An Energy-Efficient Method of Supporting Flexible Special Instructions in an Embedded Processor with Compact ISA

DONGRUI SHE, YIFAN HE, and HENK CORPORAAL, Eindhoven University of Technology

In application-specific processor design, a common approach to improve performance and efficiency is to use special instructions that execute complex operation patterns. However, in a generic embedded processor with compact Instruction Set Architecture (ISA), these special instructions may lead to large overhead such as: (i) more bits are needed to encode the extra opcodes and operands, resulting in wider instructions; (ii) more Register File (RF) ports are required to provide the extra operands to the function units. Such overhead may increase energy consumption considerably.

In this article, we propose to support flexible operation pair patterns in a processor with a compact 24-bit RISC-like ISA using: (i) a partially reconfigurable decoder that exploits the pattern locality to reduce opcode space requirement; (ii) a software-controlled bypass network to reduce operand encoding bit and RF port requirement. An energy-aware compiler backend is designed for the proposed architecture that performs pattern selection and bypass-aware scheduling to generate energy-efficient codes. Though the proposed design imposes extra constraints on the operation patterns, the experimental results show that for benchmark applications from different domains, the average dynamic instruction count is reduced by over 25%, which is only about 2% less than the architecture without such constraints. The proposed architecture reduces total energy by an average of 15.8% compared to the RISC baseline, while the one without constraints achieves almost no improvement due to its high overhead. When high performance is required, the proposed architecture is able to achieve a speedup of 13.8% with 13.1% energy reduction compared to the baseline by introducing multicycle SFU operations.

Categories and Subject Descriptors: B.1.4 [Control Structures and Microprogramming]: Microprogram Design Aids; C.1 [Processor Architectures]; D.3.4 [Programming Languages]: Processors

General Terms: Algorithms, Design

Additional Key Words and Phrases: Reconfigurable architecture, special instruction, low power, code generation

ACM Reference Format:

She, D., He, Y., and Corporaal, H. 2012. An energy-efficient method of supporting flexible special instructions in an embedded processor with compact ISA. *ACM Trans. Architect. Code Optim.* 10, 3, Article 15 (September 2013), 25 pages.

DOI: <http://dx.doi.org/10.1145/2509420.2509426>

1. INTRODUCTION

Mobile devices like smart phones are becoming key elements in improving life quality and productivity. The rapid development in embedded processors enables such devices

A preliminary version of this article appeared at the 2012 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES12).

This work is supported in part by the Dutch Technology Foundation STW, project NEST 10346, and the Ministry of Economic Affairs of the Netherlands, project EVA PID07121.

Authors' addresses: D. She (corresponding author), Y. He, and H. Corporaal, Electronic Systems Group, Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, the Netherlands; email: d.she@tue.nl.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/09-ART15 \$15.00

DOI: <http://dx.doi.org/10.1145/2509420.2509426>

Fig. 1. Special operation patterns.

to run high-performance applications, for example, wireless communication and high-definition video codecs. However, the advance in power supply technology cannot keep up with the exponential growth of energy demand in such devices. As a result, power efficiency is becoming a bottleneck in high-performance embedded systems, especially for those ones that run on limited power sources like batteries. Therefore there is a great need for improving the energy efficiency of embedded processors. Many applications contain frequently executed operation patterns in the DataFlow Graphs (DFGs), like the ones in Figure 1. In this work, *special instructions* are defined as instructions that execute such patterns. When properly utilized, special instructions are able to dramatically reduce the number of instructions and the amount of communication between datapath components, which has great impact on performance as well as energy consumption. In Application-Specific Instruction set Processor (ASIP) design, it is common to synthesize instruction sets that support such patterns to achieve better performance and energy efficiency [Karuri et al. 2009; Clark et al. 2003; Leupers et al. 2006]. And we have already seen commercial products that adopt this approach, for example, Tensilica Xtensa [Tensilica 2013; Leibson 2006], Stretch software configurable processors [Stretch 2013; Gonzalez 2006], and IMEC ADRES processor [Mei et al. 2003].

In this article, we tackle the problem of integrating flexible special instruction support in a generic embedded processor with a compact Instruction Set Architecture (ISA). Most previous works focused on improving the performance [Clark et al. 2003, 2004; Karuri et al. 2008]. Besides performance improvement, the main focus of this work is on energy efficiency of the processor for different types of applications. The successful uses of special instructions in ASIPs have shown that it is an effective approach towards improving energy efficiency. But in most mainstream processor architectures, only a few of such instructions are used, as supporting arbitrary operation patterns in a generic processor incurs large overhead. From an energy-efficiency perspective, the overhead includes the following.

- There are more bits in the instruction to encode opcodes for all possible patterns and extra operands in the special instructions. In a compact ISA like the ARM Thumb [ARM 2013], the problem is even more serious as the number of bits in the instruction is very limited. If the instruction is made wider, fetching an instruction consumes more energy.
- There are more ports in the Register File (RF) to provide sufficient data bandwidth for the special function units. An RF with more ports is much less energy efficient. Also, even the normal instructions need to pay the extra cost. Methods like register file clustering [Karuri et al. 2007] or FU internal registers [Leupers et al. 2006] are able to partially solve the problem. But such methods usually lack flexibility and in addition, often lead to very complex code generation.
- There is increased complexity in many parts of the processor, including the instruction decoder and bypassing network.

To achieve high energy efficiency, the support for special instructions needs to have low overhead, while still being able to support applications from different domains. In this article, we propose a scheme for integrating a special instruction unit (SFU) into a RISC-like embedded processor with an ISA that has 24-bit instruction width. The SFU supports flexible *operation pair* patterns. To integrate the SFU into the RISC datapath with minimum overhead, we introduce the following.

- A partially reconfigurable decoder allows low-overhead reconfiguration for each kernel to use its specific patterns. As a result, arbitrary pair patterns can be executed on the proposed architecture, while no extra bits are needed for the special instruction opcode.
- A bypass network in the datapath is exposed to software, thereby reducing the requirement, both for operand encoding and register file ports.

The use of a reconfigurable decoder and an explicit bypass network imposes some constraints on the special instructions the processor can execute, for example, for a three-input special instruction, at least one of the operands has to come from the bypass network. A compiler backend is designed to generate energy-efficient code for the proposed architecture. The compiler selects patterns and performs energy-aware instruction scheduling to utilize the SFU and the explicit bypass network. Experimental results show that for a set of benchmarks from different application domains, the proposed architecture achieves an average of 25% reduction in dynamic cycle count, which is only 2.1% worse than the architecture without constraints on the special instructions. As for energy consumption, the proposed architecture achieves an average reduction of 15.8%, while the unconstrained architecture only reduces energy by 1.1%. By introducing multicycle long-latency SFU operations, the proposed architecture is able to achieve a speedup of 13.8% with 13.1% energy reduction compared to the baseline, which is useful when high performance is essential. The key contributions are as follows.

- We propose an architecture that supports flexible operation pairs in a processor with a compact 24-bit RISC-like ISA. The proposed architecture has a partially reconfigurable decoder and a software-controlled bypass network, allowing the processor to support operation pairs without increasing the instruction width or number of register file ports. The proposed architecture is fully implemented in Verilog HDL.
- A compiler backend has been designed for the proposed architecture. It is capable of utilizing the SFU and the explicit bypass network to generate energy-efficient target code. A complete functional compiler for the target architecture is implemented based on the LLVM framework [Lattner and Adve 2004].
- Comprehensive and detailed experimental results demonstrate that the proposed architecture and compiler are able to achieve a significant improvement in energy efficiency.
- We show that by introducing multicycle long-latency operations in the SFU, the proposed architecture is able to achieve decent balance between performance and energy efficiency.

