Reconciling Specialization and Flexibility Through Compound Circuits

Sami Yehia, Sylvain Girbal
Thales Research & Technology
Embedded Systems Lab, France
{Sami.Yehia, Sylvain.Girbal}@thalesgroup.com

Hugues Berry, Olivier Temam
Alchemy Project
INRIA Saclay, France
{Hugues.Berry, Olivier.Temam}@inria.fr

Abstract

While parallelism and multi-cores are receiving much attention as a major scalability path, customization is another, orthogonal and complementary, scalability path which can target not easily parallelizable programs or program sections. The key assets of customization are cost and power efficiency. *The key limitation of customization is* flexibility. *However, we argue that there is no perfect balance between efficiency and flexibility, each system vendor may want to strike a different such balance. In this article, we present a method for achieving any desired balance between flexibility and efficiency by automatically combining any set of individual customization circuits into a larger* compound *circuit. This circuit is significantly more cost efficient than the simple union of all target circuits, and is configurable to behave as any of the target circuits, while avoiding the routing and configuration cost overhead of FPGAs. The more individual circuits are included, the larger the number of applications which can potentially benefit from this compound customization circuit, realizing flexibility at a minimal cost. Moreover, we observe that the compound circuit cost does not increase in proportion to the number of target applications, due to the wide range of common data-flow and control-flow patterns in programs. Currently, the target individual circuits correspond to loops, like most accelerators in embedded systems, but the aggregation method can accommodate circuits of any size. Using the UTDSP benchmarks and accelerators coupled with an embedded PowerPC405 processor, we show that this approach can yield an average performance improvement of 2.97, while the corresponding synthesized aggregate accelerator is 3 time smaller than the sum of individual accelerators for each target benchmark.*

1 Introduction and Related Work

The current clock frequency limitations are forcing architects to leverage additional transistors rather than faster transistors, i.e., space rather than speed. With the current focus on multi-cores, it is all too easy to forget that there is at least one alternative scalability path to parallelization capable of leveraging additional on-chip space/resources: *customization*; and this scalability path deserves at least as much attention as parallelization. Customization has several strong assets: specialized circuits are significantly more power efficient and significantly less costly (chip footprint) than general-purpose processors for performing the same task [8]. Furthermore, this scalability path is *complementary* to parallelization, not a competing option; after an application has been parallelized, the behavior of each parallel section, or remaining sequential sections can further benefit from customization; this hybrid approach is typically used in many high-performance embedded systems [18].

The real challenge of customization is *flexibility*. The ultimate specialized circuit is the whole program mapped to a chip, as proposed by Budiu et al. in [8], or Schreiber et al. in [23]. Assuming there are enough resources for whole-program mapping, this chip is very efficient but it can only be used for a single fixed program. At the other end of the flexibility spectrum lays the far less efficient general-purpose processor. In-between, several approaches attempt to strike the right balance between flexibility and efficiency. FPGAs are flexible but less efficient than specialized circuits, consuming more power [20, 24] and dedicating up to 90% of the on-chip space for the routing networks; at the same time, they are difficult to program, requiring steps akin to circuit synthesis, and increasingly come combined with a general-purpose processor for higher ease of use [3], especially in interacting with the rest of the system. ASIPs (Application-Specific Integrated Processors) combine general-purpose processors with manually designed specialized circuits, because they recognize that many programs use similar, recurring patterns, and because they want to achieve higher efficiency than FPGAs and do not necessarily need their generality, especially for targeted domains. ASIPs either target fine-grain acceleration such as Tensilica [2] and FITS [9], or coarser loop-based acceleration such as ARM OptiMODE [12] and other programmable loop accelerators such as in [10, 19]; more recently, Clark et al. [13] investigated such a loop-based accelerator for a large range of benchmarks. TRIPS/GPAs [22] is another efficiency/flexibility tradeoff which spatially expands the program on a grid of ALUs, so that it is significantly more power efficient than traditional general-purpose processors, without sacrificing flexibility; it still remains significantly less efficient though than specialized circuits.

The first premise of this study is that there is no such thing as a *right* balance between flexibility and efficiency, it is a vendor-specific issue which typically depends on the target market. Therefore, in order to expand the reach of customization, we rather focus on striking the *desired* (as opposed to the *right*) balance between flexibility and efficiency. The second premise of this study is that an approach is scalable if it can almost seamlessly leverage additional on-chip resources to improve flexibility and/or efficiency.

The degree of flexibility of a given customization/acceleration circuit can be measured by the number of applications which can potentially benefit from that circuit. In order to adjust this degree as needed, we focus on designing customization circuits that can accomodate a variable, either small or large, number of applications. For that purpose, we build *compound* circuits which are aggregate of the individual customization circuits required for each of the target applications. The compound circuits are *automatically* derived from the set of individual customization circuits. However, these compound circuits are not simply the union of the individual customization circuits: we leverage common data-flow and control-flow patterns among programs to preserve the cost benefit of customization; in fact, the compound circuits are significantly denser than the union of the individual customization circuits thanks to these commonality properties. These circuits are *configurable* as configuration logic (muxes) are introduced when aggregating circuits. By properly configuring these muxes, it is possible to make the compound circuit behave as any of the original individual circuits. However, this configurable logic and the additional links are kept to a minimum; as a result, there is no generic routing network such as in traditional FPGA, or in coarser grain configurable accelerator like in [11, 27, 5] where all possible connections between operators have to be implemented with an expensive network of switches or multiplexers. Previous research proposed to merge more than one hardware datapath to map several specialized instructions on the same hardware such as in [7, 28, 21] in the context of ASIP design but none of them leverage both data flow and control flow to allow full loop accelerator aggregation. Furthermore, our approach includes state nodes such as registers and memory buffers in the aggregate data paths while other approaches essentially target reuse of operators.

