Custom Code Generation for Soft Processors

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

***Abstract*— Embedded systems designers that use FPGAs are increasingly including *soft processors* in their designs (configurable processors built in the programmable logic of the FPGA). While there has been a significant amount of research on adding custom instructions and accelerators to soft processors, these are typically used to extend an unmodified base ISA targeted by generic compilation such as with unmodified `gcc`. In this paper we explore several opportunities for the compiler to optimize the code generated for soft processors through application-specific customization of the base ISA—techniques that are orthogonal to adding custom instructions. In particular we explore: (i) low level software-hardware trade-offs between basic instructions; (ii) the utility of ISA-specific features—in particular for the delay slots and `Hi`/`Lo` registers in the MIPS ISA; and (iii) application specific register management. We find that through these techniques that have no hardware cost we can improve the area efficiency of soft processors by 12% on average across a suite of benchmarks, and by up to 47% in the best case.**

I. Introduction

As embedded systems designers increasingly employ FPGAs, their designs are likely to contain one or more *soft processors*—processors that are implemented in the programmable logic of the FPGA [1], [2]. Soft processors are useful because they can easily be programmed (rather than writing HDL), and a designer can instantiate the exact number of processors required and can have them incorporated into the greater design to ease placement and routing. A key advantage of soft processors is that they can be customized to match the target application or applications. For example, a great deal of recent research has focused on the ability to add *custom instructions* to soft processors, where frequently executed code segments are replaced with encapsulated hardware implementations that can be "called" by the soft processor to improve performance [3]–[7]. However, for many designs, rather than improving the performance of a soft processor at all costs, the designer desires a soft processor that is "fast enough" for the target application, and would rather save area for other uses—perhaps to help fit the overall design into a given FPGA component. Recent research explores architectural area/performance/power trade-offs and customization opportunities for a wide range of soft processor designs [8], [9]. However, this work assumes a fixed ISA (MIPS I), and the evaluation is based on default `gcc` compilation—missing many important opportunities for further customization.

A. Generating Custom Code for Custom Processors

In this paper we investigate several opportunities for the compiler to customize the code that is generated for soft processors—to understand the range of impact of such techniques, and to give designers more fine-grain control of the area/performance trade-off space for soft processors. Using the SPREE infrastructure (Soft Processor Rapid Exploration Environment) [8], we study the impact of our techniques on the wall-clock time and area of a wide range of soft processor architectures running a set of general-purpose benchmark applications. In particular we focus on three main areas of customization: (i) low-level software/hardware trade-offs, for example in shifter implementations and in hazard detection and observation; (ii) inclusion of ISA-specific features, for example the MIPS load and branch delay slots, and `Hi`/`Lo` multiplication result registers; and (iii) register management, for example operand scheduling to minimize forwarding logic, and reducing the number of architected registers. We also study the combination of these techniques and their resulting impact on area and performance. It is important to understand that in this paper we do not study the addition of custom accelerators in the form of custom instructions and co-processors, although these are complementary to the compiler techniques that we propose. Finally, the initial work presented here suggests future efforts into larger-scale compiler optimizations for soft-processors and other customizable architectures.

B. Related Work

Some of the trade-offs we examine in this paper have been explored previously in other contexts. Shrivastava *et. al.* demonstrated that instruction scheduling can exploit incomplete bypassing in embedded processors [10]. The CUSTARD [5] customizable soft processor has the ability to customize forwarding lines, and provides a variable size register file and optional branch and load delay slots—although to our knowledge these have not been specifically evaluated.

Design decisions similar to some of those we discuss in this paper were made for commercial soft processors, although there is no published evaluation that quantifies their value. For example, the commercial NIOS II and Microblaze processors implement three-operand multiplication (rather than having special multiplication registers such as the MIPS `Hi`/`Lo` registers), and the NIOS II has no delay slots while the Microblaze supports variants of branches with and without delay slots. Support for unaligned memory operations has recently been added to `gcc`, but the corresponding hardware implementation of those operations is patented by MIPS [11].

Table 1. Benchmark applications evaluated.

Source	Benchmark	Modified	Dyn. Instr. Counts
MiBench [12]	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 [13]	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 [14]	DHRY	i	47,564
RATES [15]	GOL	di	129,750
	DCT*	di	269,953

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

C. Contributions

This paper makes the following three main contributions: (i) proposal and evaluation of several techniques for custom code generation for soft processors, including software-only and custom shifters, software hazard observation, and operand scheduling; (ii) evaluation of the area/performance trade-offs for several MIPS-specific ISA features, including `Hi`/`Lo` registers, load and branch delay slots; (iii) composition of those techniques to improve on the state of the art of generating application-specific soft processors.

II. Infrastructure for Varying Soft Processor Compilation, ISAs, and Architectures

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-bits MIPS I [16] ISA; integer division is implemented in software, and for now interrupts are not supported. Using the 20 embedded benchmark applications described in Table 1, we evaluate our compiler techniques for generating custom code for varying soft processor architectures.

We use the SPREE system [8] to generate a wide range of soft processor architectures (full details are available in a previous publication [17]). SPREE takes as input ISA and datapath descriptions and produces RTL which is synthesized, mapped, placed, and routed by Quartus 5.0 [18] using the default optimization settings. The generated processors target the Altera Stratix FPGAs, in particular the `EP1S40F780C5` device—a mid-sized device in the family with the fastest speed grade. We determine the area and clock frequency of each soft processor design using the arithmetic mean across 10 seeds (which produce different initial placements before placement and routing) to improve our approximation of the true mean. For each benchmark, the soft processor RTL design is simulated using Modelsim 6.0b [19] (i) to 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]).

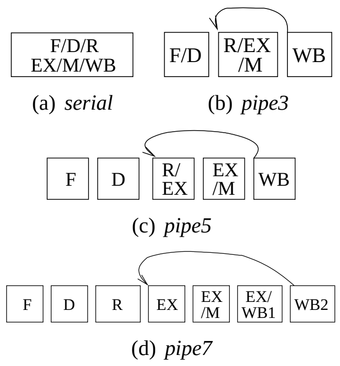


Fig. 1. Processor pipeline organizations studied. The pipeline stages are: `F` for fetch, `D` for Decode, `R` for register, `EX` for execute, `M` for memory, and `WB` for write-back. The arrow indicates a path for forwarding two operands at once.

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 these processors in *equivalent LEs*, a number that additionally accounts for the consumed silicon area of any hardware blocks (e.g., memory and multiplication units). For the processor clock rate, we report the maximum frequency supported by the critical path of the processor design. To combine area, frequency, and cycle count to evaluate an optimization, we use a metric of *area efficiency*, in million instructions per second (MIPS) per thousand equivalent LEs. Finally, we obtain dynamic power metrics of our benchmarks with Quartus' Power Play tool and report the numbers without the I/O pins power in nano-Joules per instruction (nJ/instr).

