A Just-in-Time Customizable Processor

Liang Chen*, Joseph Tarango†, Tulika Mitra*, Philip Brisk†
*School of Computing, National University of Singapore
†Department of Computer Science and Engineering, University of California, Riverside
Email: {chenliang,tulika}@comp.nus.edu.sg, {jtarango,philip}@cs.ucr.edu

***Abstract*—A *traditional extensible processor* with customized circuits achieves high performance at the cost of flexibility, while a *dynamically extensible processor* with reconfigurable fabric offers flexibility for instruction-set extensions (ISEs) but suffers from computational inefficiency. We introduce a novel architecture called *Just-in-Time Customizable (JiTC)* processor that reconciles the conflicting demands of performance and flexibility in extensible processors. Our key innovation is a multi-stage accelerator, called *Specialized Functional Unit (SFU)*, that is tightly integrated in the processor pipeline. The SFU design is derived through a systematic study of a large range of representative embedded applications. The SFU can be reconfigured on per-cycle basis to support different application-specific instructions at near-ideal performance of an extensible processor. We also provide an automated compilation tool chain for JiTC processor. The experimental results confirm the efficiency and applicability of our approach.**

I. Introduction

An Application-Specific Instruction-Set Processors (ASIP) with highly optimized Instruction-Set Architecture (ISA) can offer high-performance for the specific application domain. However, designing an ASIP requires long turn-around time. *Extensible processors* have emerged to be promising alternatives to ASIPs. An extensible processor typically consists of a base processor and a set of *instruction set extension (ISE)* (also referred as *custom instruction* in literature) components. These ISE components help to extend the base ISA through customization or reconfiguration according to the specific applications. In *traditional extensible processors*, designers statically customize the processor by adding application-specific ISEs, using customized circuits tightly integrated with the processor pipeline. These traditional extensible processors provide high performance and energy efficiency; however, they lack flexibility as they are highly specialized cores restricted to accelerating specific applications. To increase flexibility, ISEs can be synthesized on a reconfigurable fabric that has been integrated into the processor forming a *dynamically extensible processor*. With the provided reconfigurability, the application-specific instruction set customization can be done in the post-fabrication phase. On the other hand, the cost of reconfiguration can be quite high, and the ideal choice of the fabric, whether fine-grained or coarse-grained remains an open question.

The objective of this paper is to *design an extensible processor that combines customized circuit-like efficiency and reconfigurable logic-like flexibility in the implementation of application-specific instruction-set extensions.* To achieve this goal, we first derive a near-ideal reconfigurable accelerator based on an empirical analysis of a set of representative embedded applications, and then develop architectural mechanisms to reduce the reconfiguration overhead through integration with the processor's pipeline. The application analysis classifies the commonly occurring sequences of arithmetic and logical operations, which are essentially candidates for selection as ISEs. To support these sequences, we identify several smaller functional units that can be combined to form a reconfigurable multi-stage ALU (arithmetic logic unit). This accelerator, which we call a *Specialized Functional Unit (SFU)*, can execute ISEs from a variety of applications. The SFU is integrated into the processor pipeline in parallel with the ALU, and is accessed using a non-traditional instruction fetch and decode mechanism, which we call a *Just-in-Time Customizable (JiTC)* core. When an opcode matching an accelerator function is read from the instruction cache, the traditional fetch-and-decode mechanism is suppressed, enabling execution of an ISE on the SFU.

978-1-4799-1071-7/13/$31.00 © *2013 IEEE*

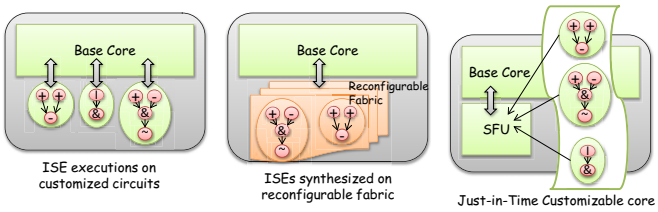


Fig. 1: *Different customizable processor design approaches*

Figure 1 illustrates the difference between the JiTC core and prior approaches to provide application-specific ISE functionality. On the left, a traditional extensible processor is formed by augmenting a base processor core with ISEs executed in customized circuits; this approach is inflexible and cannot accelerate other applications or adapt to post-fabrication software evolution. In the center, reconfigurability is added by accommodating ISEs on a reconfigurable fabric. This creates a dynamically extensible processor, which provides high flexibility, but suffers from many inefficiencies, especially when the cost of programmable interconnect is taken into account. The JiTC core is shown on the right, where ISEs execute on the SFU, as opposed to a reconfigurable fabric. The SFU is essentially a small collection of parallel multi-stage ALUs with different functionalities and operation sequences; thus, it is smaller and more efficient than a reconfigurable fabric. More importantly, it also provides reconfigurability and requires far less reconfiguration overhead. Comparing to traditional extensible processors, the JiTC core is more area-efficient. In JiTC core, the SFU is time-multiplexed to execute multiple ISEs, while all ISE circuits are customized individually and independently in a traditional extensible processor.

In the following, we present the design of the SFU and the JiTC core that allows tight integration of the SFU in the pipeline datapath. Our synthesis results at 45nm technology show that the SFU can operate at 606 MHz frequency and requires only $0.08mm^2$ area. We also develop an automated compilation tool chain that can generate executables to exploit the SFU and a cycle-accurate simulator that accurately models the JiTC core. The experimental results confirm that the JiTC core can effectively provide acceleration through ISEs across a wide range of applications.

II. Related work

This work is most closely related to projects that integrate accelerators into processor pipelines. The best performance can be achieved through traditional extensible processors where the acceler-

ator containing ISEs is implemented in customzied circuits as shown in [2], [3], [27], [18], [37], [38]. But these traditional extensible processors suffer from limited flexibility and accelerating multiple applications on the same architecture calls for reconfigurability. Dynamically extensible processors, on the other hand, can have reconfigurable accelerators to accommodate the ISEs with tradeoffs between flexibility and performance. One historic debate involves the granularity of the accelerator: should it be fine-grained, similar to an FPGA [10], [17], [33], [36], should it be coarse-grained, i.e., an array of ALUs with a programmable interconnect [11], [12], [14], or should the granularity be even coarser at the level of expressions [1]. This work creates what is essentially an expression-grained ALU similar to the Expression Grained Reconfigurable Array (EGRA) [7] but much smaller and supporting multi-stage execution.