The remainder of this article proceeds as follows: Section 2 describes the DFG patterns we consider in this work and the design of the SFU that executes such patterns. The proposed integration of SFU into the processor datapath with explicit bypass is depicted in Section 3. Section 4 introduces the compiler backend design for the proposed architecture. Detailed and comprehensive experimental results that demonstrate the effectiveness of the proposed design are given in Section 5. Section 6 discusses related work. Finally, Section 7 concludes our findings and discusses future work.

2. OPERATION PATTERNS AND SPECIAL FUNCTION UNIT

Each basic block of a program can be represented by a DataFlow Graph (DFG) $G(V, E_d, E_f)$, where:

- V is a set of nodes. Each node in V represents either an actual operation or a live-in variable (stored in the register file or encoded in the instruction immediate field). In this work, we assume that the operations represented by nodes in V can be directly mapped to a Function Unit (FU) in a typical RISC processor. Such operations are defined as *basic operations*
- E_d is a set of directed edges. An edge $e = (u, v)$ represents that node v consumes the output of u , that is, there is true data dependency between u and v .
- E_f is a set of directed edges. If an edge $e = (u, v) \in E_f$, there is false/output dependency between node v and u .

For a basic block, the DFG is a Directed Acyclic Graph (DAG). A *special operation pattern* is defined as a subgraph of a DFG that contains more than one basic operation. Figure 1 shows some examples of these patterns. Compared to a combination of basic operations that performs the same computation, executing a special operation pattern using a special instruction has a few advantages.

- Fewer instructions are needed to execute the operations, resulting in less control overhead.
- The communication between the basic operations can be done within the FU, which is usually much more efficient.

For a certain application, some special operation patterns appear frequently [Arnold and Corporaal 1999]. In Application-Specific Instruction Set Processor (ASIP) design, a common approach for improving performance as well as energy efficiency is to synthesize special function units that support these patterns [Karuri et al. 2009; Clark et al. 2003; Leupers et al. 2006; Yu and Mitra 2004a]. Different from the work of ASIP design, the goal of this work is to support special operation patterns in a RISC-like generic processor, without introducing heavy modifications to the existing architecture and code generation framework. Instead of trying to support arbitrary operation patterns, we focus on a specific type of operation pattern, namely, operation pairs. The definition of the operation pair pattern, as well as motivation of choosing such patterns, are given in Section 2.1. The design of a Special Function Unit (SFU) that provides flexible support for these patterns is depicted in Section 2.2. In Section 2.3, we analyze a set of kernels based on the patterns supported by the proposed SFU.

2.1. Operation Pair Patterns

In this work, we want to integrate the support for special instructions without making major modification to the original RISC architecture. A single-issue RISC processor typically has a Register File (RF) with two read ports and one write port (2R1W). Though there are some other possible sources for input operands, like the immediate field and the bypass network, the number of source operands cannot grow dramatically without heavy modification to the instruction format. The same holds for the destination operand. In addition, the number of arbitrary operation patterns in different applications is huge. The FU that supports all these patterns becomes very complex and inefficient. So in this work, we focus on *operation pair* patterns, that is, patterns with two operations a and b that meet the following criteria.

- There is true dependency between a and b , that is, $(a, b) \in E_d$.
- There are at most three input operands. More formally, for a set of edges P that contains all edges to a or b in E_d except (a, b) , we have $|P| \leq 3$.

Fig. 2. Special function unit.

- At most only one of a and b has consumers outside the pattern, that is, at least one of the following holds:
 - The result of a is only consumed by b : $\{(a, c) | (a, c) \in E_d, c \neq b\} = \emptyset$. If this constraint is met, only b may have consumers outside the pair pattern.
 - b has no consumer: $\{(b, c) | (b, c) \in E_d\} = \emptyset$. If this constraint is met, only a may have consumers outside the pair pattern.
- There is no path from a to b in G other than (a, b) , that is, combining a and b does not create cycles, which makes the scheduling more difficult.

Integrating such patterns in a RISC processor is relatively easy: we only need to supply one more source operand than for a normal operation.

2.2. Special Function Unit Design

The design of our special function unit is shown in Figure 2. The SFU supports two levels of basic operations. To avoid introducing large area and timing overhead, only one multiplier is included in the SFU, which is put in the first level. An operand switch network is included in the SFU to allow more flexible operand encoding in a processor. The design of the SFU allows almost arbitrary combinations of operation pairs that satisfy the constraints in Section 2.1. To improve the energy efficiency of the SFU, *operand isolation* is used to isolate each subfunction unit. So a unit only toggles when it actually needs to perform computation, thereby reducing unintended circuit activity.

Due to the extra layer of subfunction units, the proposed SFU is likely to increase the delay of the critical path of the processor. When a processor with the proposed SFU needs to run at high frequency, there are two ways to mitigate the effect: (i) add a pipeline register inside the SFU, either between the first and second layer or at the output (or in the middle) of long-latency units (e.g., the multiplier); (ii) allow the long-latency operations to run in multiple cycles, thereby allowing the SFU to run at a higher frequency. By using either method a processor with the proposed SFU is able to run at a frequency close to one without the SFU.

To control the SFU, the following signals are needed: (i) opcode for each level; (ii) input data selection for each level; (iii) subfunction unit selection; (iv) extra operand

Table I. Kernel Description

Kernel	Description	Domain
FIR	5-tap finite impulse response filter	Image processing
Histogram	256-bin histogramming	Image processing
YUV2RGB	YUV to RGB color space conversion	Image processing
IDCT	2D 8x8 Inverse cosine transformation	Codec
MatVec	Matrix vector multiplication	Linear algebra
CRC	Cyclic redundancy check code calculation	Network/Storage
DES	The Data Encryption Standard algorithm	Security

Fig. 3. Cases for the same pattern that need different coding.

information, for example, the type of the immediate value. When fully decoded, the control signal for the SFU shown in Figure 2 requires 18 bits.

2.3. Application Analysis

We analyzed seven kernels listed in Table I, which come from various application domains. The DFGs of each kernel are scanned to find all possible pair patterns that can be executed by the proposed SFU. In total, 35 distinct patterns are needed. The number of patterns can grow much larger if more applications from different domains are included. In addition, to generate a valid special instruction from an operation pattern, more information needs to be encoded, for example, if there is immediate and whether the immediate is for the first or the second operation. Figure 3 shows an example of a three-input operation pair that requires different control coding. In general, the number of operation pair patterns is $N_1 \times N_2 \times V$, where N_1 is the number of operations in layer 1, N_2 is the number of operations in layer 2, and V is the number of variants introduced by operand formats described previously. The total number of different special instruction patterns is large when the SFU supports flexible operation pair patterns.

However, if we look into each individual kernel, we can see that the number of patterns used in one kernel is much smaller than the total number of patterns. Table II shows the statistics of pattern matches in the seven representative kernels from different domains. The statistics show that it is possible to exploit the *temporal locality* of patterns to reduce the number of patterns a processor needs to support

Table II. Kernel Pattern Statistics

Total	Hist	FIR	IDCT	Y2R	MatVec	CRC	DES
35	2	3	9	9	11	9	12

Table III. Key Features of the Baseline ISA

Instruction width	24 bits
Pipeline stages	4
Register file	32b×32, 2R1W
Opcode	6 bits
Immediate	8 bits

Fig. 4. A typical RISC datapath.

during the execution of an application or a kernel. Findings in Woh et al. [2009] and Yu and Mitra [2004a] also lead to a similar conclusion. This observation can be used to guide the design of efficient special instruction support in processors, which is discussed in Section 3.