As shown in Figure 1, the processors that we evaluate are unpipelined (`serial`), 3-stage-pipelined (`pipe3`), 5-stage-pipelined (`pipe5`), and 7-stage-pipelined (`pipe7`). The unpipelined processor is the smallest (889 LEs, 67.7 MHz): it has a multiplier and a serial shifter. The pipelined processors all have forwarding lines for both operands by default. The 3-stage pipeline has a shifter that is implemented with the multiplier, and is the most area-efficient processor generated by SPREE [9] (1174 LEs, 78.3 MHz). The 5-stage pipeline also has a multiplier-based shifter, and implements a compromise between area efficiency and maximum operating frequency (1283 LEs, 86.79 MHz). The 7-stage pipeline has a barrel shifter, is the largest processor, and has the highest frequency (1557 LEs, 100.59 MHz).

III. Low-Level Software-Hardware Trade-Offs

A powerful trade-off for soft processor designs is the implementation of common routines in either software (through regular instructions in the base ISA) or custom hardware (implemented as custom instructions in addition to the base ISA). However, for area-sensitive applications we find it can be compelling to explore similar trade-offs in the actual base ISA

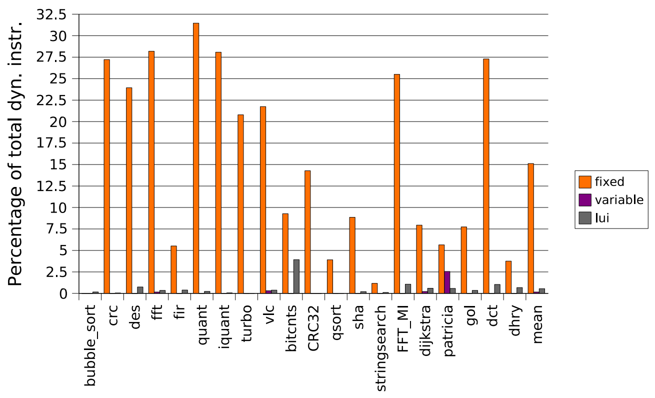


Fig. 2. Percentage of dynamic instructions that contain shift operations, broken down by those that have a fixed shift-amount encoded in the instruction (`sll`, `sra`, and `srl`), those that have a variable shift amount stored in a register (`sllv`, `srav`, and `srlv`), and the `lui` instruction which also has a fixed shift-amount (16 bits left).

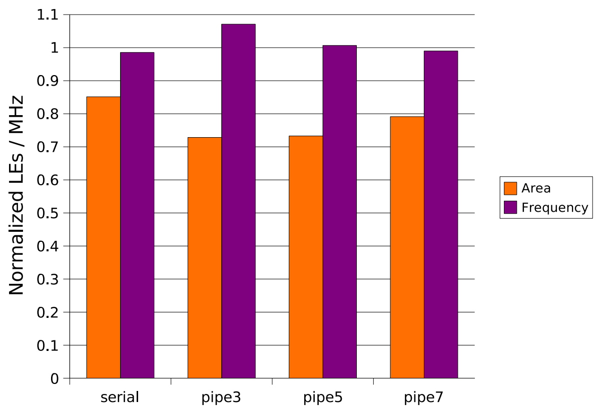


Fig. 3. Impact of removing the dedicated shifter unit, relative to the corresponding default processors with software multiplies.

and architecture. For example, previously we demonstrated that "subsetting" the base ISA—so that the hardware support for any instructions that are not used by an application is deleted from the processor—results in an average area reduction of 25% and up to 60% for some applications [9]. In this section we evaluate two opportunities to further subset the ISA and hardware by having the compiler compensate in software: (i) by removing the shift unit or replacing it with one or more much smaller fixed-amount shift units, and (ii) by removing the hazard detection logic and instead observing dependences by having the compiler schedule instructions and insert no-ops.

A. Shifter Implementations

It has been shown that it is advantageous to implement shift operations using a hard multiplier if one is available [8]. However, for an area-limited design that does not contain a hard multiplier (opting instead for software multiplication if needed), a dedicated shifter can consume more than 250 LEs. Instead we investigate the possibility of implementing various shift operations either partially or entirely in software. Shifts can be implemented entirely in software using non-shift operations such as `add` and `subtract`. Alternatively, we could implement a small number of fixed-amount shifts in hardware (in far less area than a full variable-amount shifter), and use those operations to build up other shift amounts through software (e.g., call a shift-right-by-four operation three times to implement a shift-right-by-twelve operation).

Figure 2 shows the percentage of dynamic instructions executed for each benchmark that perform a shift operation, for example *shift left logical* (`sll`), *shift right arithmetic* (`sra`), and *load upper immediate* (`lui`, which shifts left by 16 bits). Some instructions have a variable shift amount stored in a register (`srav`), as opposed to an immediate shift amount encoded in the instruction (`sra`). The results demonstrate that while shift instructions can be quite common (an average of 15% of dynamic instructions across all benchmarks), the vast majority shift by a fixed amount. In general, any variable shifts can potentially be implemented entirely in software, or else through use of a fixed-amount unit shifter—with the possible exception of the PATRICIA benchmark for which variable shifts are more common (2.5% of dynamic instructions).

To further demonstrate the potential for eliminating shift instructions, Figure 3 shows the impact of removing the dedicated shifter unit for various processors, each relative to the corresponding default processor with software multiplies. We observe that removing the shifter results in significant area savings for all processors. `pipe3` and `pipe5` benefit from the largest area savings because they implement shifts with a multiplier that can be eliminated when removing the support for the shift operations. While the shifter is on the critical path for `pipe3`, the clock frequency of the other processors is not significantly affected, even if it varies somewhat due to the impact on overall placement and routing. Given these potential savings, we are motivated to investigate ways to eliminate shift instructions from the base ISA, while minimizing the impact on overall performance.

In the absence of a dedicated shift unit, shift operations can be supported through clever use of other instructions. Left shifts can be replaced by repeatedly adding a number to itself as many times as the shift amount (effectively doubling the number every time); this technique can also be applied to the 16-bit left shift required by *load-upper-immediate* (`lui`) instructions. The right shift operation is more challenging, but it can be replaced by a method similar to software division that performs successive subtractions; note that *shift right arithmetic* (`sra`) requires sign extension to the most significant bits, while *shift right logical* (`srl`) does not. We found that supporting shift operations only in software resulted in unacceptable cycle-time increases—orders of magnitude for many applications; hence we are motivated to compromise with hardware support for a small number of fixed value shifters.