Stojilovic et al. [32] created an application-specific CGRA sharing some similarities in operation fusion with ours; they identified commonly occurring operation sequences in ISEs that were identified using established algorithms [27]; they then merge the sequences to create a common super-sequence, which is treated as a sequential 1D array; they then replicate the column to create a 2D array and add a bus-based FPGA-style interconnect [35] to provide routing between the rows. Our approach extracts operation sequences and applies merging optimizations in form of regular expressions. Rather than the duplications, we take advantage of reconfigurations with support from control memory. In terms of functional unit fusion, our approach also shares similarities with CGRA express [26], but our ALU is smaller and we sidestep the overhead of programmable interconnect.

Interlock collapsing ALU [34] was proposed to parallelize the execution of up to two interlocked consecutive instructions. In contrast, we realize fusion using bypass line. [36] can collapse multiple instructions with up to 10 inputs and multiple outputs dynamically using special functional units, which are based on FPGA-like elements requiring large number of control bits and longer latencies. [12] also uses dynamic approaches with coarse-grained programmable accelerators. Dynamic Strands [29] collapses sequential instructions using close-loop ALUs where the output of an ALU is forwarded to its inputs using self-bypass lines. We identify the ISE statically, thereby avoiding potentially expensive identification hardware, but require extra control memory to hold the control signals.

Dataflow mini-graphs [8], [9] and static strands [28] are static approaches requiring compiler support. However, our compiler can identify additional patterns apart from mini-graphs and strands, potentially leading to higher speedup. We systematically design the SFU to execute the patterns rather than simply adding self-bypass lines in ALUs [28] or using ALU pipeline [8], [9]. Our approach bears some similarity to DySER [14], which controls a coarse-grained accelerator using a long instruction word. We benefit from storing the configuration in a control memory and reserving only few opcode in the instruction set.

Additional works have focused on dynamic reconfiguration of programmable accelerators, including: MorphoSys [30], an array designed specifically to facilitate dynamic reconfiguration, RISPP [6], which dynamically reconfigures selected columns of an FPGA that implement custom accelerator functions; Warp processor [23], which transparently converts software binaries to placed-and-routed FPGA bitstreams; and KAHRISMA [19], a hybrid fine-grained/coarse-grained accelerator. Rather than explicitly reconfiguring our SFU, we store control signals for multi-cycle operations in on-chip memory and execute a set of statically determined ISEs.

III. Analysis of Instruction-Set Extensions

The SFU is designed to accelerate commonly occurring expressions, which are encapsulated as ISEs. We first analyze the ISEs found across a range of applications to identify the characteristics that lead to performance acceleration. Section VII describes the experimental setup used for this analysis.

Figure 2 shows the dataflow graph (DFG) representing an ISE with four inputs, two outputs, and six arithmetic and logical operations. The ISE obtains speedup by either exploiting *instruction-level parallelism (ILP)*, or chaining consecutive operations in a single-cycle. The out-of-order processors with dynamic instruction scheduling can extract ILP automatically, but with high area overhead and energy consumption; alternatively, a compiler can extract ILP and schedule operations statically as in a VLIW (Very Long Instruction Word) architecture. Operation chaining, in contrast, depends on the frequency of the processor; for the DFG in Figure 2, a multiply-accumulator could execute the multiply-add portion of the ISE in one cycle, while a chain of arithmetic and logic operators could execute the shift and logical-AND operations in a second cycle. The fundamental question that we ask and answer in this section is whether parallelism or operation chaining has a stronger correlation with the speedup obtained by an ISE.

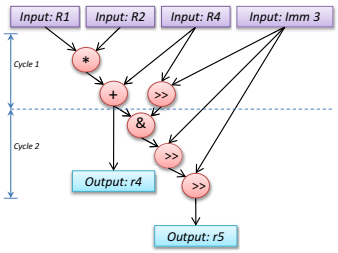


Fig. 2: *Dataflow Graph (DFG) of an ISE*

A. Exploring Inter-Operation Parallelism

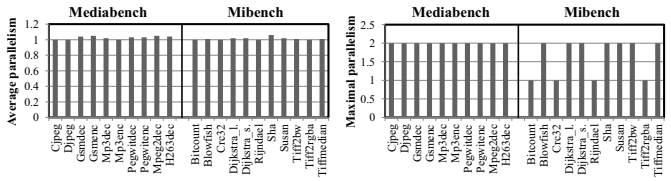
ILP extraction and exploitation are fundamental to computer architecture research. In terms of ISE identification, increasing the I/O bandwidth to/from an ISE leads to wider dataflow graphs with higher ILP [2], [3], [4], [37]; however, *prior work [11], [12], [37] has reported that up to four inputs and two outputs are sufficient to achieve near-optimal speedup for ISEs in most cases*. Therefore, we study ISEs with at most four input and two output operands in this work. The average parallelism of an ISE is defined as

$$avarage\ parallelism = \frac{\#\ of\ total\ operations}{critical\ path\ length}$$

where the critical path length is the number of operations along the longest path in the DFG. For example, in Figure 2, the critical path length is 5 and the average parallelism is 6/5 = 1.2. The average parallelism captures the ILP available within ISEs, i.e., the average number of operations that can execute in parallel per cycle. The maximal parallelism is the maximum number of DFG operations executing concurrently using the *As Late As Possible (ALAP)* scheduling policy. For example, in Figure 2, with an ALAP scheduling, the ADD operation executes in parallel with the first shift, so the maximal parallelism for the ISE is 2.

We analyzed the average and maximal parallelism of ISEs found in 21 Mibench and Mediabench applications. The average parallelism is close to 1, and the maximum parallelism never exceeds 2, across all of the applications as shown in Figure 3. This confirms that ISEs with

up to 4 inputs and 2 outputs have limited ILP, and at most two parallel functional units should suffice to exploit the limited parallelism.



