Increasing Hardware Efficiency with Multifunction Loop Accelerators

Kevin Fan Manjunath Kudlur Hyunchul Park Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI 48109
{fank, kvman, parkhc, mahlke}@umich.edu

ABSTRACT

To meet the conflicting goals of high-performance low-cost embedded systems, critical application loop nests are commonly executed on specialized hardware accelerators. These loop accelerators are traditionally designed in a single-function manner, wherein each loop nest is implemented as a dedicated hardware block. This paper focuses on hardware sharing across loop nests by creating multifunction loop accelerators, or accelerators capable of executing multiple algorithms. A compiler-based system for automatically synthesizing multifunction loop accelerator architectures from C code is presented. We compare the effectiveness of three architecture synthesis approaches with varying levels of complexity: sum of individual accelerators, union of individual accelerators, and joint accelerator synthesis. Experiments show that multifunction accelerators achieve substantial hardware savings over combinations of single-function designs. In addition, the union approach to multifunction synthesis is shown to be effective at creating low-cost hardware by exploiting hardware sharing, while remaining computationally tractable.

Categories and Subject Descriptors

B.5.2 [**Register-transfer-level Implementation**]: Design Aids—*Automatic synthesis*

General Terms

Algorithms, Design, Experimentation

Keywords

high-level synthesis, application-specific hardware, loop accelerator, modulo scheduling, multifunction design

1. INTRODUCTION

The markets for wireless handsets, PDAs, and other portable devices continue to grow explosively, fueled by demand for new functionality, added capabilities, and higher bandwidth. These devices require higher performance, lower cost, and more energy-efficient computer systems to meet user requirements. To achieve these challenging goals, specialized hardware in the form of loop accelerators are commonly used for the compute-intensive portions of applications that would run too slowly if implemented in software on a programmable processor. Low-cost design, systematic verification, and short time-to-market are critical objectives for designing these accelerators. Automatic synthesis of accelerators from high-level specifications has the potential to meet these objectives.

There is also a growing push to increase the functionality of special-purpose hardware. Many applications that run on portable devices, such as wireless networking, do not have one dominant loop nest that requires acceleration. Rather, these applications are composed of a number of compute-intensive algorithms, including filters, transforms, encoders, and decoders. Further, increasing capabilities, such as supporting streaming video or multiple wireless protocols, places a larger burden on the hardware designer to support more functionality. Dedicated accelerators for each critical algorithm could be created and included in a system-on-chip. However, the inability to share hardware between individual accelerators creates an inefficient design. Processor-based solutions are the obvious approach to creating multi-purpose designs due to their inherent programmability. However, such solutions do not offer the performance, cost, and energy efficiency of accelerators as there is an inherent overhead to instruction-based execution.

To understand the performance and cost efficiency of automatically synthesized loop accelerators, an accelerator was generated to implement a 256-state, K=9 Viterbi decoder and compared to an ARM926EJ-S running the algorithm at 250 MHz. The accelerator was synthesized in 0.18μ technology at 200 MHz and was 47x faster than the ARM while being 90% smaller. Thus, loop accelerators offer the potential for substantial efficiency gains over programmable processors by removing the control overhead of instruction-based execution and specializing the datapath to a particular application.

The focus of this paper is on automatic design of multifunction loop accelerators from high-level specifications. Our goal is to maintain the efficiency of single-function accelerators (ASICs) while exposing opportunities for hardware sharing across multiple algorithms. The inputs to the system are the target applications expressed in C and the desired throughput. The proposed system is built upon a single-function loop accelerator design system that employs a compiler-directed approach, similar to the PICO-NPA (Program In Chip Out) system [18]. Accelerators are synthesized by mapping the algorithm to a simple VLIW processor and then extracting a stylized accelerator architecture from the compiler mapping.

To create multifunction designs, the single-function system is extended using three alternate strategies. The simplest strategy is to create individual accelerators for each algorithm and place them next to each other. This method is referred to as a summed design, and is the baseline for comparison. The second strategy is to again create individual accelerators for each algorithm. The data and control paths for each accelerator are then intelligently unioned

CODES+ISSS'06, October 22–25, 2006, Seoul, Korea.

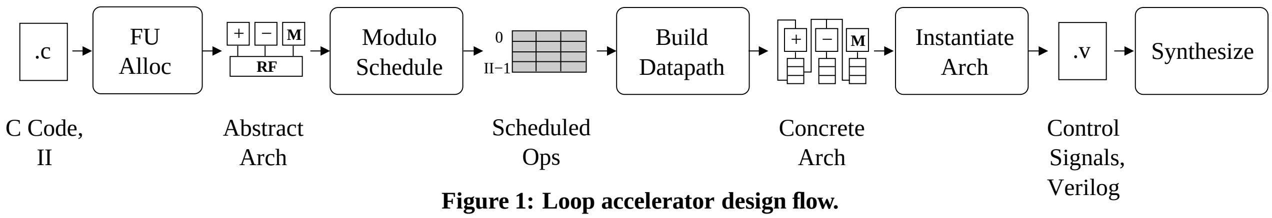

Figure 1: Loop accelerator design flow.