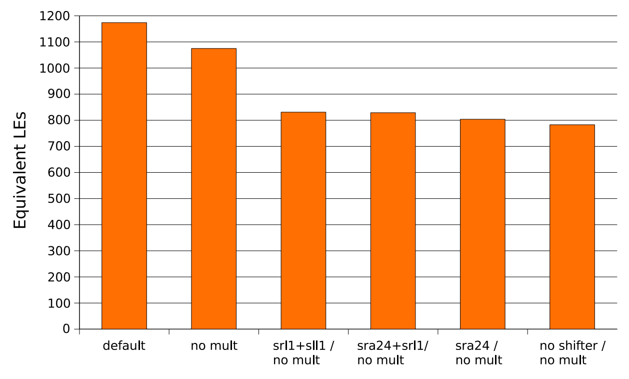
We investigate the impact of having up to two fixed-amount hardware shifters in lieu of a variable-amount shifter, as shown in Table 2. We decided which are the best two fixed-

Table 2. Selection and impact of the two fixed-amount hardware shifters for each benchmark that provide the maximum cycle count improvement. The last column represents the fraction of original shifts that are not directly translated to a number of fixed-function shifts.

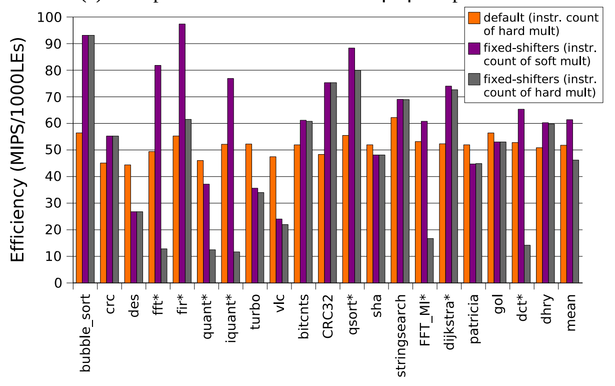
	1st Shifter		2nd Shifter		Relative	% Shifts not
		Shift		Shift	Increase	Fully
Benchmark	**Type**	**Amt.**	**Type**	**Amt.**	**in Cycles**	**Translated**
bubble_sort	-	-	-	-	1	-
crc	srl	24	sll	2	1.27	29%
des	srl	1	sll	1	2.58	0
fft	srl	1	sll	1	1.18	0
fir	srl	1	sll	1	1	0
quant	srl	1	sll	1	2.44	0
iquant	srl	1	sll	1	1.29	0
turbo	srl	2	sll	8	2.39	51%
vlc	srl	1	sll	1	3.2	0
bitcnts	srl	4	srl	1	1.33	0
CRC32	srl	8	sll	2	1	48%
qsort	srl	1	sll	1	1	0
sha	srl	1	sll	5	1.68	49%
stringsearch	sra	24	sll	2	1.02	18%
FFT_MI	srl	1	sll	1	1.57	0
dijkstra	srl	1	sll	1	1.11	0
patricia	srl	1	sll	24	1.8	61%
gol	sra	24	sll	1	1.66	33%
dct	srl	1	sll	1	1.47	0
dhry	sra	24	sll	1	1.33	35%

amount shifters for each benchmark based on the projected total dynamic cycle savings of each. Note that this calculation accounts for the fact that any shift operation that requires a *multiple* of one of hardware shift-amounts may be implemented through a software routine that calls the hardware shifters an appropriate number of times. From the table it is apparent that left and right logical shifts of 1 bit are the most beneficial, followed by shifts of 24-bits. We also report the increase in dynamic cycles relative to the default implementation with software multiplication (and a variable-amount hardware shifter). The increase in cycles ranges to negligible for 5 benchmarks to a worst case of 2.58 for DES, and a mean increase of 1.57 across all benchmarks which seems to be reasonable enough to be exploited as an area/performance trade-off. Finally, we report the percentage of original shifts that are not fully translated to a number of fixed-function shifts instructions but rather require software routines (that may in turn use the fixed-function hardware shifters, in particular for divisions).

Figure 4(a) shows the area impact of gradually decreasing hardware support for shifting for the `serial` processor (default), including two common choices of fixed-amount hardware shifter pairs (`srl1` & `sll1`, and `sra24` & `srl1`). The frequency of those processors is increased by 1% when removing the multiplication support and 8% on average when removing the shifter or having fixed-function shifters. Figure 4(b) shows the area efficiency of processors with up to 2 fixed-function shifters. To compute the area efficiency of this optimization, we first use the instruction count of the benchmarks with software multiplies to compare constant amounts of work. We find that area efficiency is improved by 18% on average across all benchmarks (with a standard deviation of 37%). Also in Figure 4(b), we show the efficiency



(a) Comparison of variants of the `pipe3` processor



(b) Area efficiency of up to 2 fixed-function shifters per benchmark

Fig. 4. Results showing: (a) the area cost for variants of the `pipe3` processor, including two popular fixed-amount shifter configurations from Table 2; (b) the area efficiency for a `pipe3` processor in its default configuration or equipped with up to 2 fixed-amount shifters. The source of instruction count to compute the MIPS value is indicated. Starred benchmarks (*) require multiplications.

of processors with fixed-function shifters when using the instruction count of the default processors equipped with hardware shifters. We can see that having soft multiplies and fixed-function shifters proves to be more area efficient for 3 benchmarks that use a hardware shifter (FIR, QSORT and DIJKSTRA).

B. Removing Hazard Detection Logic

A nice feature of SPREE is that it automatically generates hazard detection logic which stalls the pipeline so that register dependences are observed. However, hazard detection logic consumes a non-trivial fraction of processor area: roughly 10% or 110 LEs. Alternatively, the compiler could become responsible for observing register dependences, implemented through instruction scheduling where possible and insertion of `no-op` instructions as a last resort. Figure 5 shows the potential benefits of removing hazard detection logic, which are an area savings of 10% for `pipe3` and `pipe5`, and 6% for `pipe7`, and an increase in clock frequency of 3% for `pipe3`, and 6% for `pipe5` and `pipe7`. The `serial` processor is not affected by this transformation because it has no hazard detection logic. Since these results are promising, in future work we will investigate the impact on cycle count,