(a) Average parallelism (b) Maximal parallelism

Fig. 3: *Parallelism explorations for Mediabench and Mibench benchmark suites*

B. Exploring Critical Path Length

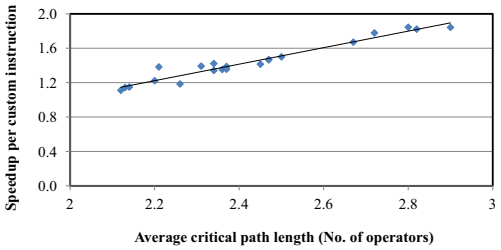


Fig. 4: *Correlation between critical path length & speedup*

Impact of operation chaining on speedup: To investigate the impact of operation chaining on the speedup of ISE, we measure and report the ISE's average critical path length. This metric is closely related to the number of dependent operators that could be chained and executed in one cycle. Figure 4 shows the correlation between the average critical path length and the average speedup per ISE. Each point in the graph corresponds to one particular application from the set of 21 Mibench and Mediabench applications. The linear trend line establishes a linear correlation between the two variables. Thus, *ISEs with a longer critical path tend to achieve the highest speedups*.

Hot sequences in operation chaining: Our objective is to design an SFU that exploits operation chaining; however, the SFU cannot possibly support every possible operation sequence. Thus the first step is to identify "hot" sequences, i.e., those that appear frequently in ISEs across a wide variety of applications. The primary objective of the SFU is to execute the hot sequences efficiently.

First, we classify the operations into five groups: arithmetic operations (A type), logical operations (L type), shift (S type), wire (W type), and multiplication operations (M type). W type operations are essentially move instructions that are converted to wiring when synthesized as part of an ISE. All operations belonging to the same class have approximately equal latencies, and can be implemented using a single physical execution block. A sequence is hot if it occurs above a certain threshold; we use a threshold value of 5%, which captures the most frequent sequences. We restrict the number of operators per sequence to 3, as the average critical path length does not exceed 3 (see Figure 4).

Figure 5 shows the hot sequences from an analysis of the 21 Mibench and Mediabench applications. While not all possible operator chain combinations are possible, only a handful of operator chains (colored in red) appear frequently: there are five hot two-operator sequences and six hot three-operator sequences. The frequency of occurrence of each sequence is averaged across the benchmarks; certain sequences may occur frequently in some, but not all, of the applications. For example, the sequence *MWA* has a frequency of 66% in the `Tiff2bw` application; however it only has 5% frequency averaged across all benchmarks.

In this section, we confirm that *under the specified I/O constraint, there is limited parallelism available in ISEs and only few sequences are hot across various applications*. The analysis results will be used as design guidelines in Section IV.

IV. SFU Design

The SFU is designed to support the 11 hot sequences that appear frequently in ISEs across the 21 MiBench and MediaBench applications that we analyzed in the preceding section. The hot sequences are: *AA, AS, LL, SA, SL, ASA, LLS, LSA, SAS, MWA, WMW*. The SFU is designed to execute each of these hot sequences in one clock cycle. We build up the SFU incrementally, starting from a basic functional unit, described next.

Basic functional unit (BFU): Figure 6(a) illustrates a basic functional unit, which is an ALU followed by a shifter. The BFU supports five operation sequences: *A*, *L*, *S*, *AS*, and *LS*. To support sequences *A* and *L*, the shift length is set to zero; to support sequence *S*, the ALU performs an identity operation (e.g., OR identical inputs) on the input operand. A regular expression that enumerates all possible sequences supported by the basic functional unit is $(A \mid L \mid \varepsilon)(S \mid \varepsilon)$ where ε is the empty string.

Fused basic functional units: The basic functional unit can only support one (*AS*) out of eleven hot sequences. The fused basic functional unit chains two basic functional units sequentially along with a bypass line, as shown in Figure 6(b). Using regular expression notation, the fused basic functional unit supports sequences $((A \mid L \mid \varepsilon)(S \mid \varepsilon))^2$, which encompasses all hot sequences that do not include a multiplier.

Complex functional unit: The two remaining sequences are *MWA*, *WMW*, where *W* selects the upper- or lower 32-bit portions of the 64-bit output of a 32×32-bit multiplier; this is done using multiplexers and control signals. The remaining subsequence, $M(A \mid \varepsilon)$, is a fused multiply-addition operation that can be implemented using a MAC unit. We refer to this operator as a complex functional unit, as shown in Figure 6(c). The rationale for including the ALU and the shifter in the complex functional unit is to provide additional parallelism in the SFU as explained next.

Component	Area(μm^2)	delay(ns)
Basic Functional Unit	9856.7078	0.7231
Fused Basic Functional Unit	27913.4943	1.5424
Complex Functional Unit	49780.5275	1.6379
SFU	80502.7823	1.6499

TABLE I: *Area and delay for the SFU components*

Specialized functional unit (SFU): We synthesized the different components of the SFU in Synopsys Design Compiler with PDK 45nm standard cell library. Table I provides the area and delay values for the components. We observe that the fused basic functional unit in Figure 6(b) has a latency of 0.7231ns while the complex functional unit supporting multiplication in Figure 6(c) has a latency of 1.6379ns. Therefore, we form the SFU by placing a complex functional unit in parallel with a fused basic functional unit, without extending the critical path. This dual-path architecture supports the maximum parallelism of 2 that is present in the ISEs. Functional units within the SFU are fully connected, and each internal functional unit has access to four input and two output registers. Fig. 6(d) shows the SFU architecture, which supports single-cycle execution of all eleven hot sequences at 606MHz clock frequency and $0.08mm^2$ area.

