional pipeline registers in deeper pipelines result in decreased propagation of glitches.

4.2.4 Tuning with Unpipelined Multi-cycle Paths

Adding pipeline registers increases frequency but can also increase total CPI, as data hazards and branch penalties result in additional pipeline stalls. Alternatively, registers can be used in a more direct way for trading clock frequency and CPI: registers can be inserted *within* a bottleneck pipeline stage that occasionally prevent that stage from completing in a single cycle, but that also allow the stage (and hence the entire pipeline) to run at a higher clock frequency. As an example, consider the execution stage if the multiplication unit were the bottleneck: instead of using a pipelined multiplication unit, we could use an unpipelined 2-cycle multiplication unit. In this case, a multiplication instruction will use the execution stage for two cycles (stalling the pipeline for one extra cycle), while non-multiply instructions will use the execution stage for only one cycle. In this way, we can potentially improve performance by trading CPI for maximum operating clock frequency.

As a concrete example, we consider the 5-stage pipeline with 2-cycle multiplier-based barrel shifter. This processor has critical path through the shifter which limits the clock speed to 82.0 MHz while achieving 1.80 average CPI across

[3] The 1-stage pipeline is omitted as it yields no benefit over the 2-stage, and the 6-stage is omitted as we were unable to achieve adequate clock frequency improvement over the 5-stage.

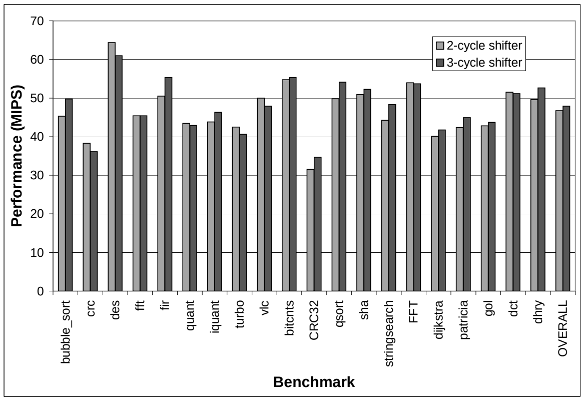

Figure 8: The performance tradeoff in implementing unpipelined multi-cycle paths on a processor across the benchmark set.

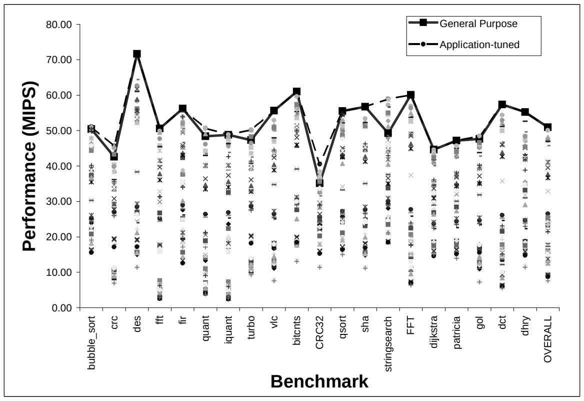

Figure 9: Performance of all processors on each benchmark—in bold is the best-on-average, (*general-purpose*) processor, the dashed line is the fastest for each benchmark (*application-tuned*).

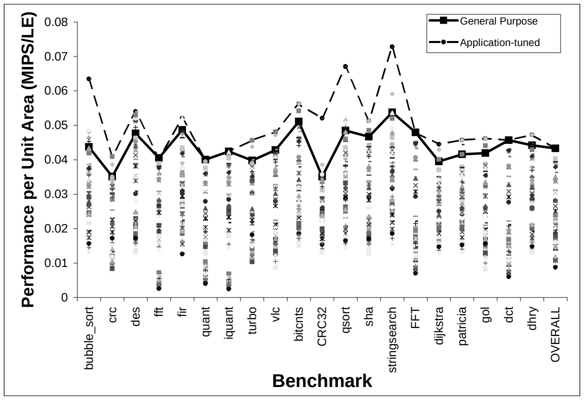

Figure 10: Performance-per-area of all processors on each benchmark—in bold is the best-on-average (general-purpose) processor over all benchmarks, the dashed is the best per benchmark (application-tuned).