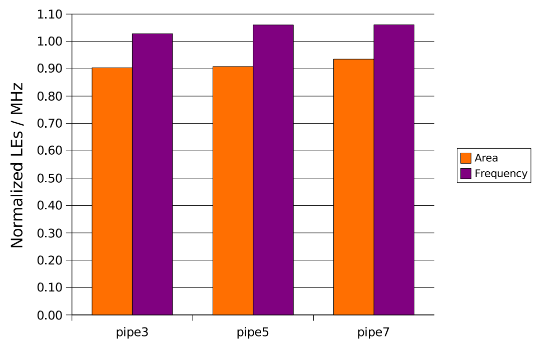
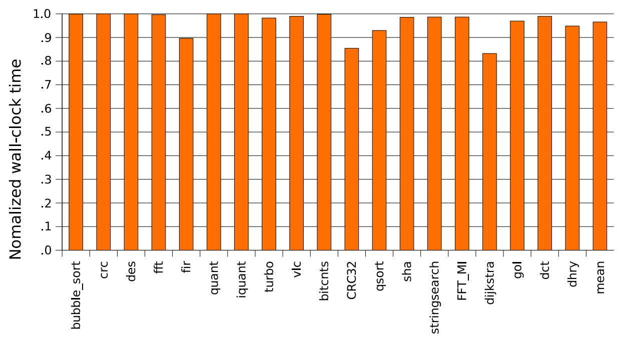
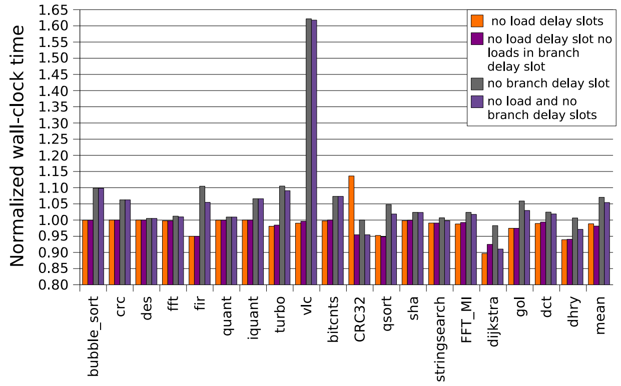


Fig. 5. Measurements of various soft processors with hazard detection logic removed, normalized to the corresponding soft processors having hazard detection logic.



(a) Removing load delay slots, `serial` processor



(b) Removing delay slots, `pipe3` processor

Fig. 6. Impact on the wall-clock time of removing delay slots, normalized to the corresponding default compilation/processor (with delay slots).

```
loop_start:
  branch loop_start
  nop
  load
  nop
```

(a) With the load delay slot.

```
loop_start:
  branch loop_start
  load
```

(b) Without the load delay slot.

Fig. 7. Code showing a load instruction scheduled into a branch delay slot by the compiler as a side-effect of the removal of the load delay slot.

code size, and overall performance of compiler scheduling and no-op insertion. However, note that such compiler scheduling can be non-trivial, for example to account for variable-cycle operations such as shifts—a practical solution may be to only partially remove hazard detection for simple cases.

IV. Impact of Unique ISA Features

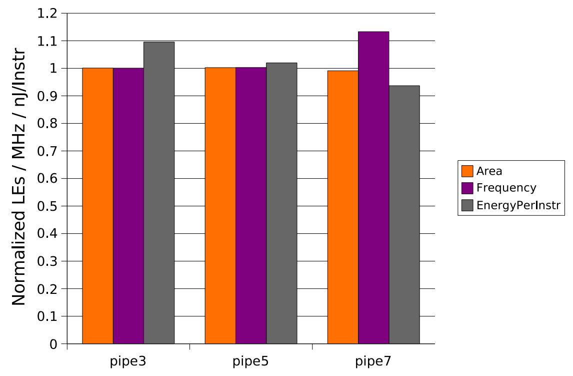
Customizable and parametric processors are often built on a base RISC ISA, which can then be extended with custom instructions. Depending on the base ISA, there may be unique ISA features which may or may not benefit a given application. Since our infrastructure is based on the MIPS ISA, we investigate the MIPS-specific features of load and branch delays slots, `Hi`/`Lo` registers, and unaligned memory references; for example, the Nios II ISA is similar to MIPS, although it does not support any of those features. Hence we are motivated to evaluate the impact of these features.

A. Load Delay Slots

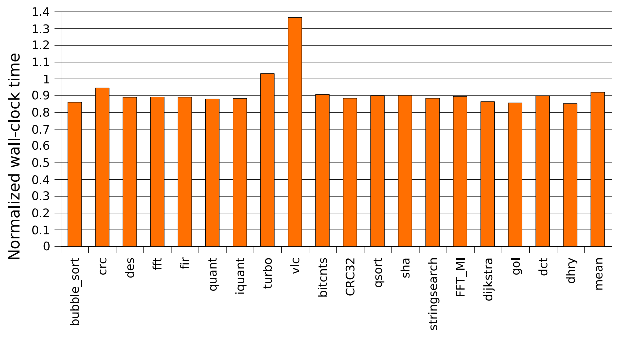
The MIPS instruction set has two delay slots: one that follows load instructions, and one that follows branch and jump instructions. A delay slot is a placeholder in which an instruction may be scheduled, so long as it does not depend on the result of a load, or will be executed regardless of whether the corresponding branch is taken; if there is no appropriate instruction to occupy a delay slot, a no-op instruction is used. Delay slots are useful in helping tolerate delays due to hazards in a processor's pipeline. Note that there is a negligible hardware cost for supporting load delay slots, while branch delay slots can complicate several aspects of pipeline control logic.

Figure 6(a) shows the impact on wall-clock time of removing the load delay slots on the `serial` processor. Since this processor is not pipelined and has a one-cycle memory access latency, load delay slots have no benefit and removing them only improves wall-clock time. We also evaluate removal of load delay slots for the 3-stage pipelined processor `pipe3`, as shown in Figure 6(b): on average this results in a small (1%) reduction in wall-clock time due to cycle count savings, although the savings for some benchmarks is significant. For pipelined processors, the forwarding lines can reduce stalls and make load delay slots unnecessary (again, since we have a 1-cycle access to the memory system).

For CRC32 removing the load delay slot leads to a slowdown of 14% due to unfortunate circumstances: as illustrated in Figure 7, the compiler scheduled a load in a branch delay slot, such that the load is then unnecessarily executed along with every execution of the branch. In contrast, when a load delay slot is supported the branch delay slot is occupied by a no-op and the load is only executed whenever the branch is not taken. As a solution to this problematic case we implemented a compiler setting where the load delay slot is removed, but a load can never be used in a branch delay slot. In Figure 6(b)



(a) CAD metrics relative to corresponding default implementation (that implements branch delay slots).



(b) Impact on wall-clock time for the `pipe7` processor, relative to the default execution (with branch delay slots).

Fig. 8. Impact of removing the branch delay slot.

we evaluate this setting (the 2nd bar), but find that it is a compromise: it always improves on the baseline but cannot achieve the full benefit of simply removing the load delay slot in some benchmarks.

