Exploring the Design Space of LUT-based Transparent Accelerators

Sami Yehia[1], Nathan Clark[2], Scott Mahlke[2], and Krisztián Flautner[1]

[1]ARM, Ltd.
Cambridge, United Kingdom
{sami.yehia, krisztian.flautner}@arm.com

[2]Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{ntclark, mahlke}@umich.edu

ABSTRACT

Instruction set customization accelerates the performance of applications by compressing the length of critical dependence paths and reducing the demands on processor resources. With instruction set customization, specialized accelerators are added to a conventional processor to atomically execute dataflow subgraphs. Accelerators that are exploited without explicit changes to the instruction set architecture of the processor are said to be transparent. Transparent acceleration relies on a light-weight hardware engine to dynamically generate control signals for the accelerator, using subgraphs delineated by a compiler. The design of transparent subgraph accelerators is challenging, as critical subgraphs need to be supported efficiently while maintaining area and timing constraints. Additionally, more complex accelerators require more sophisticated control generation engines. These factors must be carefully balanced. In this work, we investigate the design of subgraph accelerators using configurable lookup table structures. These designs provide an effective paradigm to execute a wide range of subgraphs involving arithmetic and logic operations. We describe why lookup table designs are effective, how they fit into a transparent acceleration framework, and evaluate the effectiveness of a wide range of designs using both simulation and logic synthesis.

Categories and Subject Descriptors

B.2.0 [**Arithmetic and Logic Structures**]: [General]; C.3 [**Special-Purpose and Application-Based Systems**]: [Real-time and Embedded Systems]; C.4 [**Performance of Systems**]: [Performance Attributes]

General Terms

Design, Experimentation, Performance

Keywords

Embedded Processing, Efficient Computation, Accelerator Design

CASES'05, September 24–27, 2005, San Francisco, California, USA.

1. INTRODUCTION

Application-specific instruction set extensions are an effective method to increase application performance in the context of a general-purpose processor. With this approach, specialized hardware computation accelerators in the form of new function units are added into a processor. These accelerators are customized to execute important dataflow subgraphs found in an application as atomic units. The use of subgraph accelerators reduces the latency of the subgraph's execution, improves the utilization of other processor resources, and reduces the burden of storing temporary values to the register file. Instruction set extensions offer the important advantage of a post-programmable solution over more traditional ASIC accelerators.

Subgraph accelerators come in two general forms: visible and transparent. The visible approach extends the instruction set architecture of a baseline processor with a small number of new operations to exploit the accelerators [2, 7, 11, 12, 15, 25]. In essence, an application specific instruction processor (ASIP) is created by extending a baseline design. Commercial tool chains that automate some of this process are available, including the Tensilica Xtensa, ARC Architect, and ARM OptimoDE. Visible accelerators offer the advantage of simplicity and low hardware cost. However, they suffer from high non-recurring engineering costs including design and verification time, new lithography mask sets, and migration of the software tool chain to a new instruction set.

The transparent approach incorporates subgraph accelerators without augmenting the processor instruction set. The key idea is that during program execution subgraphs are identified and new instructions are generated to map and execute subgraphs on the accelerator [9]. In this manner, the instruction set of the baseline processor is not explicitly changed. A light-weight dynamic optimization engine is used to alter the instruction stream at run-time to exploit the accelerator. Transparent acceleration requires more hardware overhead in the form of the dynamic optimization engine, but substantially reduces the non-recurring engineering costs.

To accomplish transparent acceleration, three components are necessary: a subgraph discovery mechanism, a dynamic control generator, and the subgraph accelerator substrate. Subgraph discovery identifies the dataflow subgraphs that will be mapped and executed on the accelerator. Subgraphs can be identified offline using a compiler [10], through binary translation [14], or online using a more heavy-weight dynamic optimizer, such as rePlay [20]. Control generation occurs at run-time using a configurable hardware pattern matcher. This component dynamically recognizes subgraphs in the instruction stream, and substitutes them with new instructions that exploit the accelerator.

The focus of this paper is on the final component, the accelerator

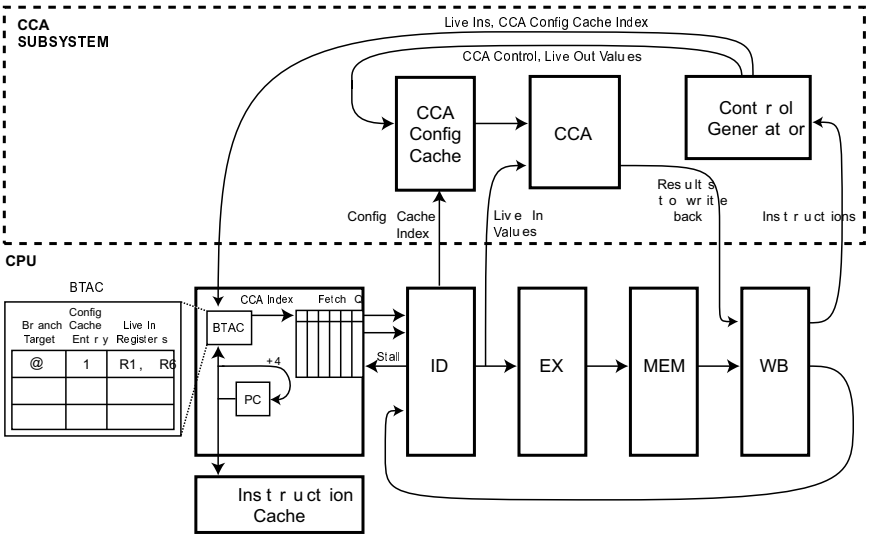

Figure 1: Transparent instruction set customization architecture.

substrate. The substrate is responsible for the actual execution of the subgraphs. An effective design must be capable of executing a wide variety of domain-specific instruction subgraphs faster and more efficiently than a conventional processor pipeline. It must also be both cost effective and power efficient to make its use feasible in an embedded computing environment, and be amenable to efficient run-time control generation. The final issue is the most difficult to quantify, but implies a programmable substrate that is configured with a modest number of control signals.

A parameterized lookup table (LUT) based computation accelerator is used as the accelerator substrate. LUT based accelerators were previously introduced in [29], where every bit was given a separate LUT configuration. Here, a generalization of the prior technique, referred to as a programmable carry function unit, or PCFU, is used. The PCFU leverages the design of carry lookahead adders to break the cascaded tree of LUTs in the original design, creating a faster and more efficient design. With an underlying lookup table structure, the PCFU is completely configurable and can execute a large variety of arithmetic subgraphs. The PCFU is dynamically configurable by analyzing target subgraphs and calculating the LUT entries.

The PCFU is not a single design, but rather specifies a parameterized design space that offers complex tradeoffs between subgraph execution capabilities with the cost and worst-case delay of the substrate. This paper presents a systematic exploration of the PCFU design space. We examine the critical tradeoffs associated with designing LUT based arrays including LUT size, number of carry signals that are propagated, and support for non-LUT operators, such as shift. To perform this exploration, a complete compilation and simulation system for PCFUs based on the ARM-9 processor are used. PCFU designs are developed in Verilog and synthesized to measure area and delay.

2. BACKGROUND INFORMATION

The design space exploration in this work is based on a transparent instruction set customization framework. In order to understand the idea, this section provides an overview of that framework, including changes required to a standard processor pipeline, and a discussion of related techniques.