the benchmark set. We can create another unpipelined multi-cycle path by making the multiplier-based shifter a 3-cycle unpipelined execution unit which results in a clock frequency of 90.2 MHz and 1.92 average CPI. The 10% clock frequency improvement is countered by an average CPI increase of 6.7%. Figure 8 shows the instruction throughput in MIPS of both processors for each benchmark and indicates that benchmarks can favor either implementation. For example, BUBBLE_SORT achieves 10% increased performance when using the 3-cycle multiplier-based shifter while CRC achieves 6% increased performance with the 2-cycle implementation. Hence we can use unpipelined multi-cycle paths to make application-specific tradeoffs between clock frequency and CPI. Note that this technique is not limited to the execution stage, and can be applied anywhere in the processor pipeline. In the set of explored processors this technique was explored in large execution units (either the shifter or multiplier) whenever these units lay in the critical path.

4.3 Application-Tuned vs General Purpose

We have demonstrated that several microarchitectural axes provide application-specific tradeoffs that can be tuned in soft processors to better meet application requirements. In this section we use SPREE to implement all combinations of these architectural axes and exhaustively search for the best processor for each application in our benchmark set. We call this processor the *application-tuned* processor. We also determine the processor performed best on average over the complete benchmark set—this we refer to as the *general-purpose* processor. We then analyze the difference between the general purpose processor and the application-specific processor and evaluate the potential for making application-specific tradeoffs in soft processor microarchitecture.

Figure 9 shows the performance of all processors on each benchmark measured in millions of instruction per second (MIPS). The processors include all combinations of pipeline stages, shifter implementations, and multiplication support as well as different multi-cycle unpipelined paths. The bold line indicates the performance on each benchmark of the best-on-average (general-purpose) processor—this processor (the 5-stage pipeline with LUT-based barrel shifting) provides the best arithmetic mean performance across the entire benchmark set, as represented by the OVERALL column. The dashed line connects the processors that are fastest for each individual benchmark. We observe that the best-on-average processor is also often the fastest processor for each benchmark, since the dashed line and bold line are often co-linear. Some exceptions are STRINGSEARCH, CRC32, and TURBO, for which certain processors perform 16.4%, 13.3%, and 5.7% faster respectively. Overall there is little room for application specific consideration with respect to maximizing performance alone since the architectural axes we consider generally trade area for speed. For example, the benefit of using a serial shifter or software multiply is in reducing area at the expense of performance; therefore, if only performance is considered, there is no motivation for using either of these two options.

We are therefore motivated to consider both area and speed at the same time by measuring performance-per-area, specifically by measuring MIPS/LE for each benchmark/processor combination as shown in Figure 10. The bold line now indicates the performance-per-area of the best-on-average processor—this processor (the 3-stage pipeline with multiplier-

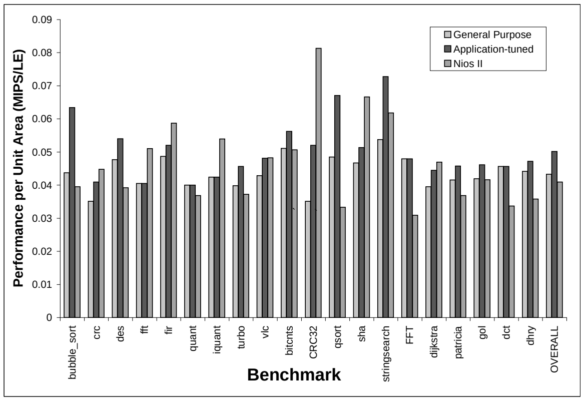

Figure 11: Performance-per-area across the benchmark set for the general-purpose, application-tuned, and Nios II processor.