3. INTEGRATING SFU INTO PROCESSORS WITH COMPACT ISA

In general, it is possible to integrate the proposed SFU design into any generic processor architecture. In this work, a 4-stage RISC processor with a 24-bit Instruction Set Architecture (ISA) is used as the baseline architecture. Here a 24-bit architecture is used to keep the baseline ISA orthogonal. But the ideas in this work still apply to more compact ISAs (e.g., a 16-bit Thumb-like ISA). The key features of the baseline architecture are described in Table III. Figure 4 depicts the datapath of the baseline processor. The four pipeline stages are fetch (IF), decode (ID), execute (EX), and writeback (WB). The ISA of the baseline is similar to the one used in OpenRISC [OpenCores 2013]. Most instructions are three-address instructions: two source operands and one destination are encoded. The baseline processor is representative of typical RISC processors used in embedded systems.

For the baseline architecture, the major limiting factors of integrating the SFU introduced in Section 2.2 are as follows.

—After adding the basic integer and control operations, only less than 16 opcodes are left in the opcode space.

Fig. 5. Datapath with unconstrained support for SFU.

- At most 3 bits can be used for encoding the extra operand in three-input instructions, which are not enough for a register index.
- The 2R1W RF cannot provide enough operand bandwidth for the SFU.

A straightforward solution to these problems is to increase instruction width and the number of RF ports. To accommodate the extra opcodes and register index, at least an additional 7 bits are needed (5 bits for the third register operand, 2 bits for extra opcodes). As a result, the width of the instruction memory increases to 32 bits. In addition, the RF needs to have three read ports (3R1W) in order to provide sufficient bandwidth for the SFU. The resulting datapath is shown in Figure 5. To avoid high area overhead, the multiplier is absorbed into the SFU. Based on the estimation of CACTI [2009], the energy consumption of each access to the instruction memory is increased by 10% to 30%, depending on the size and configuration. Based on the implementation result, the energy consumption of the RF is also increased by 12% due to the extra read port. Since both the instruction memory and the RF are among the most frequently used components, an architecture with such large overhead is unlikely to be energy efficient.

To improve the energy efficiency, this overhead has to be mitigated. In this work, we propose an energy-efficient support for the SFU by using: (i) a partially reconfigurable decoder that exploits the locality of the operation patterns to reduce the opcode encoding requirement; (ii) a software-controlled bypass network that exploits the processor pipeline to reduce the operand encoding and RF port requirements. Section 3.1 and Section 3.2 describe the details of the partially reconfigurable decoder and the software-controlled bypass network, respectively. Section 3.3 depicts how the SFU is integrated into the baseline processor.

3.1. Partially Reconfigurable Decoder for SFU

As discussed in Section 2.3, a key observation is that although a large number of patterns are needed to cover the operation patterns in different applications, only a small number of such patterns are active in one kernel, that is, in most kernels, the operation patterns have good locality. To utilize such locality, this work introduces a partially reconfigurable decoder for the SFU.

Figure 6 depicts the structure of the reconfigurable decoder for the SFU. Central to the decoder is a lookup table with eight entries, called the *pattern table*. Each entry in the pattern table stores an 18-bit control signal required by a special instruction.

Fig. 6. Partially reconfigurable SFU decoder.

Since the table only has eight entries, the free opcodes in the opcode space can be used to address it. When a special instruction is fetched, the decoder reads a pattern table entry and uses it to control the SFU; when a normal instruction is fetched, the decoder acts like a normal RISC decoder, and the pattern table is clock gated to avoid unnecessary accesses.

The pattern table is visible to the software. So when different operation patterns are needed, applications can reconfigure the SFU decoder by updating the pattern table. With reconfiguration of the pattern table, the processor is able to use all the operation patterns supported by the SFU. Reconfiguration is inexpensive in the proposed design: it only takes 10 cycles to load a complete pattern table. In most cases the operation patterns have good locality, and the overhead of reconfiguration is very low.

3.2. Explicit Bypass

In a typical pipelined datapath of a processor, like the one in Figure 4, there is a bypass/forwarding network, whose primary function is to avoid pipeline stalls caused by true data dependencies. A side-effect of such a network is that many operands can be read from the pipeline registers instead of the RF. There are two types of possible RF access eliminations introduced by the bypass network.

- Bypassing.* The result of an operation can be read from the pipeline register before it is written back to RF.
- Dead writeback elimination.* If all uses of a variable are bypassed, its writeback to RF is no longer necessary.

However, in a conventional processor architecture, such a bypass network is invisible to software, which makes it difficult to eliminate unnecessary RF accesses: (i) bypassing requires RF indexes to be checked before the decode stage, which may increase the critical path of the fetch stage, or may result in an extra pipeline stage; (ii) dead writeback elimination is impossible unless the register liveness information is explicitly encoded in the instructions. In this work, we propose to use a bypass network that is controlled by software, that is, the bypassing information is statically encoded in instructions. Figure 7 shows an example of reducing RF accesses via explicit bypassing. Besides reducing the total number of register accesses, explicit bypassing helps integrating the SFU without increasing instruction width and RF ports in two ways.

- Encoding a bypass source uses fewer bits than an RF index, since the number of possible bypass sources is much smaller than the number of registers in a typical RF (4 versus 32).
- Fewer RF ports are required when some operands are read from the bypass network.

Fig. 7. Reduce register file accesses via explicit bypassing.

Fig. 8. Special instruction example.

By imposing the constraint that at least one of the source operands in a three-input special instruction has to come from the bypass network, the special instruction can be encoded in the 24-bit instruction format, and there is no need to increase the number of register file ports. Figure 8 shows an example of using special instructions under this constraint. In processor with an unconstrained SFU, the number of instructions is reduced from 6 to 4, at the cost of increasing the number of read ports of the RF from 2 to 3. In a processor with explicit bypassing, the same code size improvement can be achieved even when there is a constraint that at least one of the operands comes from the bypass network. And with such a constraint, the requirements for the instruction bits and RF port are reduced.

3.3. Integrating SFU into Processor Datapath

We propose an architecture that is able to support all the operation pair patterns of the SFU described in Section 2.2, by employing the partially reconfigurable decoder and explicit bypass network introduced in previous subsections. Figure 9 shows the datapath of the proposed processor architecture. Note that because there are input registers for each FU, the result of one operation is stable at the output port of the FU until the next operation that uses the same FU starts. So it is possible to use the output of each FU as a separate bypass source, which increases the possibility of bypassing. Compared to the one with ideal SFU support (Figure 5), the proposed architecture imposes extra constraints on the special instructions it can execute.

- For a three-input special instruction, at least one of the source operands has to come from the bypass network.
- At most eight special instruction patterns are active at the same time. To support different patterns, the program needs to reconfigure the pattern table.

Fig. 9. Datapath with constrained support for SFU.

With these constraints, the proposed architecture is much more energy efficient: instruction width remains 24 bits instead of 32 bits and the RF is 2R1W instead of 3R1W. To use explicit bypass without changing the normal instruction format, part of the RF address space is used for the bypass source. As a result, the number of registers in the RF reduces from 32 to 28. The impact of a smaller RF is mitigated by the explicit bypassing, which eliminates the necessity of allocating registers for short-lived variables in many cases.