2.1 Architectural Framework

In transparent instruction set customization, subgraphs are delineated by a compiler without augmenting the baseline ISA, and then replaced in the instruction stream dynamically. Although determining the most effective execution substrate for dataflow subgraphs is certainly applicable to other architectures, transparent instruction set customization is the architecture chosen for evaluation.

Figure 1, from [10], shows a processor with transparent instruction set customization. In this architecture, subgraphs are delineated with a Branch-and-Link (BRL) instruction commonly used for function calls. The first time a BRL-to-subgraph is encountered, the processor treats it just like any other function call, where the subgraph instructions are executed in the standard pipeline. After retirement, these instructions are fed to a control generator.

The control generator recognizes that each BRL instruction may signal the beginning of a subgraph and attempts to map subsequent instructions onto the execution substrate. The authors in [10] use an array of combinational function units as the basis of execution, and term it a "configurable compute array," or CCA. The control generator places each subgraph instruction onto an element of the combinational array, keeping track of the communication between array nodes. This information is used to configure the CCA to execute the subgraph on subsequent encounters of the BRL. The control generator may be thought of as a complex decode unit, where the instruction being decoded is an entire dataflow subgraph, and this "instruction" is expressed using several instructions from the processor's baseline instruction set. In the event that the control generator cannot map the subgraph onto the execution substrate, e.g., if the BRL was to an actual function such as printf, then the control generator simply throws away its current set of control signals and waits for the next BRL.

Once the control signals are generated for a dataflow subgraph, they are written to a configuration cache. Additionally, a branch target address cache (BTAC) is updated to reflect that this new configuration exists in the configuration cache. The BTAC is indexed using the address of the BRL which marked the beginning of the subgraph. Whenever that BRL is encountered in the dynamic instruction stream again, the fetch stage will index into the BTAC and see that control has been generated for it. An index to the configura-

tion cache entry will then be passed through the pipeline, allowing the subgraph to be executed just like any other instruction. It is important to note that in order to prevent pipeline bubbles, whenever a BRL has an entry in the BTAC, control is not diverted to the branch target.

In this paper, we explore different implementations of the subgraph execution substrate in the context of transparent instruction set customization. Section 3 describes the execution of subgraphs using a lookup-table based approach. We then explore how different design decisions affect the performance and hardware characteristics of the resultant system in Section 4.

2.2 Related Work

A great deal of research has been done to develop dataflow subgraph accelerators. Examples include PRISM [3], PRISC [22], OneChip [8], DISC [27], GARP [13], and Chimaera [28]. All of these designs are based on a tightly integrated FPGA, which allows for very flexible computations. However, there are several drawbacks to using FPGAs. One problem is that the flexibility of FPGAs comes at the cost of long latency. While some work has addressed the issue, implementing functions in FPGAs remains inefficient when compared to ASICs that perform the same function, primarily because of routing costs. FPGA reconfiguration time can be slow and the amount of memory needed to store the control bits can quickly grow out of control. To overcome the computational inefficiency and configuration latency, the focus of most prior work dealing with configurable computation units was on very large subgraphs, which allows the amortization of these costs. This work differs in that we focus on acceleration at a finer granularity, encompassing stateless, acyclic dataflow subgraphs.

Other work [5, 23, 16, 24, 21] proposed subgraph execution structures specifically optimized for linear chains of execution. That is to say these structures only execute subgraphs that have two inputs, one output, and a small number of intermediate nodes. Constraining the subgraphs in this way has been shown to effectively increase the bandwidth of execution resources; however, it restricts the performance increase from dataflow graph compaction [30]. In this work, we develop a more generic architecture, to support the execution of more arbitrary acyclic dataflow subgraphs. These subgraphs are larger than simple linear subgraphs, and attack the computation limitations of processors more than the resource limitations.

The work in this paper is similar to that in [29] and [9], in that both of these previous works propose subgraph accelerators to execute stateless, acyclic dataflow subgraphs. This paper builds on these previous works, by improving the design in [29] and exploring many designs ignored by the previous work. Evaluation of these designs is performed using a transparent instruction set customization framework proposed in [10], although the execution substrate in this paper is entirely different.

3. PROGRAMMABLE CARRY FUNCTION UNIT

The design of the accelerator substrate on which customized instructions are executed is a major challenge in transparent instruction set customization. The accelerator should be programmable enough to cover most of the recurrent subgraphs, and at the same time be easy to configure and have low latency to execute subgraphs efficiently.

In this section, we describe a LUT-based accelerator, the Programmable Carry Function Unit (PCFU). The PCFU can execute subgraphs of any number of logical operations and a predefined

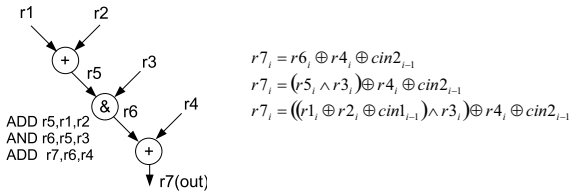

Figure 2: An example dataflow subgraph and the output expressed as a function of the inputs.

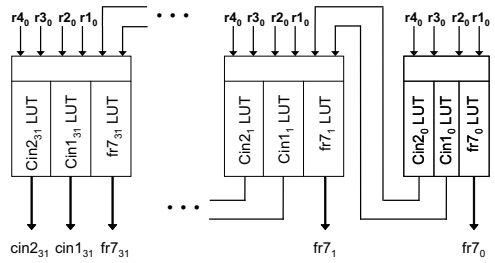

Figure 3: Organization of the LUT based *Function* unit from prior work.

number of additions/subtractions. The PCFU offers the advantage of being sufficiently programmable to cover a wide variety of subgraphs, while maintaining a relative low interconnect complexity and latency compared to FPGA devices.

3.1 Principles

The PCFU approach builds on the principles introduced in [29], with the basic idea being to extract a logical expression of each output bit as a function of the inputs to the subgraph. Given this logical expression, a LUT stores the truth table corresponding to this expression, which is used later to directly compute the output given the inputs.

Consider the example of Figure 2. Each bit of the output register, `r7`, can be expressed as a logical function of `r1`, `r2`, `r3`, `r4` and the carry bits from any additions in the subgraph. For example, the output function for bit `i` of the output of the subgraph can be expressed as $fr7_i = r6_i \oplus r4_i \oplus cin2_{i-1}$, since this is the definition of a bitwise add. Next, $r6_i$ can be re-expressed as $(r5_i \wedge r3_i)$, yielding the second equation in Figure 2. Likewise, $r5_i$ can be expressed as a function of $r1$, $r2$, and the carry signal generated by the first addition in the subgraph. Once this is done, each bit of $r7$ is expressed as a logical function of only the inputs and the carry signals, shown at the bottom of Figure 2. Using this process allows for expressing any sequence of logical, integer instructions as a function of the input registers of the subgraph. This enables direct mapping of subgraphs into lookup tables, with the only difficulty being the need to calculate the carry signals.

Figure 3 shows the accelerator proposed in [29], called the *Function* unit. In this design, each carry bit is calculated and forwarded to the higher significant bit. The LUTs $fr7_i LUT$, $cin1_i LUT$, and $cin2_i LUT$ implement the functions `fr7`, `cin1`, and `cin2` respectively. The *Function* unit provides fine grain programmability and flexibility by specifying different LUT configurations for each output and carry bit. This high flexibility comes at the cost of high latency because of the ripple scheme to propagate the carry to upper significant bits. Also, this accelerator requires a large amount of control data to configure each bit and their associated carries.