based shifter) provides the best arithmetic mean performance across the entire benchmark set, as represented by the `OVERALL` column. In this case, only 6 of the 20 benchmarks achieve their highest performance-per-area using the best overall processor; instead, the best processor for each benchmark varies. By choosing an application-tuned processor, each benchmark improves performance-per-area by 11.4% over the best overall processor on average across the entire benchmark set; furthermore, STRINGSEARCH, QSORT, CRC32, and BUBBLE_SORT improve performance-per-area by approximately 30%. In future work, we expect this number to grow significantly when supporting more advanced architectural axes such as datapath widths, branch predictors, aggressive forwarding, caches, and VLIW datapaths.

We now compare our results with the `Nios II` processor variations. Figure 11 plots the performance-per-area across the benchmark set for: (i) the best general-purpose (best-on-average) processor; (ii) the best application-tuned tuned processor; (iii) the best Nios II processor variation (`s`, `e`, or `f`). For 9 of the 20 benchmarks, the application-tuned processor yields significantly better performance-per-area than any of either the general-purpose processor or the commercial Nios II variants. The success of the Nios II on the CRC32 benchmark is due to the exceptional wall-clock-time performance of the `Nios II/s`. On average across all benchmarks, the application-tuned processors generated by SPREE are the most area-efficient, yielding 22% better results than Nios II because of the larger customization space afforded by SPREE.

4.4 ISA Subsetting

So far we have investigated microarchitectural customizations that favor an individual application but still fully support the original ISA. In this section, we propose to capitalize on situations where: (i) only one application will run on the soft processor; (ii) there exists a reconfigurable environment allowing the hardware to be rapidly reconfigured to support different applications or different phases of an application. We customize the soft processor by having it support only the fraction of the ISA which is actually used by the application. SPREE performs this *ISA subsetting* by parsing the application binary to decide the subsetted ISA, removing unused connections and components from the input datapath, and then generating simpler control. Figure 12 shows the fraction of the 50 MIPS-I instructions supported by SPREE that are used by each benchmark, which is rarely more than 50%. BUBBLE_SORT, FIR, and CRC32 use only about one quarter of the ISA. With such sparse use of the ISA, we are motivated to investigate the effect of eliminating the architectural support for unused instructions.

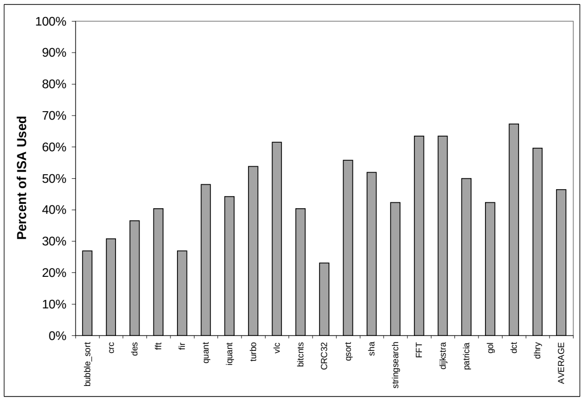

Figure 12: ISA usage across benchmark set.

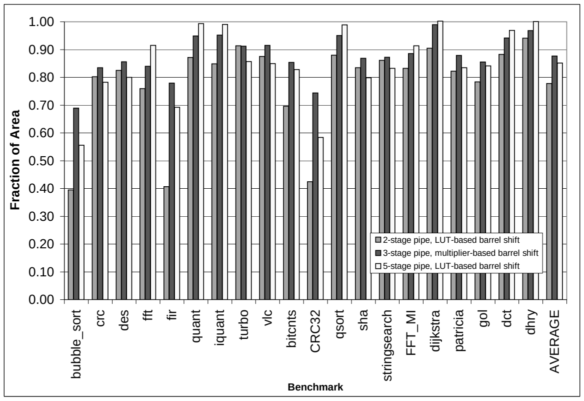

Figure 13: The impact on area of ISA subsetting on three architectures.