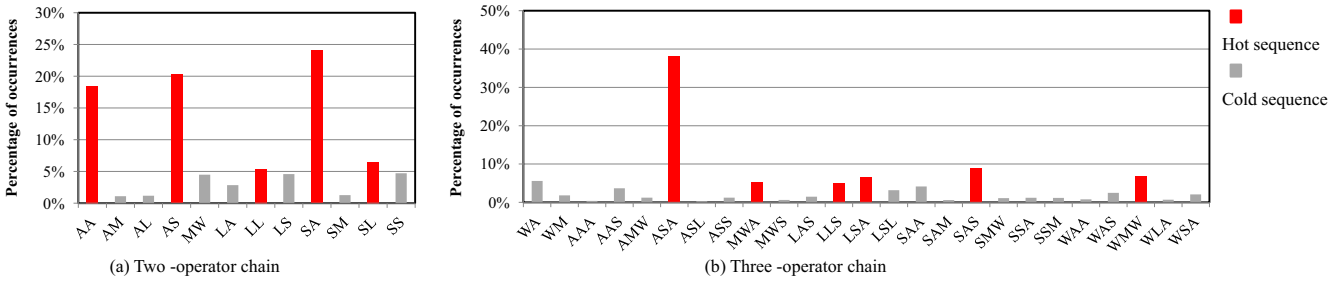


Fig. 5: *Hot sequences in operation chaining*

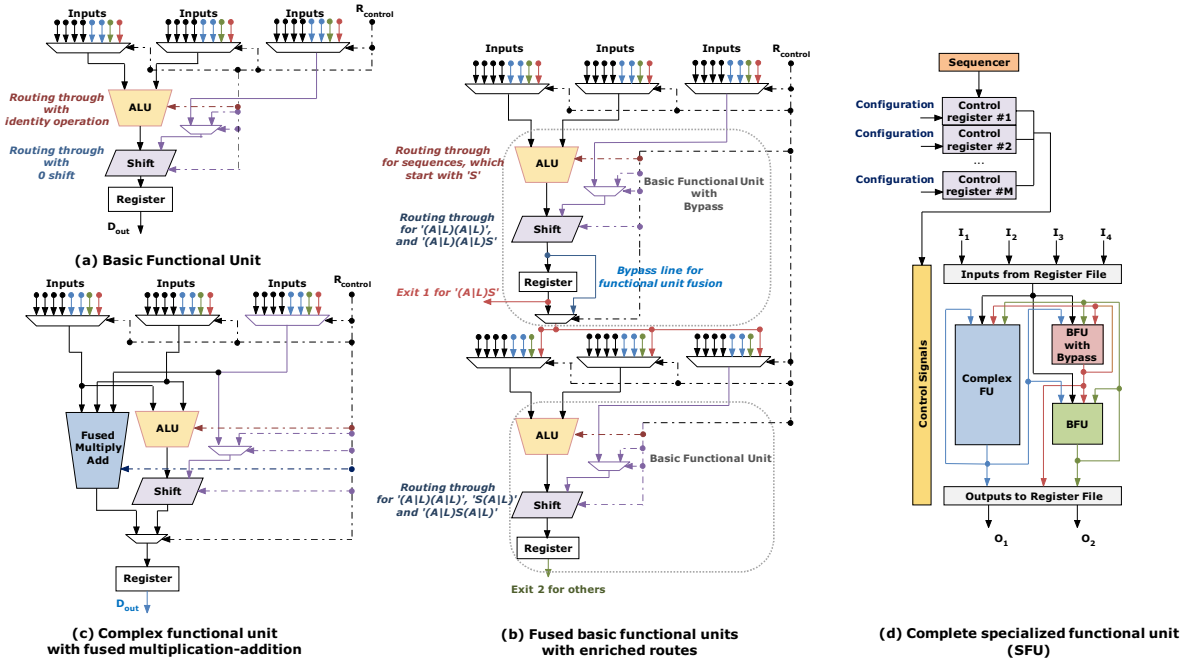


Fig. 6: *Design of the Specialized Functional Unit (SFU)*

The SFU requires 62 bits for control signals. The fused basic functional unit has six 8-input multiplexers, and the complex functional unit has three 8-input multiplexers; as each 8-bit multiplexer requiers 3 control bits, the SFU requires 27 bits to select the inputs to the various internal functional units. With two 4-input multiplexers at the output, 4 additional bits are required for output register selection. The SFU has three shifters, each of which has a 2-input multiplexer driving one of its inputs, so 3 additional bits are required to control these multiplexers. 15 bits are required for register storage (5 bits per functional unit), 12 bits for operation control (each ALU and MAC supports up to 16 operations, so 4 control bits for each are required), and 1 bit is required for bypass selection within the fused basic functional unit. This adds up to 62 control bits in total.

V. JiTC Architecure

This section describes how to integrate one or more SFUs into a processor pipeline to achieve Just-in-Time Customization. To simplify discussion, we assume a simple 5-stage, single-issue in-order RISC pipeline. Figure 7 shows the modified pipeline structure.

ISE Encoding and Decoding: We assume 32-bit ISA with 4-bits per register encoding corresponding to a 16-entry register file. As shown in Figure 8(a), 12 bits are required to encode two source and one destination registers and 8-bit opcode supports 256 ISEs.

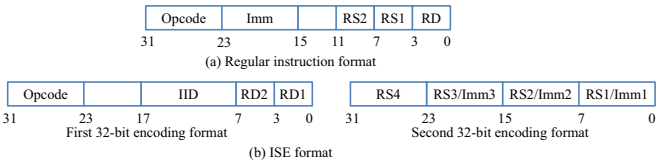


Fig. 8: *ISE encoding format*

With a 32-bit instruction format, it is necessary to encode ISEs using two consecutive 32-bit words, as shown in Figure 8(b). The first 32-bit is used to encode 8-bit Opcode and two 4-bit destination registers. We can support at most one ISE if one of the opcodes (out of 256) is reserved for ISEs. To extend the number of ISEs, we use 10 unused bits to encode the IID or ISE ID. When the opcode signifies an ISE, the IID field indicates which ISE it is. This allows us to encode at most 1024 ISEs. The second 32-bit encodes the source registers and/or immediate values. Note that we support ISEs with different addressing modes. For example, an input operand can be a register index or an immediate value. As at least one input operand should be a register, we assume the first input operand is always a register index; the other three are either registers or immediate. So we have four addressing modes: RRRR (all registers), RRRI (one immediate), RRII (two immediates), and RIII (three immediates), which can be