The introduction of a pattern table and an explicit bypass network results in extra context when an exception happens. The pattern table can be handled in a similar fashion as general-purpose registers. For explicit bypass, it is required that the processor saves the complete state for the execute and writeback stages of the pipeline. This can be done using a scan chain that automatically saves/restores the registers when an exception happens. Since the number of registers is small, the overhead in area and response time is small.

As mentioned in Section 2.2, a pipelined or multicycle SFU is required if the processor needs to run at high frequency. In this work, when the target architecture needs to run at similar frequency as the baseline RISC processor, all special instructions that use the multiplier (e.g., multiply-add) are set to finish in two cycles. With such a configuration, there are two types of resource hazards that need to be taken care of.

- SFU hazard.* A special instruction using the multiplier occupies the SFU for two cycles. Due to conflicts in subfunction units and resources like multiplexers, the next instruction in the pipeline cannot use SFU.
- Writeback hazard.* When a two-cycle instruction is followed by a single-cycle instruction, there is resource hazard in the writeback stage, since there is only one RF write port.

Both hazards can be resolved by hardware interlock. Only 2-bit extra information needs to be recorded when a special instruction is issued to the execute stage: (i) whether it is a 2-cycle instruction; (ii) whether it requires to update the writeback stage. Based on these 2-bit data and the type of the following instruction, an interlock signal can be generated. Using hardware interlock makes the hazards transparent to the software.

Fig. 10. Compiler backend flow.

4. CODE GENERATION FOR SPECIAL INSTRUCTIONS

The compiler in this work is implemented based on the open-source LLVM framework [Lattner and Adve 2004]. Figure 10 shows the flow of the compiler backend for the proposed architecture. The input of the backend is a low-level Intermediate Representation (IR), which is basically RISC assembly with virtual registers, embedded with control-flow and dataflow information. Most parts of the compiler can simply reuse the same passes as a compiler for the RISC architecture. However, the backend needs to be aware of the explicit bypass network and has to utilize the Special Function Unit (SFU). The selection of pair patterns for generating special instructions is described in Section 4.1. Section 4.2 discusses how the instruction scheduler performs energy-aware scheduling that utilizes the SFU and explicit bypassing network. The remaining parts of code generation are described in Section 4.3.

4.1. Pair Pattern Selection

To use the SFU, the compiler needs to choose pairs of DFG nodes that can be used to generate special instructions, that is, the pair pattern selection. The first step of the pair pattern selection is to find all the node pairs whose patterns are supported by the SFU in the DFG under the constraints described in Section 2.1. A set M containing all these node pairs is obtained by a scan through all nodes in the DFG. Each pair in M is called a *match*. The matches that are unable to meet the operand bypass constraints are excluded from M , for example, the ones with three nonimmediate live-in variables.

The next step is to choose a subset S of M that will be used to generate special instructions. Obviously each DFG node should only be used by one pattern in S , as duplicating DFG nodes only results in extra energy consumption in the pair patterns. A match interference graph $G_I(V, E)$ can be built.

- V is a set of nodes, each representing a possible match in M .
- E is a set of undirected edges between nodes in V . $(u, v) \in E$ means that u and v share a common DFG node. Hence u and v cannot be selected simultaneously.

An example of match interference graph is given in Figure 11: on the left is a DFG with four possible matches; on the right is the match interference graph of the four possible matches. S should be an *independent set* of G_I , that is, the nodes in S are pairwise

Fig. 11. Operation patterns matches.

ALGORITHM 1: Pattern Selection

Input : Match interference graph $M(V, E)$ and pattern frequency F_p
Output : Set of pattern matches S in which nodes do not interfere with each other

```

1  $S \leftarrow \emptyset$ 
2 while  $M \neq \emptyset$  do
3    $D \leftarrow \{d | d \in V, \nexists u \in V : \text{degree}(u) < \text{degree}(d) \text{ and } d \neq u\}$            // Minimum degree
   heuristic
4   if  $|D| = 1$  then
5      $n \leftarrow D[0]$ 
6   else
7      $Q \leftarrow \{q | q \in D, \nexists u \in D : F_p(\text{Pat}(u)) > F_p(\text{Pat}(q))\}$            // Tie-breaking with pattern
     frequency
8     // Pick the first one if  $Q$  has more than one node
9      $n \leftarrow Q[0]$ 
10  end
11   $S \leftarrow S \cup \{n\}$ 
12  Remove  $n$  from  $M$ , along with all its edges and neighbors
13 end

```

nonadjacent in G_I . The objective here is to find as many pairs as possible, which is essentially to get the *Maximum Independent Set* (MIS) of G_I , that is, the independent set with maximum cardinality.

Though finding the MIS of a graph is NP-complete in general, the minimum degree heuristic performs very well for sparse and bounded degree graphs [Halldórsson and Radhakrishnan 1994], which can be implemented in time linear in the number of edges and vertices. In the DFG pair pattern selection, many nodes in the match interference graph have the same degree, which results in many ties in minimum degree selection. Since in the proposed architecture, only a limited number of operation patterns are supported without reconfiguration, the pattern frequency is used to break the ties. The algorithm used for pattern selection is depicted in Algorithm 1. The algorithm yields $\{1, 3\}$ for the example in Figure 11, which is the MIS of the interference graph.

4.2. Energy-Aware Instruction Scheduling

In this work, a list scheduler is used for basic block-level scheduling. In the proposed architecture, the total number of physical registers is reduced as part of the RF address space is used by bypass sources. So although explicit bypass eliminates the need for many temporary registers, it is still very important for the compiler to make sure that register pressure stays low. When the list scheduler greedily chooses the node with maximum number of bypasses, the register pressure may go up. Figure 12 shows

Fig. 12. The effect of bypass scheduler on register pressure.

an example of how a greedy bypass scheduler may increase the register pressure. Here the live ranges of R3 and R4 are longer if the scheduler decides to select the multiply instruction that can use a bypass value. When register pressure is high, the add instruction that uses R3 and R4 should be selected. In this work we use a scheduling algorithm which is similar to the Integrated Prepass Scheduling (IPS) [Goodman and Hsu 1988]. The details of the scheduling algorithm are given in Algorithm 2. Depending on the register pressure of the current partial schedule and the number of available registers in the block, the scheduler switches between two policies.

- When the number of live variables is below the threshold value (number of available registers), choose the node that minimizes energy.
- When the number of live variables is above the threshold value, choose the node that minimizes register pressure.

The procedure **find_node_with_most_energy_gain** in Algorithm 2 tries to select the node with the most energy saving from the ready operation set. In this work, three possible energy-saving scenarios are considered in the node selection.

- Making sure that a special operation can meet the constraints saves one instruction. This leads to energy savings in both the core and memory.
- Operand bypassing (including dead writeback elimination) can reduce the energy consumption of the register file.
- Keeping the same opcode as the previous instruction results in less circuit switching activities in instruction decoder and FU.

Clearly special operation is the most beneficial scenario. The energy saving provided by operand bypassing alone is an order of magnitude less than special operation, because it only reduce the energy for accessing the register file, which consumes much less energy than memory accesses. And keeping the opcode unchanged in most cases results in the least energy saving among the three. So the scheduler determines the priority of the ready operations by the following rules.