To evaluate the impact of ISA subsetting, for each of the 20 benchmarks we subsetted three processor architectures: (i) A 2-stage pipeline with LUT-based barrel shifting; (ii) The 3-stage pipeline with multiplier-based barrel shifting; (iii) a 5-stage pipeline with LUT-based barrel shifting. All three processors utilize hardware multiplication support. Since the cycle-by-cycle execution of each benchmark is unaffected by this experiment, we use clock frequency to measure performance gain. The relative area of each subsetted processor with respect to its non-subsetted version is shown in Figure 13. It is evident that the three benchmarks which use only 25% of the ISA (BUBBLE_SORT, FIR, and CRC32) obtain the most significant area savings. For the 2-stage architecture, these 3 benchmarks obtain a 60% area savings, while most other benchmarks save 10-25% area. Closer inspection of these three benchmarks reveal that they are the only benchmarks which do not contain shift operations—shifters are large functional units in FPGAs, and their removal leads to a large area savings.[4] The savings is more

[4] In the MIPS ISA, there is no explicit `nop` instruction, hence

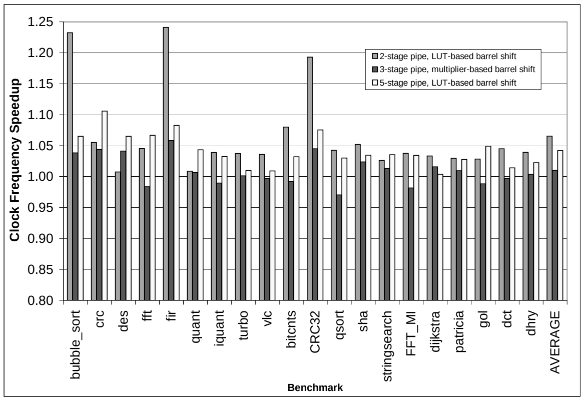

Figure 14: The impact on clock speed of ISA subsetting on three architectures.

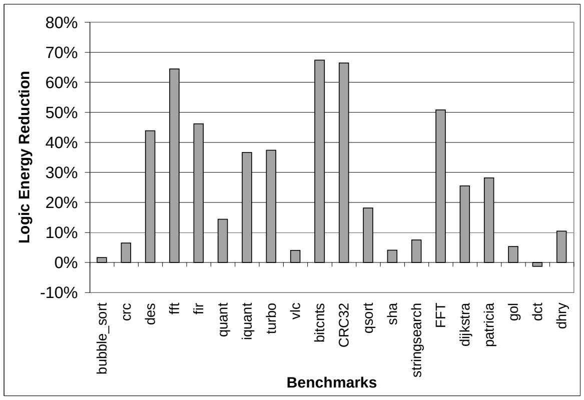

Figure 15: Energy consumption of subsetting on 3-stage pipeline.

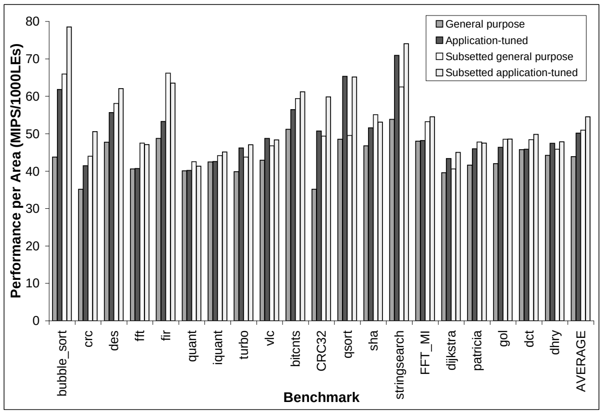

Figure 16: Performance-per-area of tuning, subsetting, and their combination.

pronounced in the 2 and 5-stage pipeline where the shifter is LUT-based and hence larger (as seen in Section 4.2.2).