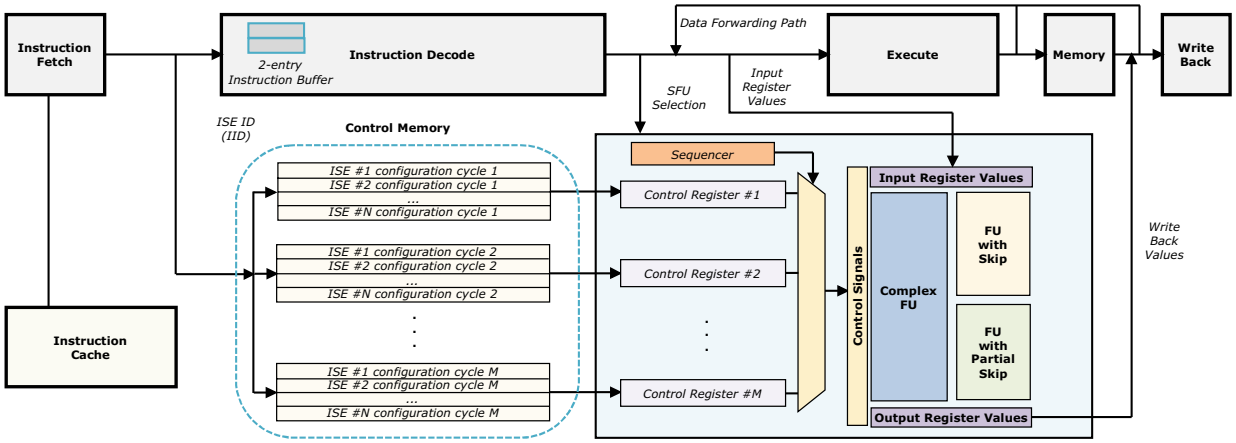


Fig. 7: *Just-in-Time Customizable (JiTC) processor architecture: Integration of SFUs in the pipeline datapath*

supported by reserving four opcodes in the ISA.

To support ISE decoding, we require a 2-entry instruction buffer between the fetch and the decode stage so that the decoder has access to the entire 64-bit ISE. When the decoder detects an ISE opcode, it decodes the second half of the ISE in the buffer for the source operands.

Multi-cycle execution of ISE on SFU with multi-banked control memory: The SFU supports single-cycle execution of most ISEs whose critical paths consist of up to 3 operators. Indeed, *almost 90% of the ISEs can be executed in one cycle on the SFU*. Through reconfiguration, SFU can also support multi-cycle execution to achieve maximal speedup brought by ISEs with longer critical paths. Reconfiguring the SFU involves changing the values of the 62 control bits each cycle. It is compiler's job to exploit the reconfiguration ability of SFU, which we will detail in Section VI. According to our observation, almost 99.77% of the ISEs can be executed within 4 cycles.

Each ISE requires more control signals than regular instructions, especially when multi-cycle execution is taken into consideration. In JiTC core, the control signals for each ISE are stored in an on-chip control memory that is accessed in parallel with the instruction decode phase of the pipeline when an ISE is decoded. The IID field in the ISE is used as an index into the control memory. As shown in Figure 7, to support multi-cycle execution, the control memory consists of M banks, where the i^{th} bank stores the control signals required for the execution of an ISE on the SFU in the i^{th} cycle ($M = 4$ in our design). The banks are accessed in parallel to retrieve all the control signals of an ISE. With 10-bit IID field, storage space for 1024 entries is required, where each entry holds 62 bits; the approximate size of the control memory is 32KB. Additionally, the control memory needs to store the number of cycles required to execute each ISE. The number of cycles and the control bits read from the control memory are written into the SFU's sequence and control registers, respectively.

For a single-issue in-order extensible processor, only one SFU needs to be integrated; however, for multi-issue out-of-order execution, our experiments confirm that four SFUs achieve near-optimal acceleration. When all of the input operands to an ISE are ready, the ISE can start execution on the SFU. When the ISE execution inside the SFU completes, the output operands are written to the register file and the SFU becomes free to execute another ISE.

VI. Compiler Support

Our JiTC architecture acquires the compiler support consisting of automated ISE identification process, a graph-based mapper to synthesize ISE onto SFU and generators for final executable binary and configurations.

ISE identification and selection: Given an application, we first detect the "hot" basic blocks through profiling. The DFGs of these hot basic blocks are then analyzed to identify all the ISE candidate patterns [38]. We impose the restriction of at most 4 input operands and 2 output operands per candidate pattern as noted earlier [11], [12], [37]. We also do not allow memory accesses and control flow operations within an ISE. Once all the candidate patterns have been identified, a subset of these patterns is selected such that (a) each node in the DFG of a basic block is covered by at most one candidate pattern, and (b) the cumulative speedup of the selected patterns is maximized. The speedup of a pattern is defined as $\frac{t_{sw}}{t_{custom}}$, where t_{sw} is execution cycles on the base processor core and t_{custom} is the execution cycles when the pattern is implemented in customized circuit.

Mapping algorithm: A graph-based mapper is employed by our compiler to synthesize the ISEs onto SFU. To exploit the reconfigurability feature of SFU, we borrow the notion of Routing Resource Graph (RRG) [24] from the FPGA domain to represent the resources of SFU in different cycles and the connections among them. The connections are generated such that the components of the SFU in one cycle are connected to the components of the SFU in the next cycle. For example, the RRG in Figure 10(a) shows how the components of the SFU in cycle 1 are connected to the components of the SFU in cycle 2. Basically, each of the three functional units in cycle 1 is connected to any one of the three functional units and the output registers in cycle 2, which means that the value generated by one functional unit could be read by any functional units or stored in the output registers in the next cycle. Note that the input registers are connected to the functional units in the same cycle as their values could be read within the period of one cycle. Similarly, the connection between two fused functional units also appears in the same cycle. The objective of the mapper is to map the DFG to the RRG of the SFU so that the number of cycles required is minimized.

The mapping algorithm uses a greedy heuristic. Basically, it maps the nodes in the DFG one by one and each node is placed closed to its predecessors as much as possible. Its pseudo code is presented in Algorithm 1.

Algorithm 1: Mapping_algorithm

```
Input: The data flow graph (DFG) of the ISE and the SFU architectural
       specification.
Output: The result RRG if mapping is successful.
Begin
    order_list = ALAP_ordering(DFG);
    RRG = Initialize_RRG(SFU);
    For Each operator u in order_list do
        successful = 0;
        /*Take care of operator chaining within one functional unit.*/
        If u has only one immediate predecessor v and u is v's only immediate
        successor then
            If Map(v)→Res(u) == Available and Map(v)→Res(v) is connected
            to Map(v)→Res(u) then
                Map(v)→component(Res(v)) = Occupied;
                successful = 1;
        If successful == 0 then
            /*Find and map to the closest available functional unit.*/
            For cycle = 1 to 4 do
                If successful == 1 then
                    break;
                For all functional unit n in RRG(cycle) do
                    If n→status == Free and n→component(Res(v)) ==
                    Available then
                        Feasible = 1;
                        For each immediate predecessor v of u do
                            If Map(v) is not connected to n in RRG then
                                Feasible = 0;
                        If Feasible == 1 then
                            n→status = Mapped;
                            n→component(Res(v)) = Occupied;
                            successful = 1;
                            break;
        If successful == 0 then
            Return FAIL;
    Return RRG;
```

To order the DFG nodes, we assign level values to each of the nodes in the DFG according to an *As Late As Possible (ALAP)* scheduling policy and sequentialize the nodes in the same level from left to right. The nodes (functional units) in the RRG are also ordered according to their time cycles. We also ensure that the two basic functional units (BFU) are ordered first to have higher priorities in mapping compared to the complex functional unit in the same cycle.