B. Branch Delay Slots

A branch delay slot provides an extra cycle to compute the target of the branch in a pipelined datapath, before the program counter is updated with either the branch target or fall-through locations—hence the delay slot instruction following a branch must be executed regardless of whether the branch is taken. Accounting for branch delay slots requires additional control logic and increases the complexity of the processor, and hence is a potential area/performance trade-off in itself. Figure 8(a) shows the impact on the processor metrics of removing support for branch delay slots, which is negligible in terms of area and frequency except for a 13% increase in clock frequency for the 7-stage pipeline. This frequency improvement is due a change in the critical path of the processor that occurs only for that particular processor. Figure 8(a) also indicates that `pipe3` performs a significant amount of additional work, considering that it should use less energy-per-cycle since the branch delay hardware is removed.

In Figure 6(b), we show that removing the branch delay slot for the 3-stage pipeline increases the number of cycles because

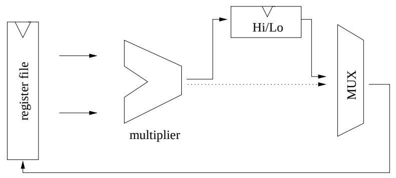


Fig. 9. Schematic of the `Hi`/`Lo` circuitry. The solid line represents the default MIPS implementation, while the dashed line represents the proposed elimination of `Hi`/`Lo` registers.

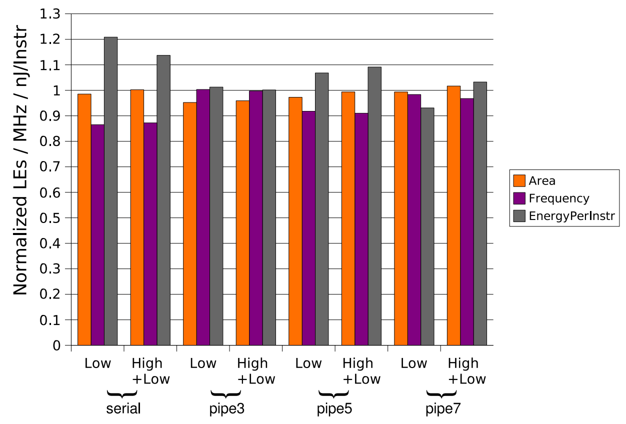
our processor simply assumes that branches are not taken—i.e. all instructions executed after the branch must be squashed when the branch is taken. In Figure 8(b) removing branch delay slots from the 7-stage processor reduces wall-clock time by an average of 8%, which is a significant improvement—this is due entirely to an increase in clock frequency, as the average cycle count actually increases in this case. We are currently implementing more sophisticated branch prediction support so that we may more thoroughly study the potential of customization of branches and their delay slots.

C. 3-Operand Multiply vs `Hi`/`Lo` Registers

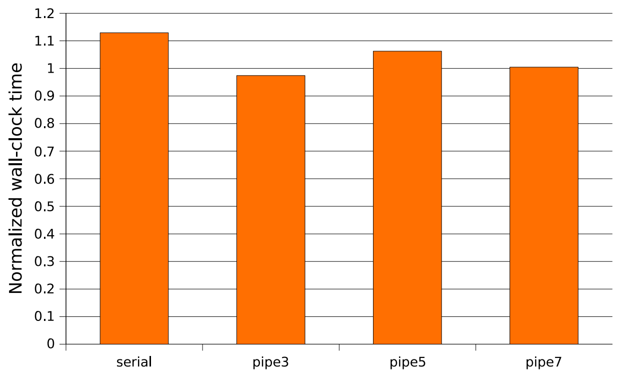
On a 32-bit architecture, the multiplication of two registers results in a 64-bit product of which the 32 most significant bits are called the *high part* and the 32 least significant bits are called the *low part*. In a MIPS processor, special registers called `Hi` and `Lo` hold the result of a multiplication so the destination of a multiplication is implicit. To become accessible to the ALU, the high and low parts of the result must be loaded in the register file by two separate instructions `mfhi` and `mflo`. Figure 9 shows the two registers that are used exclusively for the multiplication (since our processors support only software division). Those registers were originally introduced to reduce the scheduling complexities of the multi-cycle multiply and divide instructions and because they had hardware interlocks, while the rest of the processor did not.

To evaluate the costs/benefit of this particular feature of the ISA, we optionally support a three-operand multiply (similar to the NIOS II [1] or Microblaze [2] ISAs), where the destination register may be any general-purpose register, and is explicitly defined in the instruction encoding. Since only one 32 bit destination register may be specified, we require two multiply instructions: one to compute the high part of the multiplication, and one to compute the low part. A side-benefit of this approach is that only one multiplication instruction need be used if only the low part of the operation is required. We found that only the low part of multiplication is required for FFT, FIR, QUANT, IQUANT, and QSORT benchmarks, while FFT_MI, DIJKSTRA and DCT require 64 bit multiplication results (the remaining benchmarks do not contain multiplies).

Figure 10(a) shows the impact of 3-operand multiplies relative to the corresponding default multiplier implementation for the different pipelined processors. While there is a modest area savings (2% on average) due to elimination of the actual `Hi` and `Lo` registers (which are cheap in an Altera FPGA), processor frequency suffers significantly in



(a) CAD metrics for processors that implement only one instruction to compute the low part (*Low*) or two instructions to compute both the high and low parts (*High + Low*) relative to the corresponding default multiplier implementation.



(b) Impact on wall-clock time normalized to the execution with the default multiplier averaged over all benchmarks that contain multiplications (see Table 1).

Fig. 10. Impact of 3-operand multiplies.

most cases because the write-back path from the multiplier to the register file becomes a critical path. However, we find that the average cycle count is reduced by 2% for the 3-operand multiplication (with a standard deviation of 3%), due to a reduction on average of the number of instructions required for multiplication: when only the lower 32-bit result is required, only the one 3-operand multiply instruction is required, while for the 2-operand multiply instruction a `mflo` instruction is additionally required. Finally, Figure 10(a) shows that 3-operand multiplies generally require more activity in the processor, leading to a greater energy per instruction consumption and countering expected dynamic power savings due to a slower clock speed [21].

Figure 10(b) shows that wall-clock time is improved by 3% on average for the 3-stage pipeline, but unchanged for the 7-stage pipeline. Taking area into consideration, our conclusion is that the 3-operand multiplication (along with the removal of the `Hi`/`Lo` registers) is beneficial only for our 3-stage pipelined processor.