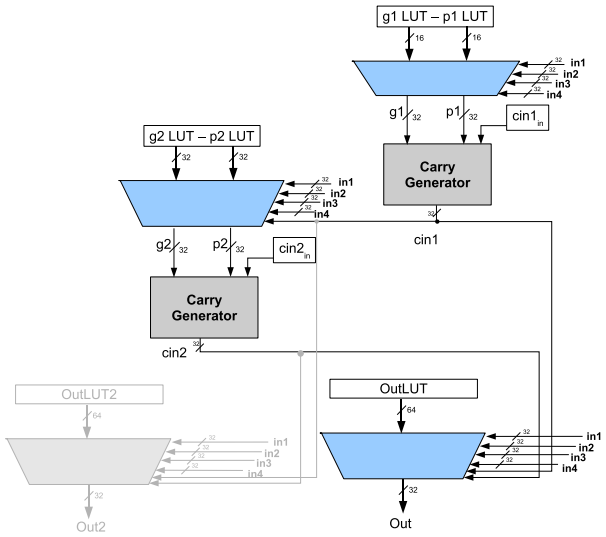

Figure 4: Baseline PCFU design.

The PCFU is also a LUT-based accelerator, but avoids both the large configuration and the high latency of the ripple carry propagation by defining one LUT (and associated carry LUTs) for all output bits. Aside from saving on-chip space, this approach allows us to leverage fast carry propagation schemes, such as Kogge-Stone [17] or Brent-Kung [6] parallel prefix adders.

Most of the existing fast carry propagation techniques are based on first calculating a (g_i, p_i) pair [18], where given inputs bits a_i and b_i, $g_i = a_i \wedge b_i$ (generate), and $p_i = a_i \oplus b_i$ (propagate). If subtraction is done instead of addition, b_i is replaced by $\bar{b_i}$. This p-g pair of values is then fed to a carry propagation network to calculate the carry bits. The PCFU design generalizes the calculation of the (g_i, p_i) pair by creating a pair of LUT configurations (gi LUT - pi LUT) for each addition/subtraction. For example, in Figure 2 the (g,p) pairs of the 2 additions can be expressed as $g1_i = r1_i \wedge r2_i$, $p1_i = r1_i \oplus r2_i$, $g2_i = ((r1_i \oplus r2_i \oplus cin1_{i-1}) \wedge r3_i) \wedge r4_i$, and $p2_i = ((r1_i \oplus r2_i \oplus cin1_{i-1}) \wedge r3_i) \oplus r4_i$. By separately computing the carry signal using these LUTs and carry propagation networks, the PCFU breaks the dependence of output bits on the values of lower order input bits. That is, bit 31 of the output is not a function of bit 0 of the input values as long as we have the carry signal precomputed. This enables the PCFU to have a much lower latency than most FPGA-based accelerator designs, which need to propagate the carry signal from lower order bits.

Figure 4 shows the design of a PCFU that can collapse a sequence of dependent instructions with up to two additions or subtractions, and *any* number of logical operations, given a fixed number of inputs. Note that, although Figure 4 may suggest that the two additions/subtractions need to be dependent, the PCFU can collapse any two addition/subtraction regardless of their position in the subgraph. That is, they may be in parallel, dependent or even interleaved with other logical operations.

For a given subgraph, the basic idea of the PCFU is to generate a LUT (OutLUT) configuration for the output function and appropriate configurations (gi LUT and pi LUT) to generate the carries for each individual addition/subtraction in the subgraph. The purpose of the $cini_{in}$ signal is to implement subtractions. The primary benefit of using the PCFU over previous work [29] is the use of more advanced carry generation networks and fewer configuration bits in the accelerator.

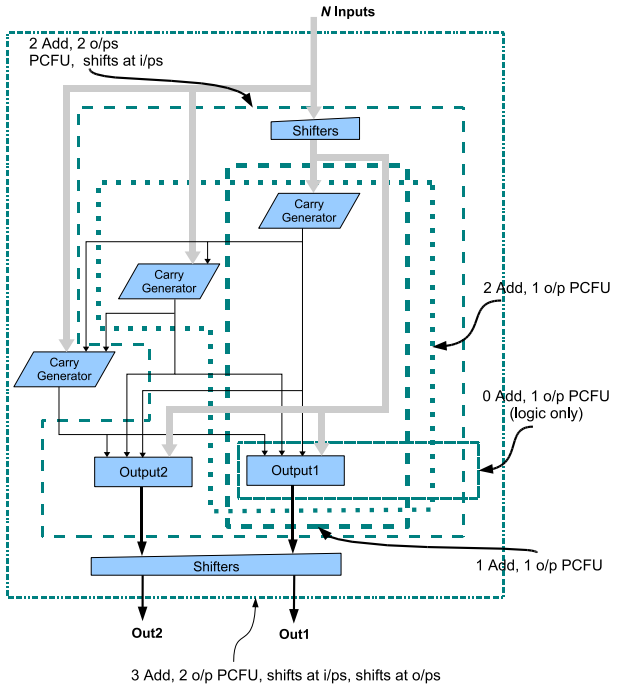

Figure 5: PCFU design space.

3.2 PCFU Design Space

Figure 5 shows basic building blocks of a generalized PCFU that can support N inputs, 3 additions/subtractions, 2 outputs, and shift operations at the inputs and outputs of the dataflow subgraph. The basic building blocks of the PCFU are the carry generator for each addition/subtraction supported and an output LUT for each subgraph output supported.

Increasing the number of outputs supported by the PCFU is a fairly straight forward process, which only requires adding an output LUT in parallel with the already existing output LUTs. None of the other structures in the PCFU are affected.

Supporting additional inputs is more complicated, since it involves increasing the size of the LUTs for the carry generators and the output LUTs. This is because the logical function for each bit depends on another boolean variable (the new input), which doubles the size of each truth table used to compute results.

Similar to increasing the number of inputs, increasing the number of adds that are supported doubles the size of the output LUTs, since the outputs are now a function of another carry-in signal. Beyond this, supporting more adds requires an additional set of carry propagate LUTs, which are dependent on the inputs and all previous carry-in signals. This means that the added carry-propagate LUT is larger than all the previous carry propagate LUTs combined. A new carry generation tree lies directly on the critical path of the PCFU, as well.

Supporting shift operations within subgraphs is desirable, but infeasible on the PCFU. Allowing shifts would make each output bit a function of every input bit, instead of the small number of input bits in the proposed design. This would make the LUTs very large. However, separate shifters may be added at the inputs and/or the outputs of the PCFU to support shift operations at the inputs and outputs of the dataflow subgraphs. This would not change the size of the LUTs, but would lengthen the critical path of the PCFU.

Each of these vectors in the design space is explored in Section 4.