The greedy heuristic works as the following. Suppose we are mapping a node u in the DFG that has predecessors $v_1, v_2, \ldots v_x$. We identify the closest common free successor functional unit of $Map(v_1), Map(v_2), \ldots, Map(v_x)$. $Map(v)$ stands for the functional unit to which operator v has been mapped to. We simply map u to this free functional unit. If u has no predecessors, then the chosen free functional unit would be the one with minimal cycle value.

To explore the potential of operator chaining within one functional unit, we need to consider the components within one functional unit. Fortunately, the only special case we need to take care of is when u has only one immediate predecessor v and v has u as its only immediate successor. This is because there is no external connection between two connected components within one functional unit. Suppose $Res(u)$ stands for the component resource u requires. If the functional unit $Map(v)$ has an available component $Res(u)$ and its component $Res(v)$ is connected to the component $Res(u)$, then we can directly map u to functional unit $Map(v)$.

The process ends once all the nodes of the DFG have been mapped to the RRG. In the rare event that the mapping fails because a pattern requires more than 4 cycles, the pattern cannot be accelerated using our SFU and is eliminated from further consideration.

Mapping example: We now show an example of how the DFG in Figure 2 is mapped to the RRG of SFU. First, we assign order to the DFG nodes using *ALAP* scheduling policy, shown in Figure 9. The first operator we encountered is a *multiplication*; so we find the first free complex functional unit with the lowest cycle time, which is the complex functional unit in cycle 1 and map this *multiplication* to the multiplier component of it.

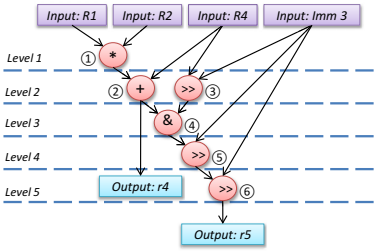
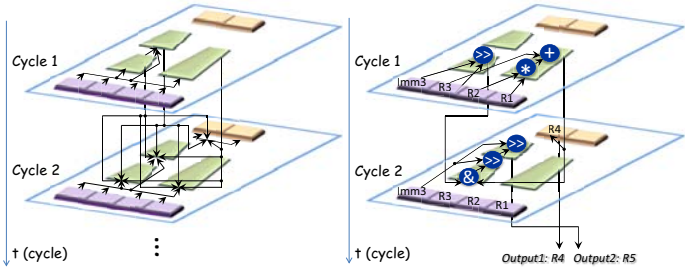


Fig. 9: L*evel order assignment for DFG nodes with ALAP scheduling*



(a) Routing Resource Graph (RRG) of SFU (b) Final mapping

Fig. 10: *Routing Resource Graph (RRG) of the SFU and the final mapping of the DFG to the RRG*

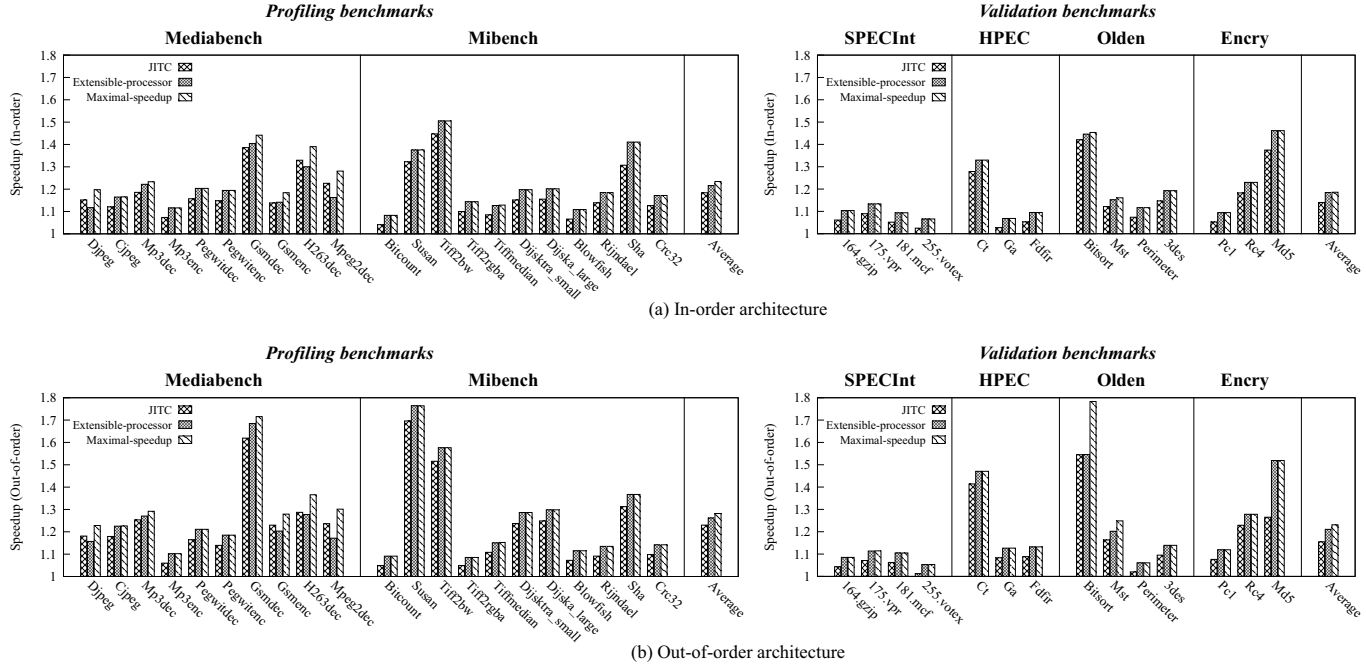
The second operator is an *addition* operator. The *addition* has only one immediate predecessor which is the *multiplication* we just mapped and the *multiplication* has this *addition* as its only immediate successor. So we can try to map this *addition* to the same complex functional unit. Fortunately, we find the MAC inside the complex functional unit can support this mapping with the connection requirement satisfied.