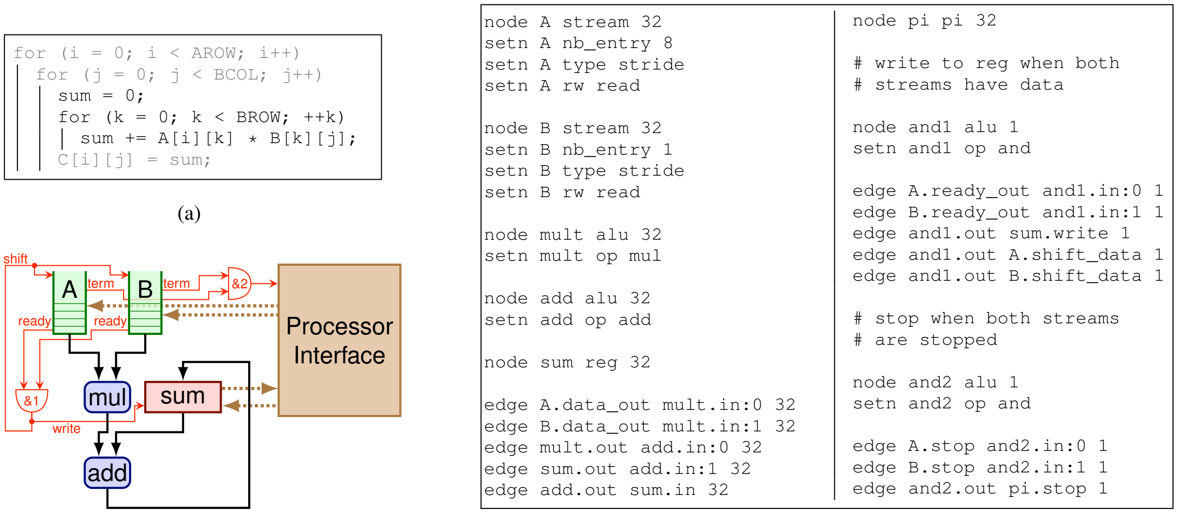
We illustrate our approach with a benchmark suite for DSPs; we pick one core routine from each of 9 benchmarks and automatically combine them all into a single circuit which is 3 times (66%) smaller than the sum of all circuit areas (0.3 mm^2 vs. 0.9 mm^2), and increases individual circuits critical path by a maximum 37% (6.08 ns vs. 4.41 ns). We also create several combinations of individual circuits, the average gain in area and increase in critical path are respectively 43% and 21%.

Previous research has shown how to automatically convert program parts into an intermediate representation which can later be translated into a circuit [20, 13]. In this article, we start from individual circuits already converted from target program sections into an intermediate representation composed of a few elementary operators (ALUs, registers, memory buffers), and we focus on the creation of compound circuits based on multiple different applications. Like many accelerators for embedded systems, our circuits are currently focused on program sections corresponding to loops, but like [20], the generic nature of the intermediate representation allows to target any code section. We give particular attention to the interaction with memory in order to ensure high throughput; in theory, the performance of customization circuits should only be limited by the memory bandwidth.

While accelerators converted from programs are likely to be significantly less efficient than manually tuned accelerators, increasingly stringent time-to-market constraints, coupled with a growing number of applications per embedded system (e.g., smartphones) are calling for more streamlined methods for deriving multi-objective accelerators. Such an approach is in fact complementary with later manually tuning accelerators or accelerator parts: for new embedded systems, the key is to hit the market first; later on, as volumes ramp up and the longer product lifespan is confirmed, manual tuning for even greater cost and power efficiency can take place. Finally, while this study is more targeted towards embedded systems, the general-purpose market is progressively warming up to the idea of adding specialized features as a way to further improve and differentiate chips, even at the cost of market fragmentation, as illustrated by Intel's future Larrabee chip.

2 Overview

Let us consider the example of an embedded processor targeted to several signal processing tasks. Such tasks often include computing-intensive kernels such as convolution filters, matrix operations and fast Fourier transform (FFT). We want to create circuits (accelerators) for each such task where the processor could offload the corresponding computations in order to achieve performance and power gains. Rather than individually building each circuit and implementing them all on the processor, or using an FPGA co-



```
for (i = 0; i < AROW; i++)
  for (j = 0; j < BCOL; j++)
    sum = 0;
    for (k = 0; k < BROW; ++k)
      sum += A[i][k] * B[k][j];
    C[i][j] = sum;
```

(a)

(b)

```
node A stream 32                  node pi pi 32
setn A nb_entry 8
setn A type stride                # write to reg when both
setn A rw read                    # streams have data

node B stream 32                  node and1 alu 1
setn B nb_entry 1                 setn and1 op and
setn B type stride
setn B rw read                    edge A.ready_out and1.in:0 1
                                  edge B.ready_out and1.in:1 1
node mult alu 32                  edge and1.out sum.write 1
setn mult op mul                  edge and1.out A.shift_data 1
                                  edge and1.out B.shift_data 1
node add alu 32
setn add op add                   # stop when both streams
                                  # are stopped
node sum reg 32
                                  node and2 alu 1
edge A.data_out mult.in:0 32      setn and2 op and
edge B.data_out mult.in:1 32
edge mult.out add.in:0 32         edge A.stop and2.in:0 1
edge sum.out add.in:1 32          edge B.stop and2.in:1 1
edge add.out sum.in 32            edge and2.out pi.stop 1
```

(c)

Figure 1. (a) Matrix multiply dot product, (b) Circuit, (c) Intermediate representation.

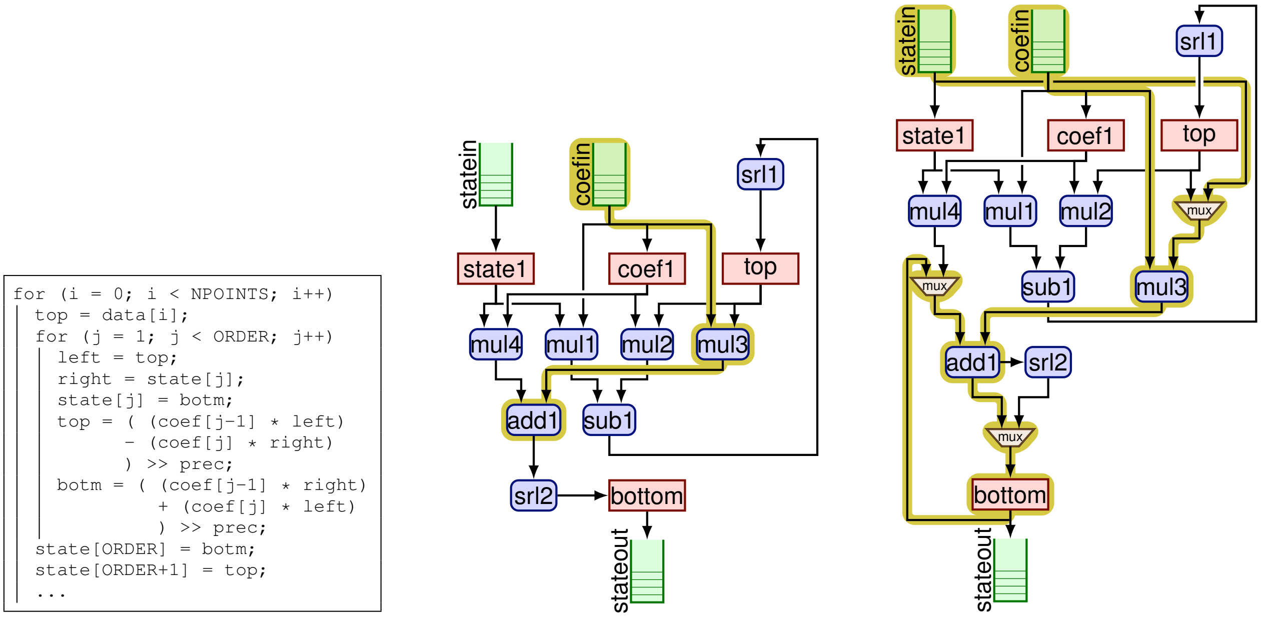
processor with more than necessary operators and routing network, we want to minimize chip real-estate without sacrificing performance or power by *combining* all three tasks into a single *compound* circuit. A key point is to design a systematic compounding process so that such circuits can be quickly generated for any combination of tasks.