- (1) The scheduler first tries to select a special operation node that constraints are guaranteed to be met.
- (2) If it fails to select a unique candidate, or no such special operation node exists, the scheduler chooses the node that benefits most from bypassing.

ALGORITHM 2: Basic Block Scheduling

Input : DFG $G(V, E_d, E_f)$ and register pressure threshold t_r
Output : The schedule of the DFG $T : V \mapsto \mathbb{N}$

```

1  $R \leftarrow \emptyset$  // Ready set
2  $L \leftarrow \emptyset$  // Live variable set
3  $S \leftarrow \emptyset$  // Set of scheduled operations
4  $c \leftarrow 0$  // Cycle counter
5 while  $|S| \neq |V|$  do
6   if  $|L| < t_r$  then
7      $o \leftarrow \text{find\_node\_with\_most\_energy\_gain}(R, T)$ 
8   else
9      $o \leftarrow \text{find\_node\_reduces\_max\_register\_pressure}(R, T, L, G)$ 
10  end
11  for  $s \in \{u | u \in V, (o, u) \in E_d \cup E_f\}$  do
12    if  $s$  is enabled by  $o$  then
13       $R \leftarrow R \cup \{s\}$ 
14    end
15  end
16  for  $p \in \{u | u \in V, (u, o) \in E_d\}$  do
17    if  $o$  is last use of  $p$  then
18       $L \leftarrow L \setminus \{p\}$ 
19    end
20  end
21  if  $o$  has value output then
22     $L \leftarrow L \cup \{o\}$ 
23  end
24   $S \leftarrow S \cup \{o\}$ 
25   $T[o] \leftarrow c$ 
26   $R \leftarrow R \setminus \{o\}$ 
27   $c \leftarrow c + 1$ 
28 end

```

- (3) If there are still multiple candidates, the scheduler chooses the node with the same opcode as the previous operation.
- (4) If all tie breaking does not work, the scheduler chooses the first node it finds.

The proposed algorithm has the same complexity as classical list scheduling, that is, $O(N^2 \log N)$, where N is the number of operations to be scheduled. Compared to the scheduling algorithm and DFG transformation in She et al. [2012a], the proposed scheduling algorithm is able to handle multicycle operations more efficiently.

4.3. Final Code Generation

After the scheduling, a scan through all instructions is performed to check for invalid special instructions, that is, the instructions that do not meet the constraints given in Section 3.3. If a special instruction is found invalid, the checker decomposes it into normal instructions. Due to the nature of explicit bypassing, this transformation does not increase register usage.

The register allocation is done with a graph-coloring algorithm. The register allocation is almost the same as the one used for a normal RISC processor, except that small constant values (ones that can fit in the instruction immediate field) in special instructions are not always encoded into the immediate field when it results in an instruction with two immediate values, which is invalid.

Afterwards the compiler collects pattern information and decides where to insert the reconfiguration codes. In this work, there are two possible scenarios.

- If the number of patterns used in a function is less than or equal to the pattern table size, all patterns are loaded at the entry block of the function.
- If the number of patterns used in a function exceeds the pattern table capacity, the compiler tries to insert reconfiguration code before each intensive loop such that the patterns used in the loop can be put in the table. The loop information can be obtained through static estimation or profiling.

When both ways fail to accommodate all used patterns, the compiler selects the most frequently used patterns. And a special instruction whose pattern is not in the pattern table is decomposed to two normal instructions.

As shown in Figure 10, whenever a code transformation changes the schedule, the bypass status of each instruction needs to be updated, so the validation of the special instructions needs to be repeated. This process terminates: in the worst case, the loop stops when all special instructions are decomposed to normal instructions. In practice, only one or two iterations are sufficient in most cases.

In the proposed design, the scope of the pattern table is within a function. This might introduce overhead across function calls. But the frequently called simple functions usually get in-lined when compiler optimization is enabled. And since the cost of configuring the pattern table is only 10 cycles, we expect the reconfiguration overhead to be negligible in most cases. When there is a function call, the pattern table becomes part of the context, and needs to be treated in a similar way as the general-purpose registers. In the proposed toolflow, the caller is responsible for maintaining the table values. The main reason is that in the proposed flow, the pattern table entries are constant values generated at compile time, so the caller does not need to save the values. If the table is a callee-saved context, the callee needs to actually store the table values to the stack, which results in larger overhead. The results in Section 5 show that the reconfiguration overhead is indeed quite small.

5. EVALUATION AND ANALYSIS

Table IV presents the architectures used in the experiments. The proposed architecture, that is, the one with partially reconfigurable decoder, explicit bypass network, and constrained special instruction patterns (see Section 3.3), is called *SFU-I24*. And the architecture that integrates SFU without the constraints introduced in *SFU-I24* is called *SFU-I32*. The *SFU-I24-C2* is an architecture that is almost identical to *SFU-I24*, except for the two-cycle SFU and interlock logic that help to achieve higher frequency. The datapaths of the baseline, *SFU-I32*, *SFU-I24/SFU-I24-C2* are shown in Figure 4, Figure 5 and Figure 9, respectively. All four cores are implemented in Verilog HDL and synthesized with TSMC 90nm low-power library at 1.2V and typical case. Clock gating is used to minimize dynamic power consumption. The core energy consumption is estimated with the backend information and the real toggle rate generated by post-synthesis simulation. The area and energy consumption of the memory are estimated with CACTI [2009], using 90nm low operating power technology. Table V shows the energy model of the memory used in the experiments.

5.1. Area and Frequency

The implementation results of the four architectures are shown in Table VI. The increase in the core area is understandable and expected, as the SFU, as well as its decoding part, are much more complex compared to simple FUs in RISC. The core area of *SFU-I32* is slightly larger than *SFU-I24* as it needs to support more patterns in the decoder. *SFU-I24-C2* uses slightly more area than *SFU-I24* because of the interlock

Table IV. Configuration of Different Architectures

Architecture	Baseline (Base)	Unconstrained SFU (SFU-I32)	Proposed (SFU-I24)	Proposed w/ 2-cycle SFU (SFU-I24-C2)
Instruction Width	24 bits	32 bits	24 bits	24 bits
Instruction Memory	12kB 24-bit	16kB 32-bit	12kB 24-bit	12kB 24-bit
Data Memory	4k words 16kB 32-bit			
Register File	32b×32 2R1W	32b×32 3R1W	32b×28 2R1W	32b×28 2R1W
SFU Patterns	0	128	8	8
Two-cycle Special Operations	None			Special operations with multiplication

Table V. Memory Energy Consumption

Memory	16kB 32-bit	12kB 24-bit
Energy per access (pJ)	15.38	11.62

Table VI. Implementation Result Comparison

Architecture	Base	SFU-I32	SFU-I24	SFU-I24-C2
Normalized Core Area	1.000	1.309	1.268	1.278
Normalized Memory Area	1.000	1.154	1.000	1.000
Maximum Frequency (MHz)	450	325	325	385

logic for multicycle operations. The difference in memory area between SFU-I32 and SFU-I24 is significant. This is caused by the instruction memory since SFU-I32 uses 32-bit instructions, while SFU-I24 uses 24-bit instructions. In all, the SFU-I32 pays a very high price in terms of area. In contrast, the proposed SFU-I24 realizes the special instruction support with a relatively small overhead. In particular, it does not increase the memory area, which is the dominant part in many modern processors.