Figure 14 shows the clock frequency improvement for the subsetted architectures. In general we see modest speedups of 7% and 4% on average for the 2 and 5-stage pipelines, respectively. The 3-stage pipeline is not improved at all, as its critical path is in the data hazard detection logic and hence cannot be removed. More positively, the modest clock frequency speedups indicate that our pipelines have well-balanced logic delays: when logic is removed from a given path there is often another path to maintain the previous critical path length, hence the odds of a given subsetting reducing all paths is relatively small. However, there is notable performance improvement in the 2-stage pipeline for the three benchmarks without shifts, BUBBLE_SORT, FIR, and CRC32. This is because the LUT-based shifter lay in the critical path of the pipeline and caused poor balancing of logic delay. Removing the shifter allows for a roughly 20% improvement in clock frequency for these benchmarks.

Figure 15 shows the reduction in energy resulting from ISA subsetting. Note that energy dissipated by the memory is ignored since it accounts for 80-90% of the energy, and it can never be removed by subsetting since all applications fetch and write to memory. The figure demonstrates that the removal of high-toggle rate components and simplified control result in significant energy savings in the processor pipeline. The subsetted processors of some benchmarks such as FFT, BITCNTS, and CRC32 provide greater than 65% energy savings. On average across all the subsetted processors, approximately 27% of the non-memory energy can be saved. A miniscule increase in energy was seen for the DCT benchmark which we attribute to noise in the CAD system, total system energy decreased very slightly but the fraction attributed to logic increased unexpectedly.

4.5 Combining Customization Techniques

We have presented two methods for creating application-specific soft processors: (i) architectural tuning, which alters soft processor architecture to favor a specific benchmark; and (ii) ISA subsetting, which removes architectural support that is not utilized by the benchmark. In this section we compare the effectiveness of the two techniques in terms of performance-per-area both individually and combined. We define the best general-purpose processor as the single processor which achieves the greatest performance-per-area on average across all benchmarks, and the best application-tuned processors as the set of processors which achieve the best performance-per-area for each benchmark. For each processor and benchmark we then perform ISA subsetting, and measure the performance-per-area of the four combinations: general-purpose, application-tuned, subsetted general-purpose, and subsetted application-tuned.

Figure 16 shows the performance-per-area for all four combinations. As shown previously, the application-tuned processor is consistently better than the general-purpose processor. ISA subsetting is more effective on the general-purpose processor than on the application-tuned processors: the performance-per-area is improved by 16.2% on average for the general-purpose processor while by only 8.6% for the application-tuned processor. This is intuitive since since the hardware which was eliminated during subsetting was likely reduced in size during the tuning of the application-tuned processor. For example, FIR uses no shifting, therefore a small serial shifter is chosen during tuning and later

nops are encoded as a shift-left-by-zero. Therefore to remove the shifter, one must include special hardware to handle these nop instructions, or else re-encode the nop. In this work nops are re-encoded as add zero (similar to Nios II) to allow for complete removal of the shifter, since all benchmarks use adds.

removed in subsetting, resulting in a less dramatic area reduction. There is a large variation when deciding between these two methods: some benchmarks such as FIR achieve up to 25% increased performance-per-area by using the application-tuned processor over the subsetted general-purpose processor, while others such as QSORT achieve a 25% increase by using the subsetted general-purpose processor over the application-tuned processor (i.e., they are opposite). These two methods are very competitive as summarized in Figure 16 by the **AVERAGE** bars, which show the subsetted general-purpose processor having slightly higher performance-per-area than the application-tuned (by only 2.2%).

The subsetted application-tuned processor combines all customizations (both the microarchitectural tuning and the ISA subsetting) and therefore often achieves the best performance-per-area. The combination of the two techniques is complementary: on average, subsetted application-tuned processors achieve more than 10% better performance-per-area across the benchmark set than either microarchitectural tuning or ISA subsetting alone. However, for each benchmark, either technique can come to within 4% of the combined approach. Overall, the combined approach can improve performance-per-area by up to 80% for BUBBLE_SORT, and by 24.5% on average across the benchmark set.