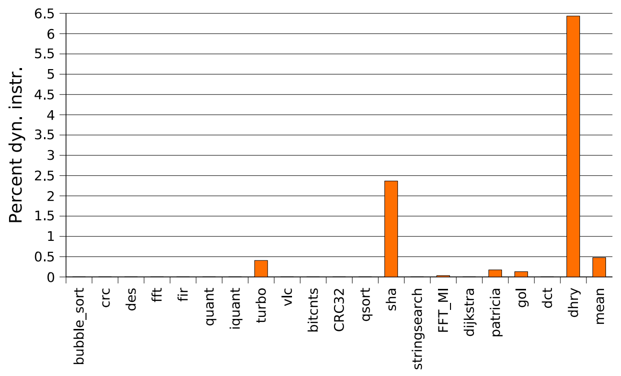


Fig. 11. Percentage of dynamic instructions removed with the addition of the patented instructions performing unaligned memory accesses.

D. *Unaligned Loads and Stores*

While the instructions `lwl`, `lwr`, `swl`, and `swr` have patent restrictions and are thus not supported by SPREE, they can be generated by `gcc`. These perform unaligned memory loads and stores, effectively comprising memory references with shift operations. In absence of those instructions, compilers typically use padding to align data to word boundaries. Since padding is not always possible, it is important to measure the cost/benefit of these instructions. In Figure 11, we show the reduction in dynamic instructions through the addition of these more powerful instructions. For TURBO, SHA, and DHRY, this savings is significant, but on average these instructions only reduce the cycle count by 0.5% and hence are not generally worth supporting.

V. Application-Specific Register Management

For a soft processor, the set of architected registers in the base ISA and their conventional uses may not necessarily match the needs of the target application, or may miss opportunities for a more efficient architecture. In this section we present and evaluate two techniques that customize the compiler's use of registers to applications: operand scheduling, and limiting the use of architected registers.

A. *Operand Scheduling*

To reduce stalls due to data hazards between registers in pipelined processors, designers employ forwarding lines to forward the result computed in a later stage directly to an earlier stage, bypassing the register file. In our soft processor designs, we optionally support one pair of forwarding lines (see Figure 1)[1]. Since operands for instructions that implement a commutative operation (such as `add` may be freely exchanged, our insight is that we could bias operands with near-distance register dependences to favor a given operand position in the instruction, potentially allowing us to reduce the performance impact of removing one of the two forwarding lines from our processors. Our algorithm for scheduling operands is as follows. For each instruction, we

[1]Additional forwarding lines are not possible in these datapaths.

Table 3. Percentage cycle count savings of forwarding lines and operand scheduling, relative to the corresponding default processor with no forwarding lines (and no scheduling), averaged across all benchmarks (i.e., 0% means no cycle savings).

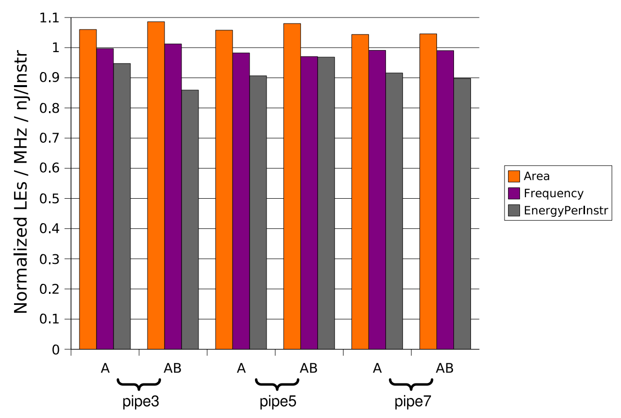
Processor	Fwd A	Fwd A + Scheduling	Fwd AB
`pipe3`	10%	11%	14%
`pipe5`	12%	14%	17%
`pipe7`	9%	11%	15%

traverse a history of instructions in the static program order—from the most recent to the oldest—to find read-after-write dependences, and to adjust the order of the operands to take advantage of the supplied forwarding lines. Care is taken not to affect the register allocation which could counter/undo our operand scheduling.

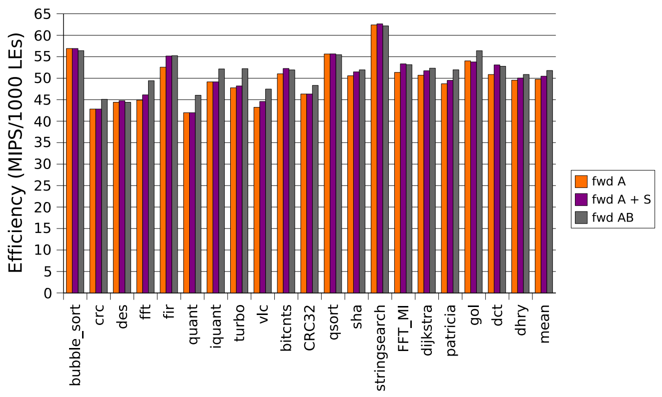
Table 3 shows the impact of forwarding lines and operand scheduling on the 3, 5, and 7 stage pipelines. We find that the addition of a single forwarding line improves the average cycle count by 9 to 12 percent for the different processors, and that the addition of compiler operand scheduling provides an additional 1 or 2 percent average improvement (but up to 8% for some benchmarks). Note that we observed that `gcc` already favors one operand, hence our scheduling efforts are on top of that bias. Addition of a second forwarding line further improves cycle count by 3 to 5 percent. In summary, while operand scheduling provides an improvement over a single forwarding line at no hardware cost, it cannot equal the benefits of an additional forwarding line.

Although our algorithm improves the effectiveness of a single forwarding line, unexploited forwarding opportunities still remain for two reasons: (i) for each instruction we can only choose one permutation of its operands; and (ii) our algorithm does not predict control flow. Figure 13 illustrates the three situations where missed forwarding opportunities occur. On average with the 3, 5 and 7 stage pipelines with a single forwarding line, the breakdown of missed forwarding opportunities after operand scheduling is as follows: 8% for commutative operations with forward branches (Figure 13(a)); 10% for commutative operations with backward branches (Figure 13(b)); 82% for non-commutative operands (Figure 13(c)). The most frequently occurring non-commutative instructions that result in missed forwarding opportunities are store instructions (`sb`, `sw`), and the set-less-than instruction (`slt` and `sltu` set a register if a comparison is true). These results motivate future improvements to our algorithm to schedule non-commutative operands. One available option would be to change the ISA definition on a per-application basis to choose the best average operand permutation for some non-commutative instructions.

Figure 12(a) shows the impact of forwarding lines on the maximum frequency, the area and the energy-per-cycle of our pipelined processors. Area is increased in all cases by less than 10% and energy-per-instruction is reduced on average by 8%, meaning that the forwarding logic consumes less energy than



(a) CAD metrics for forwarding lines (either A, or both A and B), normalized to the corresponding processors without forwarding lines.



(b) Comparison of the area efficiency of the `pipe3` processor with forwarding for one operand (*fwd A*), plus operand scheduling (*fwd A + S*), and with forwarding for both operands (*fwd AB*).