The reduced maximum frequency of SFU-I32 and SFU-I24 is mainly caused by the single-cycle SFU, which has two levels of subfunction units. In SFU-I24-C2, this is mitigated by making the special operations that use the multiplier two-cycle operations. And as shown in Table VI, SFU-I24-C2 only pays a small price for the high frequency. Compared to the baseline, there is still a 14.4% loss in frequency, which is primarily caused by the operand switch network in the SFU (see Figure 2).

5.2. Energy Consumption

Table I lists the benchmarks used in the experiments. These kernels are from various application domains. The code for the proposed SFU-I24 and SFU-I24-C2 is generated by the compiler described in Section 4. For SFU-I32, the code generation process is almost the same as SFU-I24, except that all the constraints on operand bypassing and opcode space are removed, and no reconfiguration code is generated. All benchmark programs are compiled with maximum optimization enabled (-O3). Table VII shows the absolute results of the baseline processor. The memory energy in the table includes accesses to both instruction memory and data memory. The energy consumption of each kernel is calculated by multiplying the number of cycles with the average energy (i.e., core + memory) per cycle. In the remainder of this subsection, we normalize all results to the baseline.

Table VII. Results of the Baseline Architecture

Kernel	Simulated Cycles	Average Core Energy per Cycle (pJ)	Average Memory Energy per Cycle (pJ)
Histogram	21547	11.05	16.10
FIR	40973	18.41	16.24
IDCT	2303	17.93	14.56
YUV2RGB	43032	17.88	13.82
MatVec	3729	13.27	14.00
CRC	162017	12.73	11.82
DES	857130	14.89	14.64

Fig. 13. Dynamic cycle count (overhead included).

Fig. 14. Normalized memory energy consumption.

Figure 13 shows the normalized cycle count of the four different cores. Including the overhead of reconfiguration, SFU-I24 achieves a reduction of 25.9%, which is only 2.1% worse than SFU-I32. In some kernels that need to use special operations with multiplication, SFU-I24-C2 uses more cycles. But on average it still reduces 24.8% in cycle count compared to the baseline. When the instruction width is factored in, as shown in Figure 14, the memory energy consumption of SFU-I24 is much less than SFU-I32. Though the number of fetches is reduced dramatically, SFU-I32 only achieves 3.5% average memory energy reduction due to increased instruction width. In 3 out of 7 benchmarks the energy consumption actually goes up. In contrast, the proposed SFU-I24 is able to directly convert the reduction in instruction count into memory energy saving. An average of 21.7% saving is observed. In SFU-I24-C2, a similar result is achieved: the average saving is 21.3%.

Figure 16 shows the normalized core energy consumption. Comparing to the baseline processor, the proposed SFU-I24 reaches a maximal core energy reduction of 21.5% in the FIR case, and of 11.2% on average. The main contributions of energy reduction are

Fig. 15. Normalized number of register file accesses.

Fig. 16. Normalized core energy consumption.

from: (1) reduced RF access energy; (2) reduced datapath and control path overhead due to merged operations.

On the other hand, SFU-I32 increases the average core energy by 1.7%. And it performs very poorly in two cases: FIR and IDCT, in which the core energy increases by over 8%. The explicit bypass network is an important contributing factor in this huge difference. As shown in Figure 15, the number of accesses to the RF in SFU-I24 is significantly reduced. In addition, the RF in SFU-I24 has fewer ports than the one in SFU-I32. As a result, the core of SFU-I24 consumes much less energy compared to SFU-I32, for which in both FIR and IDCT, a degradation of over 5% is observed.

In the case of SFU-I24-C2, the core energy consumption increases when a lot of special instructions with multiplication are used. The main reason is that the compiler is forced to use a less energy-efficient schedule in order to fill the delay caused by multicycle operations. When no special instructions with multiplication are used, the result is similar to SFU-I24. On average, the core energy is reduced by 5.8% compared to the baseline.

Figure 17 shows the normalized total energy consumption. The proposed SFU-I24 reduces both the memory and core energy, and it achieves an average saving of 15.8%. It reaches a maximal of 33.1% energy saving in CRC. In SFU-I24-C2, the total energy is reduced by an average of 13.1%, and the maximal saving occurs in CRC, which is 32.2%. By contrast, in the case of SFU-I32, the total energy saving is only 1.1%. The breakdown of the energy in different kernels is shown in Figure 18, which clearly shows that SFU-I24 and SFU-I24-C2 outperform SFU-I32 in both core and memory energy in most cases.

These results show that although the use of SFU is able to significantly reduce the dynamic cycle count, directly putting the SFU into a generic processor without any constraint does not result in an energy-efficient architecture.

Fig. 17. Normalized total energy consumption.

Fig. 18. Energy gain breakdown.

Fig. 19. Normalized execution time (adjusted according to max. frequency.)

5.3. Performance

The normalized execution time of the different cores used in the experiments is shown in Figure 19. The result is calculated based on the dynamic cycle count and the maximal frequency of each core. Due to the loss in frequency, both SFU-I32 and SFU-I24 suffer minor performance degradation, though they are able to reduce the cycle count by about 25%. However, in SFU-I24-C2, an average speedup of 13.8% is observed. The SFU-I24-C2 is able to achieve a good balance between performance and energy consumption, with relatively small overhead compared to SFU-I24.

The proposed architecture with a partially reconfigurable decoder and an explicit bypass network is able to reach a balance between the energy efficiency and the flexibility of the SFU, and it results in an architecture with high energy efficiency and good performance.

6. RELATED WORK

The use of complex operation patterns, called Instruction Set Extension (ISE), is common in instruction set synthesis for application-specific and reconfigurable processor design [Karuri et al. 2009; Clark et al. 2003; Leupers et al. 2006]. A comprehensive survey that covers most recent works in ISE could be found in Galuzzi and Bertels [2011]. The idea of using a dynamically reconfigurable decoder can be found in some early high-performance architectures, in the form of programmable microcode [Razdan and Smith 1994]. Reconfigurable architectures like Montium [Heysters et al. 2003] and MOLEN [Vassiliadis et al. 2004] exploit reconfigurable decoders to support flexible ISAs, but they are not tightly integrated into a general-purpose processor. The toolchain of Conservation Cores uses automatically synthesized accelerators called *c-cores* to improve energy efficiency in many core architectures [Venkatesh et al. 2010]. The ConCISE toolchain proposed by Kastrup et al. [1999] introduces an accelerator called the Reconfigurable Function Unit (RFU) based on a programmable logic device (CPLD/FPGA). The RFU is integrated into an MIPS datapath as a new FU [Kastrup et al. 1999]. Similar ideas could be found in the ADRES reconfigurable processor [Mei et al. 2003], the Stretch software configurable processor [Stretch 2013; Gonzalez 2006], the Rotating Instruction Set Processing Platform (RISPP) [Bauer et al. 2007], and the rASIP [Karuri et al. 2008] integrate coarse-grained reconfigurable components into the datapath of baseline processors. Huynh et al. proposed dynamic instruction set configuration for a flexible reconfigurable custom instruction unit, addressing the trade-offs between area, performance, and reconfiguration cost [Huynh et al. 2007]. In the aforementioned works, relative expensive reconfiguration is usually required and the cost to support flexible ISE is high compared to this work. The FITS framework [Cheng et al. 2004] explored instruction set tuning to optimize instruction encoding and operand bandwidth in embedded application-specific processors. The FITS framework tackled similar problems as this work, but it put more focus on how to tailor processors for specific applications.