Consider first the dot-product circuit of Figure 1(b), corresponding to the source code of Figure 1(a), which is at the core of many matrix operations. Using systematic code-to-circuit conversion rules in the spirit of [8], the code of Figure 1(a) is converted into our own circuit intermediate representation as shown in Figure 1(c), which is similar though simpler than traditional HDL languages. In this example, ALU operations are simply mapped to ALU operators, the variable sum is replaced with a register, and the two array references (A$[i]$ $[k]$ and B$[k]$ $[j]$) are mapped to input streams. The input streams are derived from Stream Buffers [16] and serve multiple purposes: to buffer incoming requests in order to feed the circuit with data up to every cycle if memory bandwidth allows, and also, for address generation and loop control. They are later described in more details, together with the conversion rules for more complex data-flow and control-flow constructs (e.g., indirect or reference-based addressing, conditional loops, if statements).

The aforementioned circuit elements correspond to the data-flow part of the code. The control part of the code takes the form of two 2-input 1-bit AND gates. One AND gate (&1) triggers the register write (sum), and shifts the two input streams (A, B) to the next data elements. The inputs of this AND gate are 1-bit state signals from the two input streams which indicate if the streams contain data; intuitively, this signal is triggered at each iteration for the code of Figure 1(a). Another AND (&2) gate is a termination signal sent back to the *processor interface*; the inputs are two 1-bit end signals provided by counters embedded in the stream control. The processor interface contains input latches to simultaneously parameterize all circuit state elements (registers, stream parameters, counters), and output latches to collect circuit results and control signals, e.g., the sum register and the stop signal in this example. The input and output streams interact with the memory bus through a dedicated memory interface which can process the interleaved requests of multiple streams; the memory interface is later described.

Figure 2(b) contains the circuit of the core computational part of a lattice filter (the control part of the circuit and the processor interface have been omitted), the code is shown in Figure 2(a). While the lattice filter and dot-product codes are not strictly identical, or contained within one another, they share several data-flow and control-flow paths. For instance, multiplier mul3 is similarly fed with data from an input stream (coeff_in), and the result itself fed to an adder (add1); the common path is underlined in Figure 2(b). By adding 3 muxes and 3 wires as shown in the figure, it is possible to add a data path identical to the dot-product one (two input streams multiplied together, the result added with a register, and the new result fed back into the register) as emphasized in Figure 2(c). The control bit of the muxes acts as a toggle between a dot-product behavior and a lattice filter behavior, and the muxes are then *configuration multiplexers*; the resulting compound circuit contains both original circuits.

This compound circuit has a cost of 0.144 mm^2 using a 90nm TSMC library, while the lattice filter circuit has a cost of 0.14 mm^2, and the dot-product circuit a cost of 0.055 mm^2;



```
for (i = 0; i < NPOINTS; i++)
  top = data[i];
  for (j = 1; j < ORDER; j++)
    left = top;
    right = state[j];
    state[j] = botm;
    top = ( (coef[j-1] * left)
          - (coef[j] * right)
          ) >> prec;
    botm = ( (coef[j-1] * right)
             + (coef[j] * left)
             ) >> prec;
  state[ORDER] = botm;
  state[ORDER+1] = top;
  ...
```

(a) (b) (c)

Figure 2. (a) Lattice filter , (b) Circuit, (c) Compound circuit (MxM + lattice filter).

therefore the compound circuit is 27% cheaper than implementing both circuits. At the same time, the multiplexers add a delay of 0.89 ns to the lattice filter circuit, where the critical path of the lattice filter is 4.21 ns, the critical path of dot-product is 3.93 ns and the critical path of the compound circuit is 5.1 ns, i.e., a 29% increase w.r.t the dot-product and a 21% increase compared to the lattice filter.

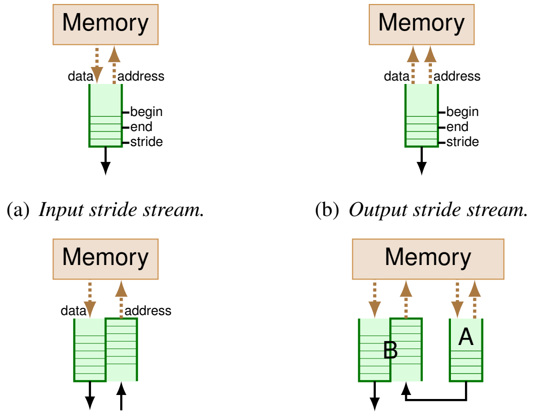
Moreover, by considering the two circuits as graphs, where operators and states are nodes, and wires are edges, it is possible to use systematic graph mapping exploration to map one circuit into another. At any stage during the exploration, if mapping is deemed impossible, it is possible to add new operators, configuration muxes and wires for achieving compatibility, with the goal of finding the cheapest possible solution. We later explain how to automate the process, and further optimize it using evolutionary heuristics.

3 Aggregating Individual Circuits Into Compound Circuits

3.1 Circuit representation and conversion

As mentioned before, circuits are described using an intermediate representation format akin to hardware description languages like Verilog (but simplified); in fact we later automatically translate this format into Verilog for synthesis evaluation (see Section 5). Circuits are composed of nodes (operators, state elements, streams) and edges (wires). Figure 1(c) provides an example of a circuit description; `node` and `edge` are self-explanatory, `setn` sets nodes characteristics.

ALU, wire and register conversion rules are rather straightforward. However, converting memory access and control parts of the circuit is significantly more complex. Budiu [8] proposed a set of systematic conversion rules which have the merit of proposing an intuitive and direct conversion from code to circuit. The benefit of an intuitive approach is facilitating the translation automation, but the downside is circuit performance, especially for load/store accesses.



(a) *Input stride stream.* (b) *Output stride stream.*

(c) *Input address stream.* (d) *B(A(i)) indirection.*

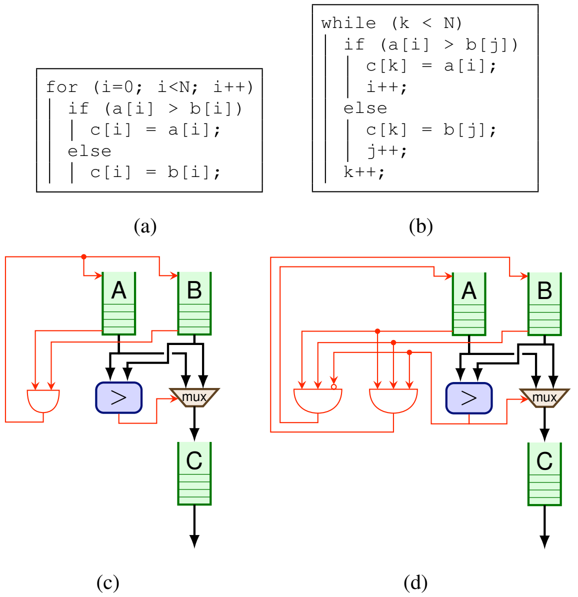
Figure 3. Stride and address streams.