The next operator is a *shift*. We simply find the first free functional unit in cycle 1 and map it there. Now we continue to map the *and* operator. The *and* operator has two predecessors; so the earliest common successor should be a functional unit in cycle 2. So we pick the first basic functional unit in cycle 2 as it has higher priority. Then the operator *shift* can be mapped to the same functional unit as the *and* operator. Finally, the last *shift* operator is mapped to the second basic functional unit in cycle 2 as it is the closest one. The final mapping is shown in Figure 10(b).

Binary executable and configuration generation: Once the compiler decides on the mapping of an ISE to the SFU, it generates the corresponding control signals for the ISE. The compiler then generates the binary executable that replaces, for each occurrence of a candidate pattern, a sequence of instructions from the base ISA with the corresponding ISE. Finally, the control signals are loaded into the control memory before the application initiates execution. Note that as the subset of ISEs selected is different for different applications, the content of the control memory is different for each application. In other words, the JiTC architecture achieves flexibility by changing the content of the control memory and thereby instantiating different ISEs per application.

VII. Experimental Evaluation

Experimental Setup: We evaluate the performance of JiTC core compared to traditional extensible processors with dedicated and high performance ISE circuits [2], [3], [18], [27], [37], [38]. As mentioned



(a) In-order architecture

(b) Out-of-order architecture

Fig. 11: *Speedup of JiTC and extensible processor over the baseline processor and the maximal speedup for extensible processor with unlimited area*

earlier, we selected 21 benchmark applications from MiBench [15] and MediaBench [21] to derive the design of the SFU. Here *we use 14 additional applications from SPECInt, HPEC, Olden, and Encrypt benchmark suites to perform cross validation of the SFU design.*

For a fair comparison, we design both the extensible processors and the JiTC core by augmenting a RISC-like baseline core [5] with no accelerator. For each of the 35 applications, we custom design an extensible processor following the standard ISE identification and selection methodology [37], [38]. That is, we design a total of 35 individual extensible processors, where each extensible processor is capable of accelerating the specific application it is designed for.

We assume that the clock period of the baseline core is determined by the latency of the MAC unit [2], [37], which also has roughly the same latency as a multiplier [31]. All the designs are synthesized using Synopsys Design Compiler version E-2010.12-SP4 with Free PDK 45nm standard cell library. The MAC unit has a latency 1.58ns; thus the frequency of the baseline core and all the extensible processors are set at 633MHz. JiTC core, however, has a frequency of 606MHz constrained by the SFU latency as shown in Table I. Further optimizations could lead to higher frequency of JiTC core.

Following prior works [11], [12], [37], we assume that each extensible processor can support ISEs with at most 4 input and 2 output operands, and cannot include any memory or control operations. The latency of an ISE in the extensible processor is obtained by dividing the latency along the critical path by the clock period of the baseline core. The area of each individual extensible processor is restricted to the area of the JiTC core. This area restriction leads to only 1.5% average performance degradation compared to the speedup of an extensible processor with unlimited area. We call the speedup obtained from the extensible processor with I/O constraint but without area constraint as *maximal speedup*.

We modified the SimpleScalar simulator [5] to integrate the SFUs and corresponding control memory in the pipelined datapath. We modeled both in-order and out-of-order pipelines for JiTC core and the extensible processors. For the extensible procesors, we assume that all ISEs are implemented as dedicated functional units in the pipeline. We extended the instruction set to support the ISE formats, and modified the gcc cross-compiler for SimpleScalar to identify the ISEs for each application and to generate binary executables that include calls to the ISEs. Table II shows the configurations for both in-order and out-of-order micro-architecture in SimpleScalar simulator setup. The configuration parameters are chosen to closely match realistic in-order (ARM Cortex-A7) and out-of-order (ARM Cortex-A15) embedded processors.

	In-order architecture	Out-of-order architecture
Pipeline	1 way	4 ways
RUU size	2 entries	128 entries
IFQ size	4 entries	16 entries
LSQ size	2 entries	16 entries
L1 I-Cache	32KB, 2-way, 1 cycle hit	
L1 D-Cache	32KB, 2-way, 1 cycle hit	
Unified L2	512KB, 4-way, 10 cycle hit	
Control memory	32KB	

TABLE II: *Simulated processor configurations*

Results for profiling benchmarks: Let us first focus on performance comparison with profiling benchmarks (MiBench, MediaBench) used to derive the design of the JiTC core (left of Figure 11). For in-order pipeline, Figure 11(a) shows the performance of JiTC core and the extensible processors compared to the baseline core with no accelerator. The speedup is defined as $\frac{t_{sw}}{t_{custom}}$ where t_{sw} is execution cycles on the baseline core and t_{custom} is the execution cycles on an extensible processor or a JiTC core. We also plot the maximal speedup for extensible processors in Figure 11. JiTC architecture achieves an average speedup of 1.184X, which is 97.40% of the speedup achieved by extensible processors (1.216X) and 94.93% of the maximal speedup (1.234X). The slight loss in performance of the JiTC core comes from two sources: the reduced clock frequency and multi-cycle execution of 10% ISEs on the SFU. The remaining 90% ISEs can execute in single-cycle on the SFU. More importantly, *JiTC has huge advantage in terms of flexibility: we*

need a different extensible processor to accelerate each application, while a single JiTC core can accelerate all the different applications with minimal performance loss. This is the fundamental reason why for some large benchmarks, JiTC can outperform extensible processor which is area-constrained to support limited number of feasible ISEs.

For out-of-order pipeline, we use 4-way decode, issue, execute, and commit. As expected, we need at most 4 SFUs in this case to achieve maximal speedup. For out-of-order pipeline, Figure 11(a) shows the performance of JiTC core and extensible processor compared to the baseline processor with no accelerator. Here JiTC achieves an average speedup of approximately 1.230X across all benchmarks in Mediabench and Mibench, which is 97.54% of the speedup achieved by the extensible processor (1.262X) and 95.98% of the maximal speedup (1.282X).