There are also studies trying to integrate ISE in general-purpose architectures [Clark et al. 2004, 2005; Jayaseelan et al. 2006]. Most of these focus on improving the performance. Clark et al. proposed integration of a Configurable Compute Accelerator (CCA) into a general-purpose processor [Clark et al. 2004, 2005]. Compared to the SFU in this article, the CCA is relatively complex and it requires up to 4 inputs and 2 outputs, as its main objective is improving performance. The control part of CCA is designed to be transparent so that the code can be executed with or without CCA. Woh et al. proposed AnySP, a wide SIMD signal processor targeting wireless and multimedia applications [Woh et al. 2009]. In AnySP the idea of operation pairs is similar to the SFU design in this work, but the operand bandwidth problem is partially solved by introducing an extra small RF. Venkatesh et al. proposed QsCores, a framework that automatically synthesizes an accelerator for a wide range of applications from source code [Venkatesh et al. 2011]. In PEPSC, an architecture designed for efficient scientific computing, Dasika et al. proposed an FPU that is capable of executing up to five back-to-back operations [Dasika et al. 2011]. In this work we exploited the locality of the special operation patterns in designing a partially reconfigurable decoder to achieve energy-efficient integration of SFU into a RISC processor with compact ISA, which allowed the proposed architecture to improve energy efficiency substantially in different application domains.

The data bandwidth from the register file to the FUs is an important constraint in ISE design [Atasu et al. 2003]. Leupers et al. introduced a special register file called *Internal Registers* (IR) for the special instruction units [Leupers et al. 2006]. The IR is an effective way of implementing application-specific special instruction, but it

lacks flexibility and complicates the code generation as the registers and FUs are no longer orthogonal, that is, an FU cannot access arbitrary registers. Karuri et al. proposed RF clustering in a single-issue processor to mitigate the register file port pressure in ISE in ASIP design [Karuri et al. 2007]. While reducing port pressure, the RF clustering, which is similar to what is used in clustered VLIW architectures, also makes the code generation much more complex. Pozzi and Ienne exploited the fact that pipelined SFUs do not need all operands in the same cycle to distribute register file accesses across multiple cycles [Pozzi and Ienne 2005]. This cannot be applied to the SFUs that are similar to the one used in this work. Utilizing the bypass network has been proven an efficient way to increase operand bandwidth and reduce register file energy in different types of architectures [She et al. 2012b; Balfour et al. 2007, 2009; Park et al. 2006]. Jayaseelan et al. proposed explicit forwarding to reduce register file port pressure and operand encoding cost for application-specific ISE in a RISC-like datapath, which resembles the idea of explicit bypass in this work [Jayaseelan et al. 2006]. However, the power model used in Jayaseelan et al. [2006] only considers the consumption of the register file. The overall energy efficiency of the proposed architecture is not clear. Cong et al. proposed shadow registers to solve the operand bandwidth issue for supporting special instructions in a configurable processor [Cong et al. 2005]. The shadow registers are similar to explicit pipeline registers, but have more flexibility. To avoid dramatic increase of control bits, the shadow registers are hash-mapped, which may be less efficient in terms of energy. In this article, we explored the trade-offs in utilizing a bypass network for energy-efficient ISE in a generic processor with compact ISA, along with detailed and realistic results. The proposed solution achieved high energy efficiency while maintaining the generality of the baseline architecture.

Code generation is a key in supporting ISE for both application-specific and general-purpose processors. Selection and scheduling for special instructions is one of the core aspects in code generation for ASIP and reconfigurable architectures. There are quite a few works on identifying and synthesizing special instructions for ASIPs [Atasu et al. 2003, 2012; Yu and Mitra 2004b]. In this work, the special instruction generation problem is limited to generating operation pairs. So in this work we focus on selecting those patterns that can meet the constraints and fit in the pattern table. Scheduling algorithms that are aware of the bypassing is another key element for generating code for the proposed architecture. Park et al. presented a greedy algorithm for increasing the bypassing in a RISC processor [Park et al. 2006]. For architectures that fully expose the bypass network to software, like TTA and STA, bypass-aware scheduling can achieve significant reduction in RF traffic, thereby improving performance and energy efficiency [Guo et al. 2006; She et al. 2012b]. But the fine-grained control over the datapath also introduces overhead in code density, resulting in increased fetching and decoding energy. In Jayaseelan et al. [2006], Integer Linear Programming (ILP) is used to perform bypass-aware scheduling in a processor with application-specific ISE. The proposed algorithm inserts register copying instructions in order to meet the constraints of the special instructions. In this work, we proposed a modified list scheduling algorithm, in which the priority calculation takes into account the energy impact of special instructions as well as explicit bypassing. The results show that the proposed algorithm is effective.

In this article, we introduced a novel architecture that uses special instructions. In contrast to aforesaid works, this work aims at improving the energy efficiency of a generic processor with a compact ISA. Two major issues: (i) opcode and operand encoding, and (ii) operand bandwidth to SFU are solved by using a partially reconfigurable decoder and explicit bypass network.

7. CONCLUSIONS AND FUTURE WORK

Integrating a Special Function Unit (SFU) that executes complex operations into a generic processor for energy efficiency is not easy, as special instructions may incur large overhead, especially when the ISA is a compact one. This article introduced an architecture for integrating SFU that supports flexible operation pair patterns in a generic processor with a compact ISA. A partially reconfigurable decoder and a software-controlled explicit bypass network are used to: (i) encode extra opcodes and operands in the limited instruction coding space; (ii) supply sufficient data to the special instructions without increasing the number of register file ports. We presented a compiler backend design for the proposed architecture. The compiler is able to utilize the SFU and the explicit bypass network to generate energy-efficient code. Results including benchmarks from different domains demonstrate that the proposed architecture and compiler are effective: average dynamic cycle count is reduced by over 25%. The total processor energy consumption is reduced by 15.8%. When high performance is required, the proposed architecture is able to achieve a speedup of 13.8% with 12.6% energy reduction compared to the baseline, by introducing multicycle SFU operations.

Future work includes supporting more complex patterns in the SFU, and exploring the further trade-offs between the complexity of the SFU and the energy efficiency of the processor architecture. Further exploration of operation pattern locality and optimization of the reconfiguration is interesting, especially for control-intensive programs.

REFERENCES

- ARM LTD. 2013. ARM thumb instruction set. <http://www.arm.com/>.
- ARNOLD, M. AND CORPORAAL, H. 1999. Automatic detection of recurring operation patterns. In *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES'99)*. ACM Press, New York, 22–26.
- ATASU, K., POZZI, L., AND IENNE, P. 2003. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of the 40th Annual Design Automation Conference (DAC'03)*. ACM Press, New York, 256–261.
- ATASU, K., LUK, W., MENCER, O., OZTURAN, C., AND DUNDAR, G. 2012. Fish: Fast instruction synthesis for custom processors. *IEEE Trans. Very Large Scale Integr. Syst.* 20, 1, 52–65.
- BALFOUR, J., DALLY, W., BLACK-SCHAFER, D., PARIKH, V., AND PARK, J. 2007. An energy-efficient processor architecture for embedded systems. *Comput. Archit. Lett.* 7, 1, 29–32.
- BALFOUR, J., HARTING, R., AND DALLY, W. 2009. Operand registers and explicit operand forwarding. *Comput. Archit. Lett.* 8, 2, 60–63.
- BAUER, L., SHAFIQUE, M., KRAMER, S., AND HENKEL, J. 2007. Rispp: Rotating instruction set processing platform. In *Proceedings of the 44th Annual Design Automation Conference (DAC'07)*. ACM Press, New York, 791–796.
- CACTI. 2009. cacti 5.3, rev 174. <http://quid.hpl.hp.com:9081/cacti/>.
- CHENG, A., TYSON, G., AND MUDGE, T. 2004. Fits: Framework-based instruction-set tuning synthesis for embedded application specific processors. In *Proceedings of the 41st Annual Design Automation Conference (DAC'04)*. ACM Press, New York, 920–923.
- CLARK, N., ZHONG, H., AND MAHLKE, S. 2003. Processor acceleration through automated instruction set customization. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03)*. IEEE Computer Society, 129–140.
- CLARK, N., KUDLUR, M., PARK, H., MAHLKE, S., AND FLAUTNER, K. 2004. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'04)*. IEEE, 30–40.
- CLARK, N., BLOME, J., CHU, M., MAHLKE, S., BILES, S., AND FLAUTNER, K. 2005. An architecture framework for transparent instruction set customization in embedded processors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*. IEEE, 272–283.
- CONG, J., HAN, G., AND ZHANG, Z. 2005. Architecture and compilation for data bandwidth improvement in configurable embedded processors. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'05)*. IEEE, 263–270.