Memory access. While program-to-circuit conversion yields significant power gains in [20], individual load accesses remain the main performance bottleneck. Like [13], our focus is on achieving high performance for small but key parts of a program, typically loop nests, rather than mapping the whole application onto a circuit. Therefore, we similarly replace references with FIFOs (exactly, stream buffers in our case) which can buffer several memory requests and keep the circuit busy. Unlike [13], we use stream buffers rather than a separate address generation unit; address generation is embedded within the stream. Address generation takes two possible forms: either a counter for *stride streams* where it is possible to specify the stride (for any type of loop), start and end addresses (for fixed-bounds loops), see Figure 3(a,b), or the address is provided as an input for *address streams* (for indirect addressing or pointer-based references to traverse lists), see Figure 3(c). The stride streams enable the efficient implementation of most array references within loops, and combine an address generation function with a loop control function. In order to preserve the latency hiding benefits of streams for small loops, we consider 1-loop, 2-loop and 3-loop streams, i.e., containing 1, 2 or 3 counters capable of characterizing up to 3-deep loops. However, for 2 and 3-loop streams, we only implemented counters and control for fixed-bound loops at the moment, i.e., inner loops with non-constant bounds cannot benefit from multi-loop streams at the moment, this is left for future work. The support for a 2-loop stream over a 1-loop stream corresponds to an additional cost of 0.00258 mm^2, or 16.7% more for a stream with two 32-byte entry (one entry is one memory block, see Section 4.2), and the same for a 3-loop stream over a 2-loop stream.

The coupling of address streams with standard ALU operators provides the same genericity as address generation units. All address streams contain a FIFO for buffering addresses, and a FIFO for buffering incoming or outgoing memory requests for respectively read and write streams. Indirect addressing is implemented by piping together a stride and an address stream, as shown in Figure 3(d); benchmark `histogram` (image histogram equalization) contains an example such indirection.

Control. In [8], the program is directly translated into a circuit, and so is the control: write signals of latches are propagated along data in a pipeline fashion, through a handshaking protocol. Since we resort to streams to achieve more efficient memory access, we break this direct translation, i.e., there is no longer a one-to-one mapping between program constructs and circuit elements (for instance loops and array references are combined into streams, several array references can use the same stream, etc). This makes the use of a similar mirroring and local control more difficult.

Consider the two example codes of Figure 4(a,b) which pick one element among two arrays, and either shift both arrays, or only the array where the element was taken. Using direct translation, these two examples result in fairly different circuits, especially for the address generation part. Expressed with streams, they both correspond to a circuit with the same data-flow part, see Figure 4(c,d), and the only distinction lays in the shift control signal of the streams: either `true` whenever both streams have data (Figure 4(a,c)) , or `true` when both streams have data and the stream was selected (Figure 4(b,d)). While it is not impossible to automatically generate the control signals for such cases, it is more difficult than the direct translation case, and we leave it for future work.



```
for (i=0; i<N; i++)
  if (a[i] > b[i])
    c[i] = a[i];
  else
    c[i] = b[i];
```

(a)

```
while (k < N)
  if (a[i] > b[j])
    c[k] = a[i];
    i++;
  else
    c[k] = b[j];
    j++;
  k++;
```

(b)

Figure 4. (a,b) Two example codes, (c,d) Circuits with identical data-flow parts.

At the same time, and unlike [13], we *explicit* the circuit control, so that data-flow and control-flow parts are indifferently combined with other circuits during the compounding process. Failing to do so, i.e., just compounding the data-flow parts and later manually recreating the control parts would not only become overly complex as the number and size of circuits increase, but it would also break a key feature of our approach: the ability to quickly and efficiently generate the compound circuits through an automatic process.

Therefore, we take the intermediate approach of manually expliciting the control of each *individual* circuit, and then *automatically compounding* these individual control circuit parts. Note that not separating control and dataflow parts in the compounding process has another advantage: some control sub-circuits can be mapped to data-flow sub-circuits. The reverse is usually not possible as control sub-circuits are often 1-bit wide, while data-flow circuits are 1-

word wide.

3.2 Creating compound circuits

Once data-flow and control-flow parts are expressed in the intermediate representation, they can also be considered as graphs, and the problem of mapping one circuit C_1 into another circuit C_2 is equivalent to deciding whether one graph is contained within another, and it is an NP problem. However, recall that we do not only want to check whether it is possible to *map* C_1 into C_2, but also to alter C_2 (by adding operators, state nodes, configuration muxes and wires) whenever mapping is not possible. Consider again the example of Figure 2 where we want to map the dot-product circuit into the lattice filter.

The process starts with a simple node types inspection. If C_2 does not contain a sufficient number of operators or state nodes of a given type, it is first complemented with the necessary and unconnected elements. Then, a randomly picked node N_1 from C_1 is compared to all nodes from C_2. The first comparison is based on nodes properties (e.g., type, width, number of input ports). For instance, a 2-input AND can be mapped to a 3-input AND (at the cost of an additional mux for the unused entry), or a 2-input 1-bit AND can be mapped to a 2-input 32-bit AND (same comment for unused bits). Note that this process allows to explore many mapping avenues in the future. For instance, we can slightly modify the output data path of multipliers so they can be used both as multipliers and adders, introduce more complex operators like multiply-add and map sub-circuits to a single operator, or even sub-circuits to sub-circuits, and so on.

Once a compatible node N_2 is found in C_2, N_1 and N_2 are said to match if their input nodes and output nodes are the same. This criterion, *recursively* applied to all nodes converges if C_1 is contained within C_2. Therefore, the input nodes of N_1 are tentatively matched to all input nodes of N_2; note that all possible input combinations must be tried, except for none-commutative operators (e.g., substraction) where the input order matters. For instance, if $N_1 = add$ from Figure 1(b) and $N_2 = add1$ from Figure 2(b), the matching will fail because the output of N_1 is a register `sum`, which cannot be mapped to the output of N_2 which is a shift operator `srl2`. That is also where our exploration differs from a pure graph comparison process. While the matching failed, the exploration then tries all other existing nodes of N_2 which are compatible with the output register `sum` of N_1. When it finds that `bottom` in C_2 is compatible with `sum` in C_1, it adds a configuration mux connected to the input of `bottom`, connects back the output of `srl2` to one of the inputs of the mux feeding `bottom`, and connects the output of `add1` to the other input of the configuration mux, see Figure 2(c). Then, this sub-circuit of N_1 and N_2 match at the cost of one 32-bit wire and one 2x1 32-bit configuration mux, and the exploration can resume. Therefore, all possible merging possibilities are explored. The algorithm is shown in Figure 5.

```
label_N1:
randomly pick unmatched node N1 of C1
label_N2:
randomly pick unmatched node N2 of C2

attempt to match N1 to N2:
if properties of N1 and N2 match
| if each source of N1 matched or matchable
|    to a source of N2
|  | match sucessful
|  else
|  | if a source S1 of N1 is matchable to a
|  |    node S2 of C2 not a source of N2
|  |  | connect S2 to N2,
|  |  | add wires and configuration muxes as needed

if match successful
| if was last unmatched N1, a solution is found
| else go to label_N1
else
| unmatch all sources
| remove added wires and muxes
| go to label_N2
```

Figure 5. Exploration algorithm.