5. CONCLUSIONS

The reconfigurability of soft processors can be exploited to meet design constraints by making application-specific tradeoffs in their microarchitecture. In this paper we used the SPREE infrastructure to generate and evaluate customized RTL implementations of soft processors, investigating the individual tradeoffs in customizing the hardware support for multiplication, the shifter implementation, the pipeline depth, and the number of cycles in unpipelined multi-cycle paths. We applied all of these customizations in all combinations and determined that the best of these application-specific processors offers considerable advantages over the best-on-average general purpose processor: an improvement in performance-per-area of 11.4% on average across all benchmarks.

We also used the SPREE infrastructure to perform ISA subsetting, where the hardware support for unused parts of the ISA are removed for each application. We obtained large reductions in the area and power of the processors with this technique—reductions of approximately 25% for both metrics on average and up to 60% for some benchmarks. Combining our techniques for microarchitectural tuning with ISA subsetting results in an even more dramatic benefit, where performance-per-area is improved by 24.5% on average across all benchmarks.

In the future we will explore a more broad set of customizations including branch prediction, caches, datapath width, VLIW datapath parallelism, and other more advanced architectural features. We also plan to investigate more aggressive customization of these processors, including changing the ISA to encourage better customization. Finally, we are interested in exploring the benefits of tuning the compiler based on a more exact knowledge of the target architecture.

6. REFERENCES

[1] ARCtangent. http://www.arc.com.

[2] Dhrystone 2.1. http://www.freescale.com.

[3] MicroBlaze. http://www.xilinx.com/microblaze.

[4] Nios II. http://www.altera.com/products/ip/processors/nios2.

[5] RATES - A Reconfigurable Architecture TEsting Suite. http://www.eecg.utoronto.ca/∼lesley/benchmarks/rates/.

[6] XiRisc. http://www.micro.deis.unibo.it/∼campi/XiRisc/.

[7] Xtensa. http://www.tensilica.com.

[8] R. Cliff. Altera Corporation. Private Comm, 2005.

[9] R. Dimond, O. Mencer, and W. Luk. CUSTARD - A Customisable Threaded FPGA Soft Processor and Tools . In *International Conference on Field Programmable Logic (FPL)*, August 2005.

[10] M. Guthaus and et al. MiBench: A free, comercially representative embedded benchmark suite. In *In Proc. IEEE 4th Annual Workshop on Workload Characterisation*, December 2001.

[11] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai. PEAS-III: An ASIP Design Environment, September 2000.

[12] D. M. Lewis and et al. The Stratix^{TM} routing and logic architecture. In *International symposium on Field-programmable gate arrays*, pages 12–20, 2003.

[13] P. Metzgen. Optimizing a High-Performance 32-bit Processor for Programmable Logic. In *International Symposium on System-on-Chip*, 2004.

[14] P. Mishra, A. Kejariwal, and N. Dutt. Synthesis-driven Exploration of Pipelined Embedded Processors. In *VLSI Design*, pages 921–926, 2004.

[15] K. Morris. Embedded Dilemma. http://www.fpgajournal.com/articles/embedded.htm, November 2003.

[16] J. Turley. Survey: Who uses custom chips. *Embedded Systems Programming*, August 2005.

[17] P. Yiannacouras. SPREE. http://www.eecg.utoronto.ca/∼yiannac/SPREE/.

[18] P. Yiannacouras. The Microarchitecture of FPGA-Based Soft Processors. Master's thesis, University of Toronto, 2005.

[19] P. Yiannacouras, J. Rose, and J. G. Steffan. The Microarchitecture of FPGA Based Soft Processors. In *CASES'05: International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 202–212. ACM Press, 2005.