Results for validation benchmarks: The benchmarks from MiBench and MediaBench were used to derive the design of JiTC core. Our objective, however, is to design a flexible architecture that can support any contemporary or emerging application domains. In order to stress test the design of our architecture, we attempt to accelerate applications from benchmark suites with completely different characteristics compared to the embedded space. We chose SPECInt [25], Encryption, Olden, and HPEC [16] benchmark suites for this evaluation. HPEC is derived from HPCC [22], [13] and PCA [20] both targeting general-purpose high-performance computing.

These validation results are shown in the right of Figure 11. JiTC still achieves similar speedup to extensible processors, around 96% on an average for both in-order and out-of-order executions. This confirms that *even though the JiTC core was designed to accelerate embedded applications, the design is flexible enough to support a completely different application domain*, e.g., SPEC and HPEC. However, for these benchmarks, the speedup achieved using customization (extensible processor or JiTC) is limited to around 1.10X. This is because these benchmarks have lower ratio of ALU operations and smaller basic blocks [8], characteristics that are not ideal for customization.

VIII. Conclusion

In this paper, we proposed a Just-in-Time Customizable (JiTC) processor core that can accelerate execution across application domains. The heart of our innovation is a specialized functional unit (SFU) that can execute most application-specific instructions at traditional extensible processor-like efficiency through fast reconfiguration. The SFU is integrated in the processor pipeline though an enhanced decoding mechanism and the compiler is modified to support to the mapping of ISEs to the SFU. Experimental results show that JITC architecture offers traditional extensible processor-like performance with far superior flexibility.

IX. Acknowledgments

This material is based upon work supported by the the Singapore Ministry of Education Academic Research Fund Tier 2 MOE2009-T2-1-033, MOE2012-T2-1-115, and the United States National Science Foundation under grants 1210182 and 1035603. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] G. Ansaloni et al. Design and architectural exploration of expression-grained reconfigurable arrays. In *SASP*, 2008.
[2] K. Atasu et al. Automatic application-specific instruction-set extensions under microarchitectural constraints. *DAC*, 2003.
[3] K. Atasu et al. An integer linear programming approach for identifying instruction-set extensions. In *CODES+ISSS*, 2005.
[4] K. Atasu et al. Fast custom instruction identification by convex subgraph enumeration. In *ASAP*, 2008.
[5] T. M. Austin et al. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2), 2002.
[6] L. Bauer et al. RISPP: rotating instruction set processing platform. In *DAC*, 2007.
[7] P. Bonzini et al. Compiling custom instructions onto expression-grained reconfigurable architectures. In *CASES*, 2008.
[8] A. Bracy et al. Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In *MICRO*, 2004.
[9] A. Bracy and A. Roth. Serialization-aware mini-graphs: Performance with fewer resources. In *MICRO*, 2006.
[10] T.J. Callahan et al. The Garp architecture and c compiler. *IEEE Computer*, 33(4), 2000.
[11] N. Clark et al. Application-specific processing on a general-purpose core via transparent instruction set customization. In *MICRO*, 2004.
[12] N. Clark et al. An architecture framework for transparent instruction set customization in embedded processors. In *ISCA*, 2005.
[13] J.J. Dongarra and P. Luszczek. Introduction to the hpcchallenge benchmark suite. Technical report, DTIC Document, 2004.
[14] V. Govindaraju et al. Dynamically specialized datapaths for energy efficient computing. In *HPCA*, 2011.
[15] M. R. Guthaus et al. MiBench: a free, commercially representative embedded benchmark suite. In *WWC*, 2001.
[16] R. Haney et al. The high performance embedded computing (HPEC) challenge benchmark suite. In *HPEC Workshop*, 2005.
[17] S. Hauck et al. The Chimaera reconfigurable functional unit. *IEEE Trans. VLSI Syst.*, 12(2), 2004.
[18] Tensilica Inc. http://www.tensilica.com.
[19] R. Koenig et al. KAHRISMA: a novel hypermorphic reconfigurable-instruction-set multi-grained-array architecture. In *DATE*, 2010.
[20] J. Lebak et al. Polymorphous computing architecture (PCA) kernel-level benchmarks. Technical report, DTIC Document, 2005.
[21] C. Lee et al. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO*, 1997.
[22] P. R Luszczek et al. The HPC challenge (HPCC) benchmark suite. In *SC*, 2006.
[23] R. Lysecky et al. Warp processors. *ACM TODAES*, 11(3), 2006.
[24] L. McMurchie and C. Ebeling. Pathfinder: a negotiation-based performance-driven router for FPGAs. In *FPGA*, 1995.
[25] Spec org. Spec cpu benchmark suits. http://www.spec.org/cpu.
[26] Y. Park et al. CGRA express: accelerating execution using dynamic operation fusion. In *CASES*, 2009.
[27] L. Pozzi et al. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(7), 2006.
[28] P. G. Sassone et al. Static strands: safely collapsing dependence chains for increasing embedded power efficiency. In *LCTES*, 2005.
[29] P. G. Sassone and D. S. Wills. Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication. In *MICRO*, 2004.
[30] H. Singh et al. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Computers*, 49(5), 2000.
[31] P. F. Stelling and V. G. Oklobdzija. Implementing multiply-accumulate operation in multiplication time. In *ISCA*, 1997.
[32] M. Stojilovic et al. Selective flexibility: Breaking the rigidity of datapath merging. In *DATE*, 2012.
[33] F. Vahid et al. Warp processing: Dynamic translation of binaries to FPGA circuits. *Computer*, 41(7), 2008.
[34] S. Vassiliadis, J. Phillips, and B. Blaner. Interlock collapsing alu's. *IEEE TC*, 42(7), 1993.
[35] A. Ye and J. Rose. Using bus-based connections to improve field-programmable gate array density for implementing datapath circuits. In *FPGA*, 2005.
[36] S. Yehia and O. Temam. From sequences of dependent instructions to functions: An approach for improving performance without ILP or speculation. In *ISCA*, 2004.
[37] P. Yu and T. Mitra. Characterizing embedded applications for instruction-set extensible processors. In *DAC*, 2004.
[38] P. Yu and T. Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *CASES*, 2004.