The huge number of possibilities makes an exhaustive exploration prohibitive beyond a few tens of nodes in either circuit, which calls for the exploration heuristics described in Section 6.1. At the same time, a first solution can be quickly found since the principle is to complement the circuit whenever mapping fails. This property is later exploited for fast exploration heuristics.

3.3 Exploration cost function

As explained before, the first step of the exploration is to complement C_2 with whatever data or control nodes (operators, registers, streams) are missing in order to include C_1. Therefore, the exploration itself will not change the data or control nodes cost, it will only have an impact on the number of configuration muxes and wires which are added. When few and small circuits are combined, this compound overhead is small; however, it will exponentially grow with the size and number of circuits, towards the extreme case of FPGAs. Therefore, it is important to minimize this compound overhead; it is the role of the exploration cost function to characterize the overhead.

The best way to evaluate the overhead is to use a synthesis tool for each solution found in order to compute its area cost. But that option would be prohibitively time-consuming, and ultimately lead to poorer solutions because only a much smaller fraction of the design space could be explored. Therefore, we resort to a more simple cost function, which can capture the cost of the configuration muxes and the wires: *the total number of input ports of the configu-*

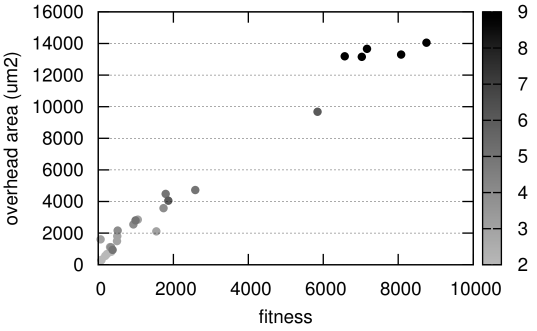


Figure 6. Correlation of fitness function and compound circuit overhead.

ration muxes weighted by their bit width. This metric simultaneously accounts for the number of configuration muxes, their size and the wires which had to be added. Figure 6 shows the nearly linear correlelation of the proposed metric (`fitness`) with the actual overhead cost (`mux area`) for multiple randomly chosen compound circuits, which aggregate from 2 individual circuits (light gray) to 9 individual circuits (dark gray).

Note that normally, the exploration should simultaneously attempt to minimize the circuit critical path (performance) and the area (cost) in a pareto-optimal way. But, not surprisingly, we found that both criteria are tightly related, leading to similar best solutions, as reducing the number and size of muxes usually yields the smallest critical paths.

4 System Integration of Compound Circuit

4.1 Processor Interface

The general architecture of the system is akin to previously proposed loop accelerators [13, 23] as shown in Figure 8(a): the compound circuit directly interfaces with the processor for initialization, configuration and value exchange operations, and with the system bus to access upper levels of the memory hierarchy. To start the accelerator, the processor needs to communicate the configuration to the accelerator (which circuit to execute) and the initialization values of the state nodes (typically initial values for the registers, stride values, base addresses and streams depths). Note that configuring the accelerator is straightforward: each circuit is given an ID, this ID is stored in a configuration register CR which is directly used to set configuration muxes.

In order to support the compound circuits, we added four instructions to the baseline instruction set: CCINIT (Initialize), CCMTA (Move To Accelerator), CCSTART and CCMFA (Move From Accelerator). The processor first initializes the accelerator with the circuit ID (CCINIT), the circuit ID is stored in the configuration register CR of the accelerator. Values to initialize state nodes (e.g., register init values, stride streams counter values, register output values) are then sent to the accelerator using the instruction CCMTA *Rsrc*, where *Rsrc* contains the value to send to the accelerator. Then a start signal is sent to the accelerator with the CCSTART instruction. An accelerator circuit comes with a set of latches directly wired, as needed, to state nodes in the circuit, see Figure 7. During execution, the pipeline waits until it receives a STOP signal from the accelerator. The pipeline can receive values directly from the accelerator using the CCMFA *Rdst* instruction, the results are written in the processor regsiter *Rdst*.

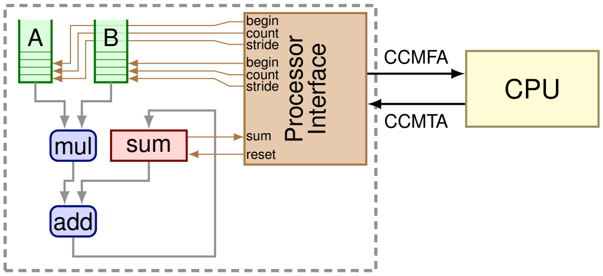


Figure 7. Processor interface.

4.2 Memory Interface

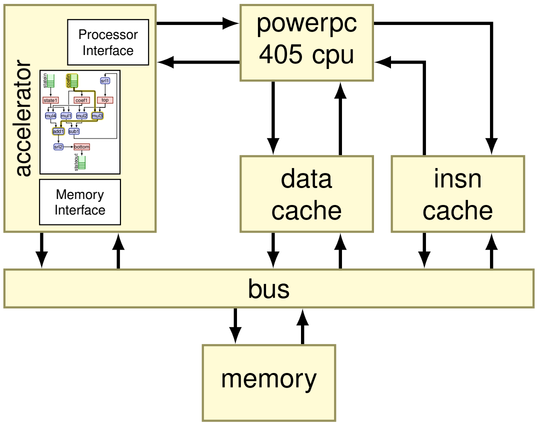
The role of streams is to minimize circuit stall cycles; failing to achieve high circuit throughput would largely void the performance advantage of using customized circuits instead of a processor. In fact, circuit performance should only be limited by the memory sub-system bandwidth; in other words, the ultimate circuit performance goal is to *saturate* the memory bandwidth.

A circuit consumes one data (word) at a time from each stream, but in most cases, the processor memory controler is designed to issue block fetch requests (e.g., cache lines) to the interconnect (bus or NoC). The original Stream Buffers [16] prefetch one cache line, and each buffer entry is a single cache line. Our streams similarly have 1-memory block (e.g., cache line) entries but they issue 1-word requests to the memory interface; however, these requests are filtered in such a way that only block fetch requests are sent to the memory controler in the end

In order to cope with multiple simultaneous streams requests, the memory interface keeps track of outstanding request and returns them to the propers streams. Because incoming requests may not come in order, requests are *preallocated* in the streams and are stored in the proper order when they return.