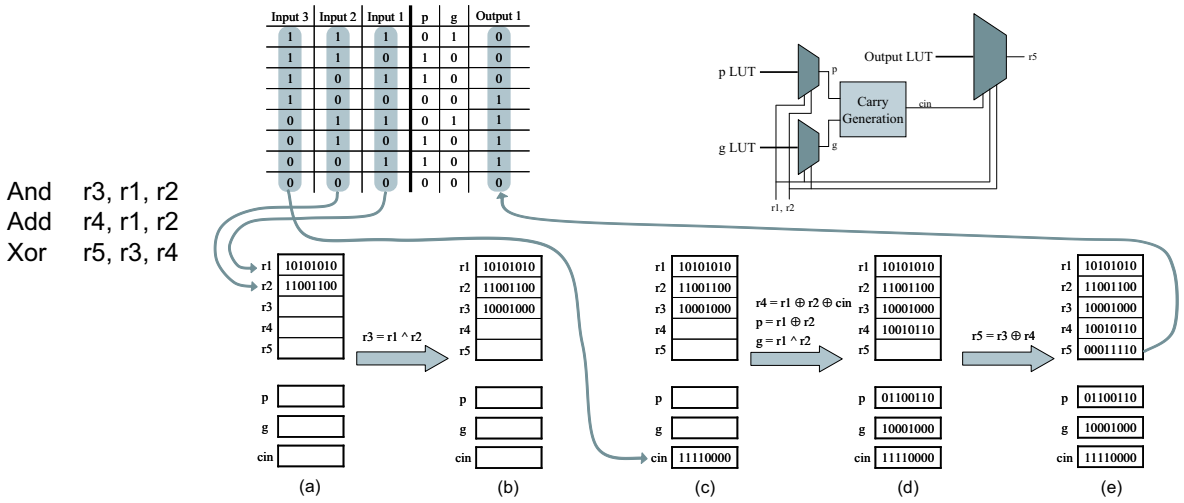

Figure 6: LUT entry generation example. Shown are the processing steps of the Meta-processor that compute the LUT entries to implement the function defined by the assembly code sequence on the left.

3.3 Function Generation - The Meta-Processor

In addition to the PCFU, transparent instruction set customization requires a method for generating control signals for the execution substrate. The PCFU control generator (also called the meta-processor) generates the LUT entries for the PCFU. This process is analogous to mapping logic onto FPGAs, but is far simpler since the computations to be mapped are stateless and well defined. For completeness, a summary of the meta-processor is given here, but a more thorough discussion is beyond the scope of this work.

The meta-processor has two main pieces, a meta-register file, and a meta-function unit. The meta-register file contains the LUT entry needed to generate the output value of that register, as opposed to the register value itself. Likewise, the meta-function unit operates on LUT entries as opposed to register values.

Figure 6 shows a simplified example of generating LUT entries for a dataflow subgraph. At the left of the figure is the subgraph to be mapped, and the right of the figure shows the meta-register file at various stages of the LUT generation. The PCFU being targeted in this example can support two inputs and one addition operation, thus each output bit is a function of three input bits (the two inputs, and the carry signal from the addition). The three input bits imply that each LUT entry is eight bits in size, as $2^3 = 8$.

Subgraph mapping begins by looking at the And instruction in Figure 6. Initially, the two sources, r1 and r2, have no LUT value in the meta-register file. Since we are interested in computing the output given all possible values for the inputs, meta-registers r1 and r2 are assigned LUT entries which ensure that all possible combinations of one and zero interact (Basically, the corresponding input columns of a regular truth table). This step essentially begins to construct a truth table. Since r1 and r2 are live-ins, the LUT entries assigned to them correspond to the input columns of a hypothetical truth table that outputs to meta-register r3.

Now that r1 and r2 have values in the meta-register file, r3 is computed in the meta-function unit as the And of those two values, and is shown in Figure 6 (b). To reiterate, the value of r3 in the meta-register file defines how to compute r3 given any values of r1 and r2, exactly like a truth table.

The subgraph mapping then moves onto the Add instruction. The output of this instruction is dependent on r1, r2, and also on the carry signal generated during addition. Recall that the carry-in signal is treated as an input to the computation so that we can leverage fast carry generation hardware, and so that each output bit is not dependent on the value of lower order input bits. Since the carry signal of this Add is an input, we assign it a value in the meta-register that corresponds to a third input column of the hypothetical truth table. This step is shown in Figure 6 (c).

Now that the inputs are defined, the LUT entries for outputs of the Add operation must be computed. Recall that a bitwise add operation is $r1_i \oplus r2_i \oplus cin_{i-1}$. Using the LUT entries for r1, r2, and cin, the meta-function unit computes this and places the result in the meta-register r4. Note that because cin is defined as an input, the meta-function unit did not need to perform an addition, only two exclusive-ors, which makes the hardware very fast and simple.

Unlike the registers, the carry signal is generated using a carry propagation network. Thus, it is necessary to define p and g, the inputs to the carry network. Recall that $p = a \oplus b$ and $g = a \wedge b$. Using the values in the meta-register file for r1 and r2, the meta-function unit simply computes the LUT values for p and g, and stores them into the meta-register value. The meta-register file state after this step is shown in Figure 6 (d).

Mapping continues with the Xor instruction. As with the previous two instructions, we first check that all the inputs are defined. In this case, r3 and r4 already have valid values in the meta-register file and this instruction does not generate a carry signal. Next, the meta-function unit computes the LUT value of r5, by simply Xoring the LUT values of r3 and r4. This is shown in Figure 6 (e). Now that the output, r5, is defined as a function of r1, r2, and carry, we store the meta-register file values of r5, p, and g as the configuration of this subgraph. This example shows how the meta-processor is able to perform logic mapping of a dataflow subgraph onto the PCFU substrate without the typical complexity associated with FPGA mapping.

4. EXPLORING THE PCFU DESIGN SPACE

The purpose of this paper is to evaluate the different tradeoffs involved in designing a PCFU for subgraph acceleration. The designs are evaluated using latency of the PCFU, die area consumed by the PCFU, as well as performance improvement of the PCFU-augmented processor.

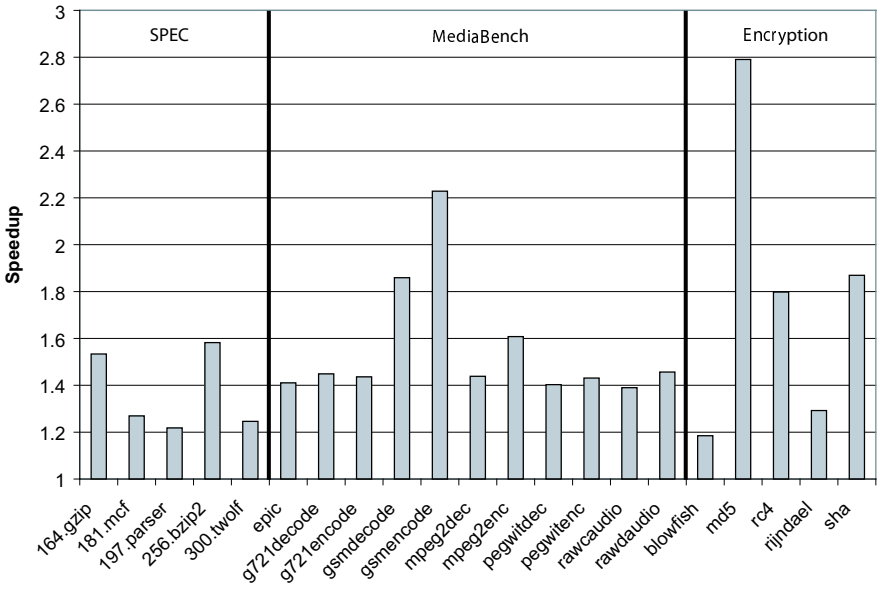

Figure 7: Effectiveness of the baseline 4-input, 2-output PCFU design.