Fig. 12. Impact of forwarding lines and operand scheduling.

```
  r1 = r2 + r3
  branch start
  r4 = r1 + 4
start:
  r5 = r4 + r1
```

(a) Forward branch.

```
  r1 = r2 + r3
loop_start:
  r3 = r1 + r4
  r4 = r3 + 1
  branch loop_start
```

(b) Backward branch.

```
r1 = r2 + r3
r4 = r5 - r1
```

(c) Non-commutative operands.

Fig. 13. Examples of missed forwarding opportunities. Italicized register names show a register that is written then read. Assuming support for forwarding the operand in the first source position, the read instruction has the register as a second operand because of other operand scheduling constraints in (a) and (b), or because of properties of the instructions (c).

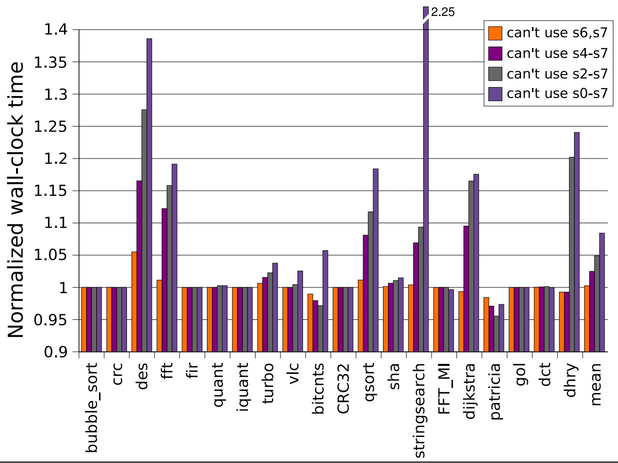


Fig. 14. Impact on wall-clock time of increasingly limiting the number of registers available to the compiler for the `serial` processor.

what is wasted on stall cycles due to data hazards. Surprisingly, certain processor configurations have an improved maximum operation frequency which should be considered within the noise margin of the placement and routing of the FPGA. As seen in Figure 12(b), because of the area cost of having two forwarding lines, compiler support allows some benchmarks (such as BUBBLE_SORT, DES and STRINGSEARCH) to be equally or more efficient with a single forwarding line than with two forwarding lines. While compiler support improves area efficiency of this processor by 2% on average (and up to 5% for FIR), a single forwarding line remains less area efficient than two forwarding lines overall (3% degradation).

B. Limiting Use of Architected Registers

Not all applications require the use of all architected registers in a base ISA to maintain good performance, and for other applications limiting the number of registers accessible by the compiler has a tolerable impact on performance. For an FPGA-based soft processor, since the register file is typically implemented using a block memory, the memory space freed by reducing the number of architected registers is not easily reclaimed by the rest of the FPGA design. However, these free register locations could potentially be exploited by new custom instructions or functional units, for a tighter integration with the processor.

In this section we evaluate the impact of limiting the use of certain architected registers for the base MIPS ISA, using `gcc` with full optimization (`-O3`). In particular, for now we examine the MIPS convention of reserving two registers for operating system purposes (`k0-k1`), and eight registers for caller-saving across a function call (`s0-s7`). We modified `gcc` to use `k0-k1` as general purpose registers but observed no significant application speedup over all our benchmarks, meaning that an increased number of registers was not helpful. We thus reverted `gcc` to not using `k0-k1` and modified it so that it does not use some registers in the `s0-s7` register range.

For our embedded benchmark set, Figure 14 shows that several applications do not fully take advantage of the 32 registers assumed by the MIPS ISA: only DES incurs an observable slowdown when removing 2 registers from the default compilation. The fact that the BITCNTS and PATRICIA benchmarks encounter a small speedup is an unexpected side-effect of register allocation, instruction scheduling and forwarding opportunities. The unpipelined processor in Figure 14 suffers the most from fewer registers among our reference processors because memory spills due to register pressure result directly in additional processor waiting cycles for memory. We verified that some benchmarks did not use any of the `s0-s7` registers with the default optimized compilation. Removing some of the 10 callee saved registers (`t0-t9`) was not yet attempted.

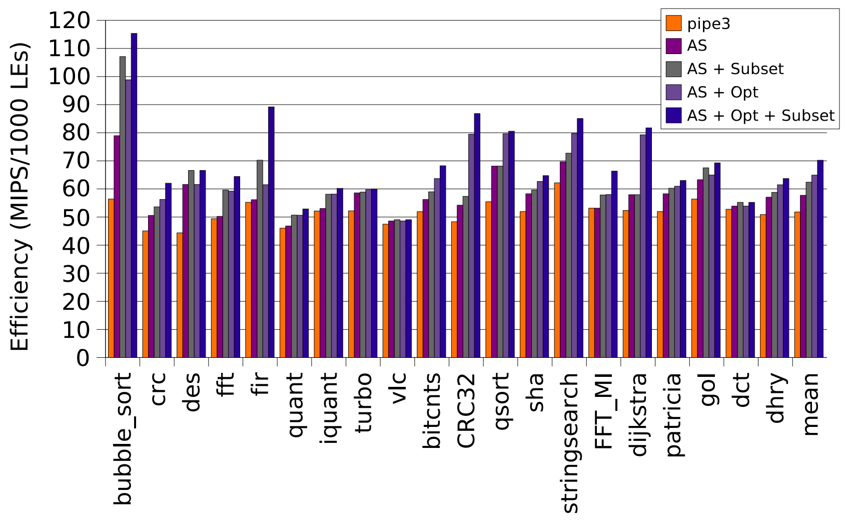
VI. Combining Customization Techniques

In this section we evaluate the impact of combining the compiler optimizations described in this paper, and their interaction with application-specific architecture and ISA subsetting as detailed in a previous publication [9].

In Figure 15, the first bar shows the area efficiency for the `pipe3` processor, since overall it is the most area efficient over all our benchmarks. In other words, we would choose `pipe3` if we required the one most efficient processor to support all benchmarks. We use `pipe3` as the comparison basis for our application-specific optimizations. For the second bar (*AS*), we select the most area-efficient processor architecture for each application (considering as design options shifter implementations, pipeline depth and forwarding lines, hardware vs software multiplication support), in a similar manner to earlier work [9] but with the addition of forwarding lines as a design option. Choosing an application-specific processor design improves efficiency by 11% on average, illustrating the power of customization for soft processors. For the third bar (*AS + Subset*), to the best application-specific processor we additionally apply ISA subsetting (removal of the processor support for any instructions that are unused by that application [9]). Subsetting further improves efficiency by an additional 8% on average, although for some applications, such as FIR and BUBBLE_SORT, the benefit is much greater since they have a large number of unused instructions.