5 Methodology

Simulated Architecture. Our architecture is shown in Figure 8(a) and consists of an IBM PowerPC405 [15] core, a simple 32-bit embedded RISC-processor core including a 5-stage pipeline and 32 registers, but no floating-point units. We consider a regular 90nm version running at a frequency of 400MHz. The memory is an SDRAM with an observed average latency of 20 cycles over our set of benchmarks. To simulate this architecture, we used the UNISIM [6] infrastructure environment.



(a) Architecture

icache	cache lines	128
	line size	32
	associativity	2
dcache	cache lines	128
	line size	32
	associativity	2
bus	datapath width	32
memory	banks	4
	rows	8192
	columns	1024
	control queue size	16

(b) *Memory hierarchy parameters*

Figure 8. Simulated architecture.

The memory sub-system is composed of two write-back L1 data and instruction caches and a main memory. Their parameters are described in Figure 8(b). The compound circuit, the processor interface and the memory interface composing the accelerator are described in Sections 3, 4.1 and 4.2.

Synthesis infrastructure. As mentioned before, automatically generating hardware representation from a source code has been previously addressed in research and existing industrial tools [4]. We developed a tool chain which automatically creates compound circuits and generates Verilog HDL based on our intermediate circuit representation. We then synthesized all circuits using Synopsys Design Compiler [1] and TSMC 90nm standard library, with the highest mapping effort of the design compiler (`-map_effort high -area_effort high` options).

Benchmarks. To evaluate the potential cost gains as well as the speedups obtained using compound circuits, we used 9 of the UTDSP benchmarks [17], which are small signal-processing kernels, described in Table 1. Since we do not implement floating point operators for now, all of the target benchmarks where modified to support 12-bit fixed point precision arithmetic; fixed-point arithmetic is frequently used in embedded systems.

Benchmark	#Calls to accelerator	Description	Area (μm^2)
compress	32768	Discrete Cosine Transform	54536.69
edgedetect	9	Convolution loop	17883.36
fft	1023	1024-point Complex FFT	219873.47
fir	1	256-tap FIR filter	54800.69
histogram	1	Image enhancement using histogram equalization (gray level mapping loop)	62132.84
iir	64	4-cascaded IIR biquad filter processing	218793.06
latnrm	64	32nd-order Normalized Lattice filter	143522.36
lmsfir	64	32-tap LMS adaptive FIR filter	74849.71
mult	100	Matrix Multiplication	54557.17

Table 1. Benchmark description.

For each benchmark shown in Table 1, we derived the intermediate circuit representation, as described in Section 2. Table 1 also indicates the area size of each accelerator, and the number of times it is called.

6 Exploration and evaluation of compound circuits

6.1 Evolutionary computation for compound circuit aggregation

Using exhaustive exploration to aggregate many large circuits into a single compound one is prohibitive in terms of computation time, because exhaustive exploration implies to find *all possible ways* of aggregating one circuit into any other subset of circuits. In this section, we present a strategy based on evolutionary computation principles [14] that is more scalable w.r.t the number of circuits and is based on the fact that finding *a single solution* for aggregating two circuits is fast, even for large circuits.

Our evolutionary algorithm proceeds though successive generations of a population of intermediate compound circuits and selection of population members based on their cost function (see 3.3). Let us assume we want to aggregate N circuits (referred below as the ancestor circuits) into a single compound one. At generation 1, we build a population composed of P circuits (P is a user-defined parameter) which is a subset (chosen at random) of the circuits we want to aggregate (the ancestors). One circuit is then chosen at random among this population and aggregated to a randomly-chosen ancestor via a single randomly-chosen exploration solution. This process is iterated E times (E is a user-defined parameter), yielding a population of E non-identical intermediate circuits, each of them aggregating 2 ancestors. Generation 2 is then obtained by picking up P circuits among these E intermediate circuits. This selection stage is carried out so that the probability that a given circuit is selected decays with increasing cost functions. The steps above (creation of E circuits from the previous generation and cost-based selection of P of them) are iterated to obtain successive generations until generation N, in which each circuit has aggregated all the ancestor circuits. The final compound circuit is the circuit of generation N with smallest cost function.

With $P = E = 1$, the above algorithm corresponds to a purely random search without optimization. Figure 9(a) shows how the typical computation time scales with the number of ancestor circuits in this case. Because the algorithm depends on random realizations, each run of the algorithm will need different computation times, but the necessary computational effort remains very low. Even with $N = 9$ ancestors, the typical computation time (on a 2.4 GHz AMD 64 Athlon processor) remains less than a second. On the other hand, the cost of the final compound circuit obtained by this non-optimized method grows rapidly (and almost linearly) with the number of circuits (Figure 9(b)).

In order to optimize the cost of the compound circuit, we need to increase P and E. For instance using $P = 3$ and $E = 6$ (see Figure 9(b)) significantly improves the cost/quality of the final compound. When up to 7 circuits are merged and the computational time is less than 80 seconds, the typical improvement brought by the optimized algorithm ranges between 2.0 and 2.5, i.e., the cost of the final compound circuit is typically less than half of that obtained without the algorithm, within the same period of time. With a larger number of circuits (8 or 9) and the same computational time, the improvement is lower but still significant (1.2 to 1.7). The user is naturally free to set different computational time/cost benefit tradeoffs at design time.

6.2 Compound circuits characteristics

As previously discussed, the major advantage of compound circuits is their ability to reuse common data paths hence avoiding paying the cost of accelerating each benchmark separately. Figure 10 shows the area (in $\mu\mathrm{m}^2$) of the 9 synthesized circuits (corresponding to the 9 target benchmarks) as well as 12 compound circuits (where the number of merged circuits are varied from 2 to 9 circuits), the gray bars show the original area for single circuits and the sum of the areas of the individual circuits for compound circuits, and the black bars show the area of the compound circuits. The percentages correspond to the area reduction. Figure 10 shows an area saving of 66% when compounding all the circuits, and an average saving of 40.9% across 2-circuit to 9-circuit compounds.

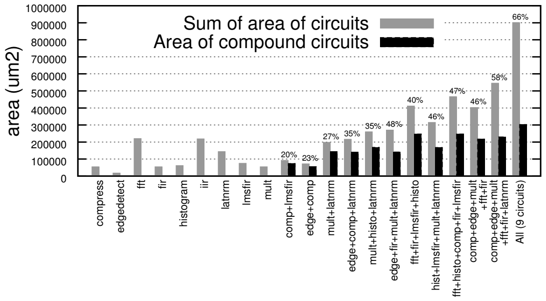


Figure 10. Area cost of individual and resulting compound circuits.