Evaluation of the performance improvement achieved using PCFUs was done using a version of the Trimaran compiler [26] ported to the ARM instruction set. The compiler was augmented with a parameterized subgraph matching engine, which allowed us to easily change the types of subgraphs selected based on the characteristics of the underlying hardware. The specific algorithms used in subgraph identification are beyond the scope of this paper and described in [11]. After subgraphs selection and function outlining, binaries were created using the GNU assembler/linker, and simulated using SimpleScalar ARM [4]. The simulator was configured to model an ARM 926EJ-S processor [1], a popular single-issue embedded core with a five stage pipeline.

In order to determine the latency and area properties of the PCFUs, several designs were synthesized, including place-and-route. The designs were implemented using Synopsys tools with a standard cell library in 0.13μ. The critical path latencies are reported, as well as the die areas given both in mm^2 and as the percentage area of an ARM926EJ-S core without caches. Note that not every design simulated, was synthesized, since creating and verifying HDL for the PCFUs is a very time consuming process.

We chose to evaluate our designs using benchmarks from the SPECint2000 and MediaBench [19] benchmark suites, as well as several encryption kernels. Full runs of each benchmark using the training input set were performed. Applications from the two benchmark suites that do not appear were omitted either due to very long runtime (in the case of 254.gap), or limitations in the compiler infrastructure. Performance is reported as the ratio of total cycles to run the binary *without* subgraphs identified compared to the number of cycles needed to run the binary *with* subgraphs identified, and is referred to as speedup. It is important to make this distinction, because binaries with subgraphs identified can still be run on hardware without a PCFU; however, the code size expansion would unnecessarily harm the baseline.

Baseline Design. In order to explore the various dimensions of the PCFU design space, we first define a starting point. Previous work [9] has shown that a subgraph execution unit with 4 inputs and 2 outputs, supporting two adds is a reasonable design choice. As such, this is baseline for our evaluation. The design of this PCFU can be seen in Figure 4.

The speedups attained using this baseline design are presented in Figure 7 for the three groups of benchmarks. Unless otherwise noted, simulation was done assuming that the PCFU requires one cycle to execute the subgraph and does not affect the cycle time of the processor. The main point to take from this figure is the magnitude of the bars. On average, a speedup of 1.62 over the baseline processor is observed, with a maximum of 2.79. This shows that transparent instruction set customization using a PCFU is an effective way to improve the performance of embedded processors. Also note that the speedup varies a great deal from application to application. This is correlated to the size of the computation subgraphs available for execution on the PCFU. Since subgraphs are bounded by memory operations, applications that perform a large amount of computation (especially logic operations) between memory accesses benefit the most.

Design Space Parameters. The PCFU design space is evaluated along three independent axes: number of inputs/outputs, number of additions, and support for shift operations. The number of additions specifies the number of carry chains that the PCFU implements. In varying the number of additions, it is also possible to emulate PCFUs with larger number of additions by connecting smaller PCFUs in series, e.g., a 2-adder PCFU can be emulated by connecting two 1-adder PCFUs in series. Shift operations are not supported directly by the PCFU, but by creating hybrid accelerator substrates consisting of PCFUs and shifters.

Number of Inputs/Outputs. The first design space parameter is the effect that the number of inputs and outputs has on the system. These parameters are very important, as they have a strong impact on the types of graphs that can be executed on the PCFU. The number of inputs/outputs in the PCFU also has an effect on the register file since each of the inputs/outputs must be read from or written to it. This means that a large number of inputs or outputs requires a larger register file, multiple cycles to read and write results, or "shadow register files" to increase the operand bandwidth without increasing the latency. All of these options carry overheads.

Design	Latency (ns)	Area (mm^2)	Area (% of ARM926EJ-S Core)
2 In, 1 Out, 2 Adds, No Shifts	3.03	0.052	2.3
2 In, 2 Out, 2 Adds, No Shifts	2.66	0.056	2.5
3 In, 1 Out, 2 Adds, No Shifts	3.24	0.068	3.1
3 In, 2 Out, 2 Adds, No Shifts	3.32	0.100	4.5
4 In, 1 Out, 2 Adds, No Shifts	3.79	0.134	6.1
4 In, 2 Out, 2 Adds, No Shifts	4.20	0.171	7.7
4 In, 3 Out, 2 Adds, No Shifts	4.57	0.230	10.4
5 In, 1 Out, 2 Adds, No Shifts	5.25	0.214	9.7
5 In, 2 Out, 2 Adds, No Shifts	5.30	0.306	13.9
5 In, 3 Out, 2 Adds, No Shifts	5.40	0.397	18.0
6 In, 1 Out, 2 Adds, No Shifts	5.47	0.465	21.1
6 In, 2 Out, 2 Adds, No Shifts	5.27	0.600	27.2
6 In, 3 Out, 2 Adds, No Shifts	5.87	0.787	35.8

Table 1: Synthesis results for PCFU designs with varying numbers of inputs and outputs.

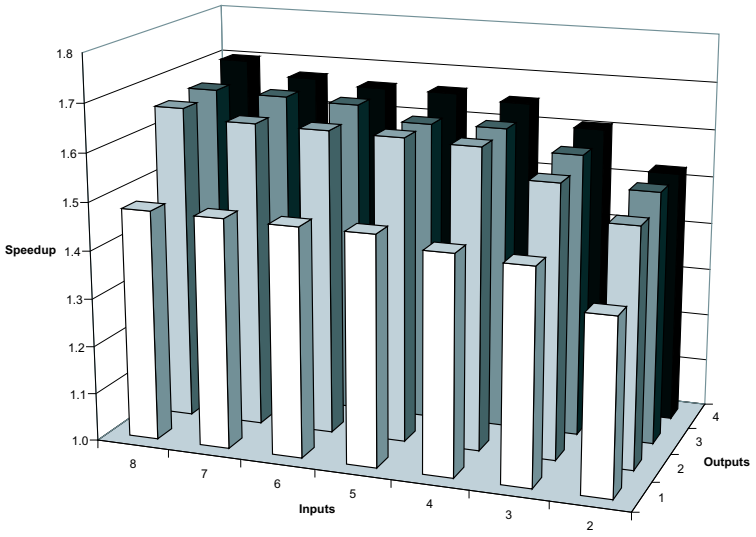

Figure 8: Effectiveness of PCFU designs with varying numbers of inputs and outputs.

From the perspective of PCFU design, the number of inputs must be carefully controlled. Increasing the number of inputs by one means that each output bit is the function of another binary variable. This essentially doubles the size of each LUT in the design. Aside from the exponential increase in area, this LUT size increase also causes the overall latency of the PCFU to increase as well.

The number of inputs and outputs also plays a role in control generation for the PCFUs. Recall that in the PCFU control generator, the meta-register file is responsible for generating the LUT entries. For every additional input, the size of the LUTs double, meaning that the size of the meta-register file also doubles. Increasing the number of outputs is less critical for control generation, as all the LUT configurations of the live-out values are stored in the meta-register file. That is, the baseline design already supports multiple outputs, so very little additional complexity is needed to support them.

The effects of adding inputs and outputs to a PCFU can be seen in the synthesis results in Table 1. In this table, each PCFU design is represented by a 4-tuple specifying the number of inputs, the number of outputs, the number of supported additions, and what shift values (if any) are supported. Initially increasing the number of inputs has a small effect on the total PCFU area and latency; moving from two to three inputs increases the area by 0.016 mm^2 and the latency by 0.21 ns. However, the exponential increase in LUT size quickly begins to dominate. For example, moving from five inputs to six causes an increase in latency of 0.22 ns, and the die area more than doubles, going from 0.214 to 0.465 mm^2. This demonstrates that the number of inputs must be carefully balanced in the design of a PCFU.