For the fourth bar (*AS + Opt*), to the best application-specific processor we apply the most effective combination of the following compiler techniques: (i) custom fixed-amount shifters, (ii) delay slot removal, (iii) 3-operand multiplication, and (iv) operand scheduling—the remaining optimizations (compiler-managed hazard detection, unaligned memory accesses, and register elimination) are not evaluated here because the SPREE infrastructure does not yet either support or exploit them. The table in Figure 15 shows the combination of compiler optimizations selected for each application. Our optimizations provide an average improvement of 7 MIPS/1000 LEs (12%) over the application-specific processor (*AS*); the maximum improvement (for CRC32) is 25 MIPS/1000 LEs (47%), mostly due to the effectiveness of the fixed-function shifters.

For the fifth and final bar (*AS + Opt + Subset*), we evaluate the combination of our optimizations and subsetting on



Benchmark	Fixed Amount Shifters	Load Delay Slot Removal	3-op Mult.	Oper. Sched.
bubble_sort	✓	✓		
crc	✓	✓		
des		✓		
fft		✓	✓	
fir	✓	✓		
quant		✓	✓	
iquant		✓	✓	
turbo		✓		
vlc		✓		
bitcnts		✓		✓
CRC32	✓	✓		
qsort	✓	✓		
sha		✓		
stringsearch	✓	✓		
FFT_MI		✓	✓	✓
dijkstra	✓	✓		
patricia		✓		
gol		✓		
dct		✓	✓	✓
dhry		✓		

Fig. 15. Area efficiency of the `pipe3` processor, best application-specific processor (*AS*), and the best application specific processor with: subsetting (*AS + Subset*); compiler optimization (em AS + Opt); and both improvements (*AS + Subset + Opt*). The table describes which optimizations were beneficial and hence enabled for each benchmark.

the application-specific processors, which improves efficiency over subsetting alone by 13% on average. We also find that our optimizations and subsetting can be complementary: for example, for FIR the efficiency of optimizations and subsetting is greater than the sum of the gain of each individually by 18 MIPS—a result of improved FPGA placement and routing. The best improvement (either relative or absolute) over the best application-specific processor with subsetting (*AS + Subset*) is 29 MIPS (51%) for CRC32. This illustrates that our optimizations can significantly impact design decisions when area, frequency and wall-clock time must be taken into consideration.

VII. Conclusions

In this paper we have presented a customization approach that consists of adapting code generation to make it more efficient by revisiting traditional architectural and ISA assumptions. We have illustrated several trade-offs between area, power, operating frequency and wall-clock time. We found: (i) that we can improve area efficiency by replacing a variable-amount shifter with two fixed-amount shifters; (ii) that hazard detection logic hinders the processor's area and operating frequency; (iii) that we can eliminate load delay slots in most cases; (iv) that branch delays slots can be removed in a 7-stage pipeline even with no branch prediction; (v) that 3-operand multiplies are only justified for our 3-stage processor (and that otherwise `Hi`/`Lo` registers are best); (vi) that unaligned memory loads and stores do not provide a significant performance benefit for our benchmarks; (vii) that we are able to remove one forwarding line with simple operand scheduling and improve area efficiency; and (viii) that we can limit the compiler's use of a significant fraction of the 32 architected registers for many benchmarks without degrading performance. To maximize the efficiency of the customized architecture of our soft processors, we combined several of these optimizations and obtained a 12% additional area efficiency increase on average (and up to 47% in the best case). By including subsetting and our optimizations, the mean improvement is 13% but the maximum is 51%.

In the future, we will study in greater depth the potential for optimizations that focus on branch prediction and the memory system. We also plan to develop methods for automatically deciding at compile time the best optimizations and architectural features for a specific application.

References

[1] J. Ball, "The Nios II Family of Configurable Soft-Core Processors," Hot Chips, Altera, August 2005.

[2] Xilinx Inc., "MicroBlaze RISC 32-Bit Soft Processor," August 2001.

[3] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," in *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*. New York, NY, USA: ACM Press, 2004, pp. 69–78.

[4] Altera Corporation, "Nios II C-to-Hardware Acceleration Compiler," http://www.altera.com/c2h.

[5] 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.

[6] R. Lysecky and F. Vahid, "A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 18–23.

[7] J. Cong, Y. Fan, G. Han, A. Jagannathan, G. Reinman, and Z. Zhang, "Instruction set extension with shadow registers for configurable processors," in *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. New York, NY, USA: ACM Press, 2005, pp. 99–106.

[8] P. Yiannacouras, J. Rose, and J. G. Steffan, "The microarchitecture of FPGA-based soft processors," in *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*. New York, NY, USA: ACM Press, 2005, pp. 202–212.

[9] P. Yiannacouras, J. G. Steffan, and J. Rose, “Application-specific customization of soft processor microarchitecture,” in *FPGA’06: Proceedings of the internation symposium on Field programmable gate arrays*. New York, NY, USA: ACM Press, 2006, pp. 201–210.

[10] A. Shrivastava, E. Earlie, N. Dutt, and A. Nicolau, “Operation tables for scheduling in the presence of incomplete bypassing,” in *CODES+ISSS ’04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM Press, 2004, pp. 194–199.

[11] MIPS Technologies Inc., “The MIPS RISC architecture,” http://www.mips.com, MIPS Technologies.

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

[13] F. Campi, R. Canegallo, and R. Guerrieri, “IP-reusable 32-bit VLIW RISC core,” in *Proc. of the 27th European Solid-State Circuits Conf*, September 2001, pp. 456–459.

[14] R. Weiker, “Dhrystone 2.1,” *SIGPLAN Notices*, 23(8), Freescale, August 1988.

[15] L. Shannon and P. Chow, “Standardizing the performance assessment of reconfigurable processor architectures,” in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003, pp. 282–283.

[16] S. A. Przybylski, T. R. Gross, J. L. Hennessy, N. P. Jouppi, and C. Rowen, “Organization and VLSI implementation of MIPS,” Stanford University, Stanford, CA, USA, Tech. Rep., 1984. [Online]. Available: http://historical.ncstrl.org/litesite-data/stan/CSL-TR-84-259.pdf

[17] P. Yiannacouras, “The Microarchitecture of FPGA-Based Soft Processors,” Master’s thesis, University of Toronto, 2005.

[18] Altera Corporation, “Quartus II,” San Jose, CA, USA, Altera.

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

[20] J. Veenstra and R. Fowler, “MINT: a front end for efficient simulation of shared-memory multiprocessors,” in *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, NC, USA, January 1994, pp. 201–207.

[21] L. Shang, A. S. Kaviani, and K. Bathala, “Dynamic power consumption in Virtex^{TM}-II FPGA family,” in *ACM/SIGDA 10th International Symposium on Field-programmable gate arrays*, Monterey, California, USA, February 2002.