- DASIKA, G., SETHIA, A., MUDGE, T., AND MAHIKE, S. 2011. PEPSC: A power-efficient processor for scientific computing. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. IEEE, 101–110.
- GALUZZI, C. AND BERTELS, K. 2011. The instruction-set extension problem: A survey. *ACM Trans. Reconfig. Technol. Syst.* 4, 2, 18:1–18:28.
- GONZALEZ, R. 2006. A software-configurable processor architecture. *IEEE Micro* 26, 5, 42–51.
- GOODMAN, J. R. AND HSU, W. 1988. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 2nd International Conference on Supercomputing (ICS'88)*. ACM Press, New York, 442–452.
- GUO, J., LIMBERG, T., MATUS, E., MENNENGA, B., KLEMM, R., AND FETTWEIS, G. 2006. Code generation for sta architecture. In *Proceedings of the 12th International Conference on Parallel Processing (Euro-Par'06)*. Springer, 299–310.
- HALLDORSSON, M. AND RADHAKRISHNAN, J. 1994. Greed is good: Approximating independent sets in sparse and bounded-degree graphs. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing (STOC'94)*. ACM Press, New York, 439–448.
- HEYSTERS, P. M., SMIT, G. J. M., AND MOLENKAMP, E. 2003. Montium - Balancing between energy-efficiency, flexibility and performance. In *Engineering of Reconfigurable Systems and Algorithms*. CSREA Press, 235–241.
- HUYNH, H. P., SIM, J. E., AND MITRA, T. 2007. An efficient framework for dynamic reconfiguration of instruction-set customization. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'07)*. ACM Press, New York, 135–144.
- JAYASEELAN, R., LIU, H., AND MITRA, T. 2006. Exploiting forwarding to improve data bandwidth of instruction-set extensions. In *Proceedings of the 43rd Annual Design Automation Conference (DAC'06)*. ACM Press, New York, 43–48.
- KARURI, K., CHATTOPADHYAY, A., HOHENAUER, M., LEUPRES, R., ASCHEID, G., AND MEYR, H. 2007. Increasing data-bandwidth to instruction-set extensions through register clustering. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'07)*. IEEE, 166–171.
- KARURI, K., CHATTOPADHYAY, A., CHEN, X., KAMMLER, D., HAO, L., ET AL. 2008. A design flow for architecture exploration and implementation of partially reconfigurable processors. *IEEE Trans. Very Large Scale Integr. Syst.* 16, 10, 1281–1294.
- KARURI, K., LEUPERS, R., ASCHEID, G., AND MEYR, H. 2009. A generic design flow for application specific processor customization through instruction-set extensions (ises). In *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'09)*. 204–214.
- KASTRUP, B., BINK, A., AND HOOGERBRUGGE, J. 1999. ConCISE: A compiler-driven cpld-based instruction set accelerator. In *Proceedings of the 7th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'99)*. IEEE, 92–101.
- LATTNER, C. AND ADVE, V. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. 75–86.
- LEIBSON, S. 2006. *Designing SOCs with Configured Cores: Unleashing the Tensilica Xtensa and Diamond Cores (Systems on Silicon)*. Morgan Kaufmann, San Francisco, CA.
- LEUPERS, R., KARURI, K., KRAEMER, S., AND PANDEY, M. 2006. A design flow for configurable embedded processors based on optimized instruction set extension synthesis. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'06)*. 581–586.
- MEI, B., VERNALDE, S., VERKEST, D., DE MAN, H., AND LAUWEREINS, R. 2003. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *Proceedings of the 13th International Conference on Field Programmable Logic and Application*. Lecture Notes in Computer Science, vol. 2778. Springer, 61–70.
- OPENCORES. 2013. OpenRISC 1200. <http://opencores.org/openrisc>.
- PARK, S., SHRIVASTAVA, A., DUTT, N., NICOLAU, A., PAK, Y., AND EARLIE, E. 2006. Bypass aware instruction scheduling for register file power reduction. In *Proceedings of the ACM Conference on Language, Compilers, and Tool Support for Embedded Systems (LCTES'06)*. ACM Press, New York, 173–181.
- POZZI, L. AND IENNE, P. 2005. Exploiting pipelining to relax register-file port constraints of instruction set extensions. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'05)*. ACM Press, New York, 2–10.
- RAZDAN, R. AND SMITH, M. 1994. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO'94)*. ACM Press, New York, 172–180.

- SHE, D., HE, Y., AND CORPORAAL, H. 2012a. Energy efficient special instruction support in an embedded processor with compact isa. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'12)*. ACM Press, New York.
- SHE, D., HE, Y., MESMAN, B., AND CORPORAAL, H. 2012b. Scheduling for register file energy minimization in explicit datapath architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'12)*. 388–393.
- STRETCH, INC. 2013. Stretch software configurable processors. <http://www.stretchinc.com/>.
- TENSILICA, INC. 2013. Xtensa customizable processors. <http://www.tensilica.com/>.
- VASSILIADIS, S., WONG, S., GAYDADJIEV, G., BERTELS, K., KUZMANOV, G., AND PANAINTE, E. M. 2004. The molen polymorphic processor. *IEEE Trans. Comput.* 53, 11, 1363–1375.
- VENKATESH, G., SAMPSON, J., GOULDING-HOTTA, N., VENKATA, S. K., TAYLOR, M. B., ET AL. 2011. QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*. ACM Press, New York, 163–174.
- VENKATESH, G., SAMPSON, J., GOULDING, N., GARCIA, S., BRYKSIN, V., ET AL. 2010. Conservation cores: Reducing the energy of mature computations. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*. ACM Press, New York, 205–218.
- WOH, M., SEO, S., MAHLKE, S., MUDGE, T., CHAKRABARTI, C., AND FLAUTNER, K. 2009. AnySP: Anytime anywhere anyway signal processing. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. 128–139.
- YU, P. AND MITRA, T. 2004a. Characterizing embedded applications for instruction-set extensible processors. In *Proceedings of the 41st Annual Design Automation Conference (DAC'04)*. ACM Press, New York, 723–728.
- YU, P. AND MITRA, T. 2004b. Scalable custom instructions identification for instruction-set extensible processors. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'04)*. ACM Press, New York, 69–78.

Received December 2012; revised February 2013; accepted April 2013