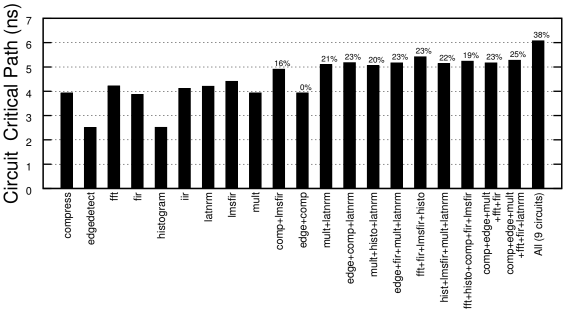
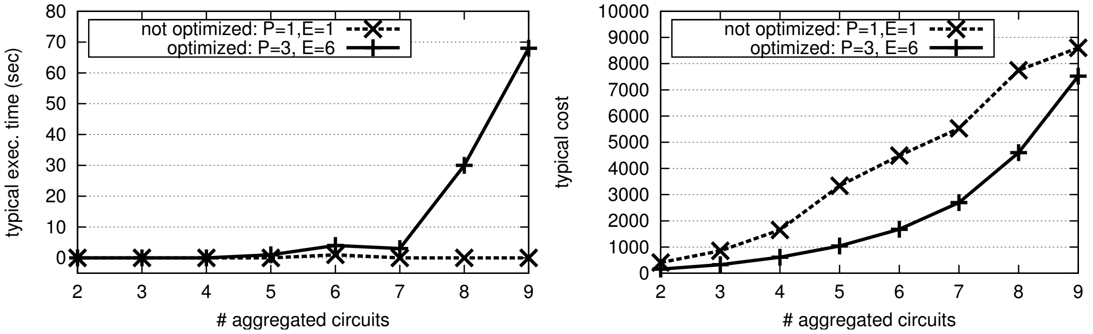


Figure 11. Critical path of individual and resulting compound circuits.

We also evaluate the overhead cost, and compare the compound circuit cost to individual circuits. Synthesis results show that a 9-circuit compound is only 38% bigger than the biggest individual circuit (`fft`). The 12 generated compounds were, on average, 8% larger than the largest individual circuit included in the compound.

Impact on critical path. Figure 11 shows the impact of compounding the circuits on the critical path. The critical path is the minimum clock cycle time with which the circuit could be synthesized without constraints violations. Naturally, the critical path of the compound circuit is higher than the critical path of each included individual circuit, due to the configuration muxes overhead. Figure 11 shows an average increase of 21% of the critical path over the circuit with the largest critical path, and a maximum increase of 37%.

In spite of the increase of the compound circuit critical path over the individual circuit critical path, the performance of the compound circuit is still significantly higher than the software version, as will be demonstrated in Section 7. Moreover, there are several venues for further compensating for the compound higher critical path la-



(a) *Evolution of the typical execution time.*

(b) *Evolution of the cost of the compound circuits.*

Figure 9. Aggregation of increasing numbers of circuits. Each value is the median on 50 realizations of the algorithm and the individual circuits are chosen at random among the benchmarks listed in Table 1.

tency. First and foremost, these results correspond to non-pipelined circuits, and the highly streamed nature of the circuits would largely benefit from pipelined versions. Second, the critical path of most individual circuits contained *within* the compound circuit is likely to be smaller than the compound critical path (except for the individual circuits using the compound critical path circuit), opening up the possibility of clocking the compound circuit in a variable way depending on which target circuit is being executed. Both venues are left for future work.

#iterations inner loop	#calls (iterations outer loop)	speedup
512	1	7.92
256	2	6.96
128	4	5.60
64	8	4.05
32	16	2.64
16	32	1.61
8	64	0.96
4	128	0.60
2	256	0.41
1	512	0.30

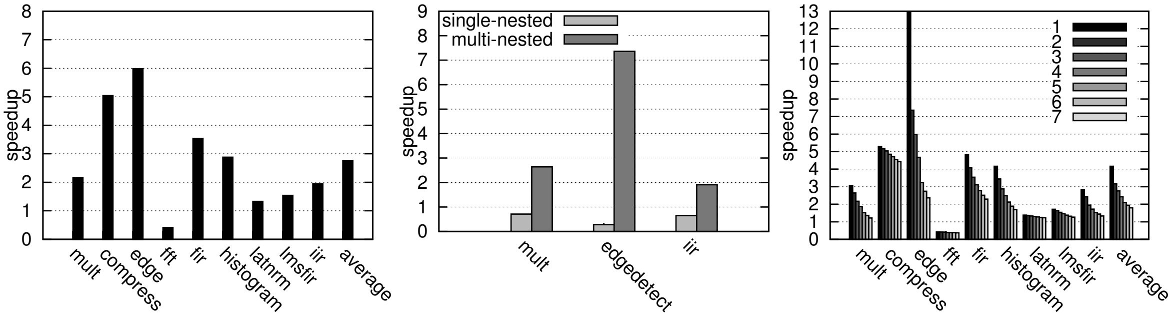
Table 2. FFT case study

7 Performance Evaluation

On average, the speedup of our circuit-based acceleration approach, using the full 9-circuit compound, is 2.8, as shown in Figure 12(a). Since the host processor runs at 400MHz (2.5ns), and the compound circuit critical path is 6.1ns, we report these speedups for a compound circuit latency of 3 cycles. The better performance comes from a combination of faster execution of sequences of ALU operations, and the latency hiding properties of streams.

Note that, depending on the target markets, fewer circuits may be bundled together, so that smaller compound circuits could be used, with a correspondingly smaller critical path. For instance, if a `comp+lmsfir` compound is used for these two circuits, the critical path at 4.9ns, see Figure 11, would enable a 2-cycle latency, breeding higher speedups. Figure 12(c) shows the speedup as a function of the circuit latency for each benchmark. This latter figure also shows that, as the processor clock frequency increases, and thus the compound circuit cycle latency increases, the speedup degrades logarithmically on average. With a 4-cycle latency (corresponding to a 1.525ns clock cycle, i.e., a 655MHz processor), the average speedup is still 2.4, and 1.8 for a 7-cycle circuit latency (potentially corresponding to a 1.14Ghz processor). Moreover, this is a pessimistic evaluation of the speedup when the clock frequency increases, because the stream buffers are more capable to tolerate longer memory latencies (due to higher processor clock frequency) than a traditional cache hierarchy. Finally, Figure 12(c), also hints at the potential performance benefit of pipelining the compound circuit. Even though the 1-cycle latency is a performance upper-bound of the pipelined version, the higher control circuit overhead should only moderately increase the overall critical path latency. Combined with the absence of pipeline hazards (other than memory stalls, already accounted for in the 1-cycle experiments), it suggests that the pipelined version performance should be in-between the present 1-cycle and 2-cycle latency speedups.

For benchmarks `mult`, `edgedetect`, `iir`, potential performance benefits are impaired by small innermost loops (10, 3 and 4 iterations respectively) when using only 1-loop stride streams (see Section 3.1); for instance, one of `iir` innermost loops has 4 iterations, while the outer loop has 64 iterations for a total of $4 \times 64 = 256$ innermost iterations instances. Short inner loops require to pay the overhead of calling and starting the circuit more frequently, and hamper latency hiding capabilities of streams. In such cases, using the slightly more costly multi-loop streams provide significant performance benefits, as shown in Figure 12(b).