Increasing the number of outputs is not as critical of an issue in terms of PCFU design. Each additional output from the PCFU requires an additional function LUT to compute the result using the inputs and the carry-in signal(s). No additional LUTs are needed beyond that, and none of the other structures change in size. This essentially means that adding an output should increase the area of the PCFU in a roughly linear fashion, and have a small or no effect on the latency. These trends can be seen in Table 1. Moving from four inputs and two outputs to four inputs and three outputs increases the area by 0.059 mm^2, and the latency by 0.37 ns. The non-linearity is due to increased MUX sizes and certain signals (e.g., the carry-ins) having to drive a larger number of cells.

One confusing trend in Table 1, is that not all of the synthe-

Design	Latency (ns)	Area (mm^2)	Area (% of ARM926EJ-S Core)
4 In, 2 Out, 0 Adds, No Shifts	0.62	0.042	1.9
4 In, 2 Out, 1 Adds, No Shifts	2.44	0.095	4.3
4 In, 2 Out, 2 Adds, No Shifts	4.20	0.171	7.7
4 In, 2 Out, 3 Adds, No Shifts	5.78	0.361	16.4
4 In, 1 Out, 2 Adds, No Shifts	3.79	0.134	6.1
4 In, 1 Out, 2 (1-1) Adds, No Shifts	3.77	0.116	5.3
4 In, 1 Out, 3 Adds, No Shifts	5.82	0.274	12.4
4 In, 1 Out, 3 (2-1) Adds, No Shifts	6.50	0.212	9.6
4 In, 1 Out, 3 (1-1-1) Adds, No Shifts	6.10	0.180	8.1

Table 2: Synthesis results for PCFU designs with varying numbers of additions supported.

sis results agree with what was predicted. For example, moving from six inputs and one output, to two outputs, and to three outputs caused the area of the PCFU to grow super-linearly. Adding more outputs also caused the latency to change a great deal, despite the fact that the critical path has the same number of logic levels in all three designs. These observations are an unfortunate side-effect of heuristics used in the synthesis tools, and are beyond our control.

Figure 8 shows the average speedup across our benchmark suite when varying the number of inputs and outputs allowed in the PCFU. When adding inputs and outputs for this experiment, we assumed that reading the inputs and writing the results back to the register file each took one cycle regardless of the number of inputs/outputs. This was done to determine how well the compiler can take advantage of the inputs/outputs available to it, independent of other hardware restrictions.

The main point to take away from Figure 8 is that four inputs and two outputs seems to be the point of diminishing return. That is, increasing the number of inputs beyond four or the number of outputs beyond two does not substantially improve the resulting performance. Four inputs and two outputs are necessary to support the most important computation subgraphs in our set of applications. Conversely, reducing the number of inputs to three or two drops the speedup to 1.55 and 1.49, respectively. Reducing the number of outputs to one drops the speedup to 1.45. While these drops may not seem significant, the average is hiding the fact that the speedup of some benchmarks drop significantly, while other benchmarks are relatively unaffected. For example, the speedup of MD5 dropped 78% moving from four inputs to two, and the speedup of EPIC fell 58% moving from two to one output.

Number of Additions. As with the number of inputs, the number of additions supported by the PCFU must also be carefully constrained. Supporting an additional add operation would necessitate creating two new LUTs and a Kogge-Stone tree to calculate the Propagate and Generate signals for that add. These new P-G LUTs will be a function of each input and all previous carry-in signals, meaning that their size will be twice as large as the previous largest P-G LUTs. Beyond the additional LUTs, the size of each function LUT doubles, since each output is also a function of this new carry signal. This increases the area of the PCFU and lengthens the critical path much more quickly than simply adding inputs or outputs.

Adding the new P-G LUTs means that control will have to be generated for them as well. This entails adding three new registers to the meta-register file for the new P-LUT, G-LUT, and carry LUT. Space for the added control of these LUTs must be added to the configuration cache as well as the meta-register file; however, overall latency of the control generation is not affected, and area increases linearly with the number of adds.

The top portion of Table 2 shows the synthesis results when varying the number of adds supported. Note how increasing the number of adds supported more than doubles the die area of the PCFU in most cases. The latency of the PCFUs also increases a great deal, since the critical path now runs through an additional carry generator and larger function LUTs. For these reasons, it is important to limit the number of additions supported in the PCFU.

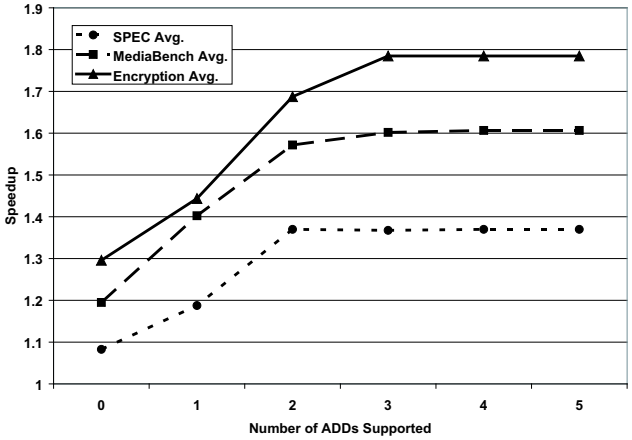

Figure 9: Effectiveness of PCFU designs with varying numbers of additions supported.

One trick that can be played to reduce the area overhead of supporting many additions, is to compose a larger PCFU out of smaller ones. For example, a two-adder PCFU could be created by serially merging two one-adder PCFUs with a MUX. The MUX is used to select a subset input values from the result of the first PCFU and the subgraph inputs. Using a MUX to control the number of inputs prevents the exponential growth of the LUTs, at the cost of potentially supporting fewer subgraphs. Our experiments have shown that the subgraphs not supported never occur in any of the applications tested, though, making this a good trade off.

The lower portion of Table 2 shows the synthesis results of these composite PCFUs. Next to the number of adds in the design column, parentheses occur indicating the formation of the composite PCFU. For example, "(1-1-1)" indicates a three-add PCFU designed as three one-add PCFUs chained together serially. This table clearly shows how creating PCFUs as a composite of smaller PCFUs is an effective way to reduce the area incurred by supporting more add operations. The variations in latency reflect the trade off of the critical path traveling though fewer large LUTs (when the PCFU is not composite) versus the critical path traveling through more small LUTs (when the PCFU is composed of 1-adder PCFUs).

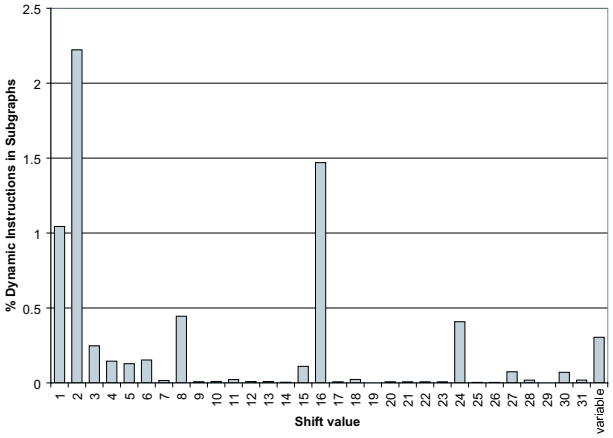

Figure 10: Distribution of shift values within subgraphs.