(a) Speedup using the 9-circuit compound (3-cycle latency accelerators).

(b) Multi-Nested loop circuits.

(c) Impact of circuit cycle latency.

Figure 12. Speedups.

However, for one program, `fft`, the performance improvement remains small. This program similarly suffers from many instances of small innermost loops, with an average of 5.12 iterations for the inner loop. Table 2 decomposes one run of this benchmark and shows the speedup obtained depending on the number of iterations of the inner loop, reported in the first column. The speedup is high for a large number of inner loop iterations and decreases quickly with the inner loop size. Since this algorithm was implemented with 1-loop streams, each instance of the inner loop is a call to the accelerator, and, on average, the number of iterations of the inner loop per call is quite small (5.12). At the moment, our multi-loop streams only accomodate fixed-bound loops, and it was not possible to implement the triangular FFT loop using these streams, though there is no fundamental implementation or cost issue, and we plan to extend our streams to cope with such loops in the future.

8 Conclusion

In this article, we investigate how to create accelerators which can benefit a variable number of target applications, thereby reaching a given point between flexibility and efficiency. The potential markets for such approaches range from the increasingly multi-purpose consumer electronics devices to many low-volume high-margin embedded systems markets (defense, medical applications,...) as well as general-purpose processors seeking market differentiation.

For that purpose, we propose a systematic and low-overhead method to combine several loop-accelerated circuits within the same compound accelerator. The compounding processing is based on automatic graph exploration and complementation, simultaneously considering the data and control flow part of circuits. We show how to make the process scale with the number of circuits by replacing exhaustive exploration with a statistical, evolutionary, process which makes it possible to find low-area compound circuits in a small design time. We show that compounding 9 circuits can reduce circuit area by a factor of 3 compared to the traditional approach of implementing one accelerator per target application, while increasing the critical path by 37% only compared to individual circuits. Without resorting to pipelined operators nor fine-tuning the clocking of the compound circuit for each target circuit, we already demonstrate an average speedup of 2.8 on a PPC405 host processor.

Compound circuits open up many challenges as well as perspectives. Combining acceleration with parallelism will enable the generation of more efficient compound circuits. Furthermore, since the circuit mapping process works by matching increasingly large sub-circuits, it can be used for identifying frequently occurring sub-circuits, i.e., recurring similar program patterns. Over time, these patterns can be further manually optimized, yielding a library of fine-tuned operators compatible/useful for a large range of programs. Also more aggressive circuit exploration and matching can be investigated to achieve more space/time tradeoffs, for instance, matching an existing barrel shifter or a simple adder to a multiply-by-two operator. Our approach can also greatly benefit from bit-width analysis [25] and progressive decomposition techniques [26] to optimize the arithmetic operators of compound circuits.

9 Acknowledgements

We would like to thank the anonymous referees for their excellent feedback on this work. We would also like to thank the Embedded Systems Lab colleagues at Thales Research and Technology for their insightful feedback and their support. This work is supported by the HiPEAC Network of Excellence, contract No. IST-004408.

References

[1] Synopsys design compiler. http://www.synopsys.com.

[2] Tensilica. http://www.tensilica.com/.

[3] Virtex-5 multi-platform fpga. http:// www.xilinx.com / products / silicon_solutions / fpgas / virtex /.

[4] Designing high-performance dsp hardware using Catapult C synthesis and the altera accelerated libraries. Mentor Graphics Technical Library, October 2007.

[5] G. Ansaloni, P. Bonzini, and L. Pozzi. Design and architectural exploration of expression-grained reconfigurable arrays. In *Proc. of the 6th IEEE Symposium on Application Specific Processors*, 2008.

[6] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. A. Penry, O. Temam, and N. Vachharajani. Unisim: An open simulation environment and library for complex architecture design and collaborative development. *IEEE Comput. Archit. Lett.*, 6(2):45–48, 2007.

[7] P. Brisk, A. Kaplan, and M. Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 395–400, New York, NY, USA, 2004. ACM.

[8] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. Spatial computation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 14–26, Boston, MA, October 2004.

[9] A. C.-H. Cheng. *Application-specific architecture framework for high-performance low-power embedded computing*. PhD thesis, Ann Arbor, MI, USA, 2006.

[10] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi. The reconfigurable streaming vector processor (rsvptm). In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 141, Washington, DC, USA, 2003. IEEE Computer Society.

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

[12] N. Clark et al. OptimoDE: Programmable accelerator engines through retargetable customization, 2004. In *Proc. of Hot Chips 16*.

[13] N. Clark, A. Hormati, and S. Mahlke. Veal: Virtualized execution accelerator for loops. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 389–400, Washington, DC, USA, 2008. IEEE Computer Society.

[14] K. De Jong. *Evolutionary Computation: A Unified Approach*. MIT Press, 2006.

[15] IBM. PowerPC 405 CPU Core. Sept. 2006.

[16] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *SIGARCH Comput. Archit. News*, 18(3a):364–373, 1990.

[17] C. G. Lee. UTDSP benchmark suite. http:// www.eecg.toronto.edu / ˜corinna / DSP / infrastructure / UTDSP.html, 1998.

[18] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. Soda: A low-power architecture for software radio. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 89–101, Washington, DC, USA, 2006. IEEE Computer Society.

[19] B. Mathew and A. Davis. A loop accelerator for low power embedded vliw processors. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 6–11, New York, NY, USA, 2004. ACM.

[20] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, M. Budiu, and S. C. Goldstein. Tartan: Evaluating spatial computation for whole program execution. In *12th ACM International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, pages 163–174, San Jose, CA, 2006.

[21] N. Moreano, E. Borin, C. de Souza, and G. Araujo. Efficient datapath merging for partially reconfigurable architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(7):969–980, July 2005.

[22] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. *SIGARCH Comput. Archit. News*, 31(2):422–433, 2003.

[23] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman. Pico-npa: High-level synthesis of nonprogrammable hardware accelerators. *J. VLSI Signal Process. Syst.*, 31(2):127–142, 2002.

[24] L. Shang and N. Jha. High-level power modeling of cplds and fpgas. *Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on*, pages 46–51, 2001.

[25] M. Stephenson, J. Babb, and S. Amarasinghe. Bidwidth analysis with application to silicon compilation. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 108–120, New York, NY, USA, 2000. ACM.

[26] A. K. Verma, P. Brisk, and P. Ienne. Progressive decomposition: a heuristic to structure arithmetic circuits. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 404–409, New York, NY, USA, 2007. ACM.

[27] S. Yehia and O. Temam. From sequences of dependent instructions to functions: An approach for improving performance without ILP or speculation. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 238, Washington, DC, USA, 2004. IEEE Computer Society.

[28] M. Zuluaga and N. Topham. Resource sharing in custom instruction set extensions. In *Proc. of the 6th IEEE Symposium on Application Specific Processors*, 2008.