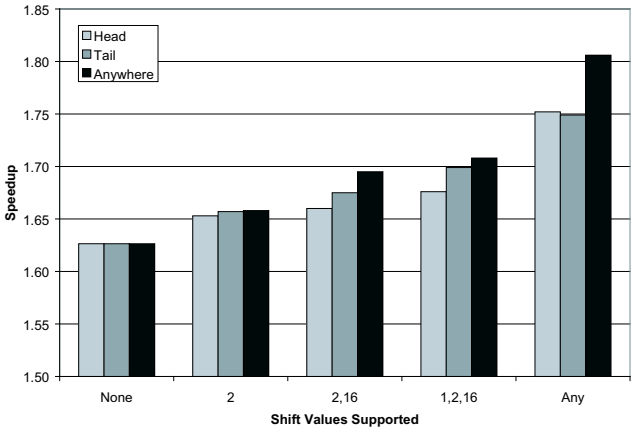

Figure 11: Effectiveness of PCFU designs with varying types of shifts supported.

Figure 9 shows the effect the number of adds supported has on the speedups attainable by the PCFU system. This figure shows that significant speedup gains can be achieved by moving from zero to one to two adds. This trend highlights the prevalence of add instructions in our applications. Moving beyond two adds yields quite limited results, though. This is mainly because computation subgraphs are limited by memory operations. The amount of computation done between memory accesses typically does not encompass more than two add instructions. This is not as true in the encryption-style applications which contain a relatively large amount of computation between memory accesses.

Support for Shifts. Another design parameter explored was the addition of shift operations in subgraphs supported by the PCFU. In the general case, it is impossible to support shifts at arbitrary places within subgraphs. Doing so would make each output bit a function of each input bit creating function LUTs the size of $2^{total\ input\ bits}$. However, it is feasible to place a shifter at the inputs or outputs of the PCFU. This would allow support for shifts in subgraphs provided there was no computation before the shift (or after the shift, in the case of shifts at the outputs) performed on the PCFU. Adding these shifters would not affect the size of the LUTs or the internal PCFU structure, but would require some additional MUXes at the inputs (and/or outputs). The downside is that the shifters would appear on the critical path.

In terms of control generation, allowing shift capabilities in the PCFU involves adding few bits in the configuration to specify the shift value and direction for each input and/or output. Though this does increase the critical path of control generation slightly, it is a trivial extension. Allowing shift operations increases the size of the configuration size as the log of the number of shift values supported, thus the area overhead increases at that rate as well.

To analyze the effectiveness of allowing shifts within subgraphs, we first examined the types of shifts that could potentially be used. Figure 10 shows the types of shifts that appeared in important subgraphs. That is, if the compiler allowed shifts to appear anywhere in subgraphs, this graph shows the types of shifts that were selected for subgraph inclusion; the shifts that would be useful to support in the PCFU. The horizontal axis in this figure is the constant value of the shift instruction (or variable in the case that the operation did not use a compile time constant), and the vertical axis shows the percentage of dynamic instructions averaged across the benchmarks. As an example, around 1.5% of dynamic instructions in our benchmarks were shifts by the constant 16, which would have appeared in subgraphs provided the PCFU supported them.

Figure 10 shows that the shifts useful in subgraphs are dominated by a relatively small number of constants. As would be expected, two is the most common shift value, since it is frequently used for address calculation in the 32-bit ARM architecture. One key trend in this figure is that variable shifts (the far right bar) were quite infrequent. This is a good sign, as supporting variable shifts in the PCFU generally requires larger area. Conversely, supporting shifts by a small number of constants merely requires a small bit of wiring and an additional MUX.

Using this information, several designs supporting shifts were synthesized; the results are in Table 3. In order to limit the area overhead associated with barrel shifters, we used logarithmic shifter in the synthesized designs. The chart shows using logarithmic shifters generally caused the latency to increase a great deal when supporting additional constants, however, it did not incur a substantial area gain. For example, supporting any shift value was nearly the same area as supporting the three most frequent constants. Also note that supporting shift values at the tail of subgraphs was less costly than at the head; this is intuitive, as there are only two outputs compared with four inputs.

Speedup results for these designs are in Figure 11. For each set of shift values supported, there are three bars displayed: one for when shifts are supported only at the inputs (or head of the subgraph), one for shifts only supported at the outputs, and one for shifts supported anywhere within the subgraph. Although the last bar is not supported by the PCFU, it provides a comparison as to how well shifts at the inputs or outputs meet the overall need for shifts in subgraphs.

In general, providing capabilities for a small number of shift values does provide a substantial amount of speedup. For example, allowing shifts by 1, 2, or 16 at the outputs improved speedups by 7% over the baseline design. Providing shifts at the end of subgraphs is slightly more beneficial than at the head of subgraphs, again, because many shift-by-two ops are used for address calculation. The address calculation feeds memory operations, which must appear outside the subgraph. Allowing shifts anywhere in subgraphs does offer significant benefit over restricting shifts to the fringes, but most of the gains from adding shifts can be attained by only adding them at the inputs and outputs.

Cost vs. Performance for all Designs. To summarize the trade offs of the design space, we combined the synthesis and simulation

Design	Latency (ns)	Area (mm^2)	Area (% of ARM926EJ-S Core)
4 In, 2 Out, 2 Adds, No Shifts	4.20	0.171	7.7
4 In, 2 Out, 2 Adds, 2 at Inputs	4.64	0.213	9.6
4 In, 2 Out, 2 Adds, 1, 2, 16 at Inputs	4.86	0.224	10.1
4 In, 2 Out, 2 Adds, Any at Inputs	5.22	0.224	10.1
4 In, 2 Out, 2 Adds, Any at Outputs	5.15	0.201	9.1

Table 3: Synthesis results for PCFU designs with varying types of shifts supported.

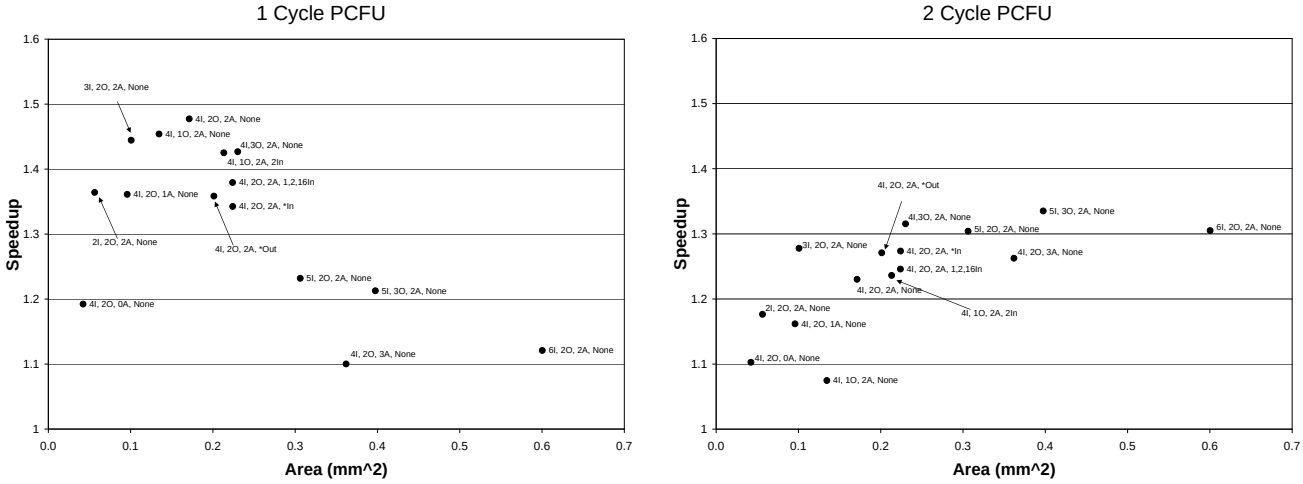

Figure 12: The cost/performance trade off across various PCFU design points.

results in Figure 12. The horizontal axis has the cost of a design, and the vertical axis shows speedup attained using that design. The speedup numbers in this figure were scaled to reflect cycle time increases. The ARM926EJ-S typically runs at 250 MHz using a standard synthesis flow in 0.13μ technology [1]. In the left portion of Figure 12, if a PCFU design could not meet the $4ns$ cycle time, then the entire processor was slowed to the frequency of the PCFU. For example, if a PCFU had a critical path of $8ns$, then we assumed two cycles of the baseline machine could occur in the same time as one cycle of the machine using that PCFU. The right graph in Figure 12 performs the same scaling, but assumes that the PCFU takes two cycles to execute (e.g., the PCFU is pipelined). This allows us to compare the cycle time versus subgraphs supported trade offs for PCFUs in the context of processors with higher clock frequencies.

The main observation to take from the 1-cycle PCFU graph is that to offset increasing the clock cycle, it is imperative to support many more subgraphs. Only one of the Pareto-optimal design points (4I, 2O, 2A, None) increased the clock cycle, and that was only by $0.2ns$. Figure 8 shows the increased number of subgraphs, by moving to four inputs/two outputs, needed to justify slowing the clock cycle. When clock cycle is taken into account, there generally are not enough large dataflow subgraphs to justify the PCFU designs targeting them.

Under the assumption of a two cycle PCFU, it is a different story, however. Assuming the PCFU takes two cycles to execute implies that none of the PCFU designs extend the clock cycle. This enables the benefits of supporting the larger subgraphs to show themselves. For example, the 5I, 3O, 2A, None design point is Pareto optimal under the two-cycle assumption. Despite this, using two cycles to support larger subgraphs did not outperform the one-cycle PCFU designs that target smaller subgraphs.

5. CONCLUSION

In this paper, we have explored the design of Programmable Carry Function Units, a hardware substrate for executing acyclic dataflow subgraphs. Several different design parameters were examined, ranging from the number of subgraph inputs/outputs supported, to the number of addition/subtractions supported, to the the types of shifts allowed. Evaluation of these designs was done with simulation as well as synthesis, to fully evaluate the hardware trade offs in the context of the ARM926EJ-S embedded processor. Overall, we have shown that implementing a carefully designed PCFU can provide substantial speedups (1.47 on average) over a baseline embedded processor for relatively little area overhead. We also demonstrated that non-pipelined PCFU designs that support more subgraphs, but increase the cycle time of the processor, are generally not wise design points. However, that conclusion is reversed in the case of pipelined PCFU designs.

6. ACKNOWLEDGMENTS

Many thanks go to Stuart Biles, Daryl Bradley, and John Biggs at ARM who helped us with the synthesis flow. Additional gratitude goes to the anonymous referees who provided excellent feedback on this work. This research was supported by ARM Limited, the National Science Foundation grant CCF-0347411, and equipment donated by Hewlett-Packard and Intel Corporation.

7. REFERENCES

[1] ARM Ltd. *ARM926EJ-S Technical Reference Manual*, Jan. 2004. http://www.arm.com/pdfs/DDI0198D_926_TRM.pdf.

[2] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proc. of the 40th Design Automation Conference*, pages 256–261, June 2003.

[3] P. M. Athanas and H. S. Silverman. Processor reconfiguration through instruction set metamorphosis. *IEEE Computer*, 26(3):11–18, 1993.

[4] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Transactions on Computers*, 35(2):59–67, Feb. 2002.

[5] A. Bracy, P. Prahlad, and A. Roth. Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 18–29, Dec. 2004.

[6] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Trans. Comput.*, C-31(3):260–264, 1982.

[7] P. Brisk et al. Instruction generation and regularity extraction for reconfigurable processors. In *Proc. of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 262–269, 2002.

[8] J. E. Carrillo and P. Chow. The effect of reconfigurable units in superscalar processors. In *Proc. of the 9th ACM Symposium on Field Programmable Gate Arrays*, pages 141–150. ACM Press, 2001.

[9] N. Clark et al. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 30–40, Dec. 2004.

[10] N. Clark et al. An architecture framework for transparent instruction set customization in embedded processors. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 272–283, June 2005.

[11] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 129–140, Dec. 2003.

[12] D. Goodwin and D. Petkov. Automatic generation of application specific processors. In *Proc. of the 2003 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 137–147, 2003.

[13] J. R. Hauser and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In *Proc. of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–21, Apr. 1997.

[14] S. Hu and J. Smith. Using dynamic binary translation to fuse dependent instructions. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 213–226, 2004.

[15] I. Huang. *Co-Synthesis of Instruction Sets and Microarchitectures*. PhD thesis, University of Southern California, 1994.

[16] Q. Jacobson and J. E. Smith. Instruction pre-processing in trace processors. In *Proc. of the 5th International Symposium on High-Performance Computer Architecture*, pages 125–133, 1999.

[17] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Comput.*, C-22(8):786–793, 1973.

[18] I. Koren. *Computer Arithmetic Algorithms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[19] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, 1997.

[20] S. J. Patel and S. S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Trans. Comput.*, 50(6):590–608, 2001.

[21] J. Phillips and S. Vassiliadis. High-performance 3-1 interlock collapsing alu's. *IEEE Trans. Comput.*, 43(3):257–268, 1994.

[22] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable function units. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 172–180, Dec. 1994.

[23] P. Sassone and D. S. Wills. Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 7–17, Dec. 2004.

[24] Y. Sazeides, S. Vassiliadis, and J. E. Smith. The performance potential of data dependence speculation & collapsing. In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 238–247, 1996.

[25] F. Sun et al. Synthesis of custom processors based on extensible platforms. In *Proc. of the 2002 International Conference on Computer Aided Design*, pages 641–648, Nov. 2002.

[26] Trimaran. An infrastructure for research in ILP, 2000. http://www.trimaran.org.

[27] M. J. Wirthlin and B. L. Hutchings. DISC: The dynamic instruction set computer. In *Proc. of the 1995 Field Programmable Gate Arrays for Fast Board Development and Reconfigurable Computing*, pages 92–103, 1995.

[28] Z. A. Ye et al. CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proc. of the 27th Annual International Symposium on Computer Architecture*, pages 225–235, 2000.

[29] S. Yehia and O. Temam. From sequences of dependent instructions to functions: An approach for improving performance without ilp or speculation. In *Proc. of the 31th Annual International Symposium on Computer Architecture*, pages 238–249, June 2004.

[30] P. Yu and T. Mitra. Characterizing embedded applications for instruction-set extensible processors. In *Proc. of the 41st Design Automation Conference*, pages 723–728, June 2004.