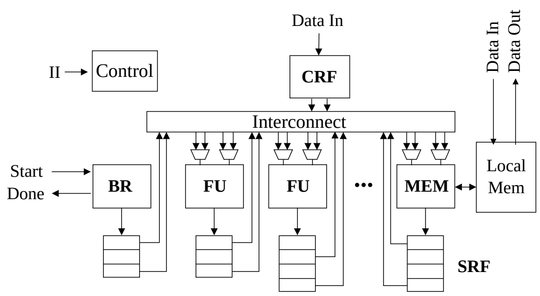

Figure 2: Hardware schema for loop accelerator.

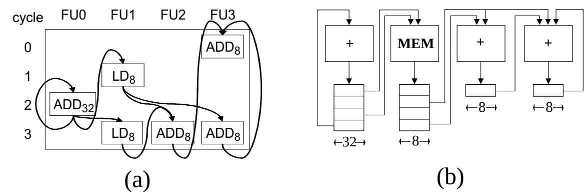

Figure 3: (a) A portion of the modulo schedule for `sobel`, and (b) the corresponding datapath.

together to create a single design capable of all algorithms. Finally, the third strategy is to perform joint cost-aware synthesis of all algorithms. We employ an integer linear programming (ILP) formulation to find a joint solution with optimal estimated cost. A consequence of the joint scheduling strategy is that synthesis time and memory usage may become prohibitive for large loop bodies or large numbers of loops. Each successive strategy represents a more complex approach and hence has more potential to exploit sharing opportunities.

2. ACCELERATOR SYNTHESIS SYSTEM

The synthesis system takes an application loop in C along with a performance requirement, and generates RTL for a hardware accelerator which implements the loop. The performance requirement is specified as an initiation interval (II), or the number of cycles between the initiation of successive loop iterations. The overall flow of the system that creates a stylized hardware implementation for a modulo scheduled loop [17] is presented in Figure 1, and each step is discussed in Section 2.1.

The hardware schema used in this paper is shown in Figure 2. The accelerator is designed to exploit the high degree of parallelism available in modulo scheduled loops with a large number of FUs. Each FU writes to a dedicated shift register file (SRF); in each cycle, the contents of the registers shift downwards to the next register. The entries in a SRF therefore contain the values produced by the corresponding FU in the order they were computed. Wires from the registers back to the FU inputs allow data transfer from producers to consumers. Multiple registers may be connected to each FU input; a multiplexer (MUX) is used to select the appropriate one. Since the operations executing in a modulo scheduled loop are periodic, the selector for this MUX is simply a modulo counter. In addition, a central register file (CRF) holds static live-in register values which cannot be stored in the SRFs.

2.1 Loop Accelerator Synthesis

The first step in the loop accelerator synthesis process is the creation of an abstract VLIW architecture to which the application is mapped. The abstract architecture is parameterized only by the number of FUs and their capabilities. A single unified register file with infinite ports/elements that is connected to all FUs is assumed. Given the operations in the loop, the desired throughput, and a library of hardware cell capabilities and costs, the *FU allocation* stage generates a mix of FUs that minimizes cost while providing enough resources to meet the performance constraint.

Next, the loop is modulo scheduled to the abstract architecture. Modulo scheduling is a method of overlapping iterations of a loop to achieve high throughput. The modulo schedule contains a *kernel* which repeats every II cycles and may include operations from multiple loop iterations. The scheduler assigns the operations in the loop to FUs and time slots, satisfying all inter-operation dependencies and meeting the II requirement. After scheduling, the accelerator datapath is derived from the producer-consumer relationships in the schedule. This includes setting the widths of the FUs and the widths and depths of the SRFs, and connecting specific SRF entries with the appropriate FU inputs. Since the datapath is derived from the schedule, the choice of scheduling alternative for each operation has a significant effect on the cost of the resulting hardware [5]. It is important that the scheduler be aware of these effects.

Figure 3(a) shows a few operations from the modulo schedule for `sobel`, an edge-detection algorithm. The II in this example is 4, thus each FU has 4 scheduling slots. The number associated with each operation indicates its width, and edges represent dataflow between operations. Figure 3(b) shows the SRFs and connections required to execute these scheduled operations. The widths of the FUs and SRFs are set to the width of the largest operation assigned to them, and the depths of the SRFs are determined by the maximum lifetime of any variable assigned to them.

Based on the datapath that is instantiated, the control path is generated for the accelerator. This consists of a modulo-II counter which directs FU execution (for FUs capable of multiple operations) and controls the MUXes at the FU inputs.

Finally, a Verilog realization of the accelerator is generated by emitting modules with pre-defined behavioral Verilog descriptions that correspond to the datapath elements. Gate-level synthesis and placement/routing are then performed on the Verilog output.

2.2 Related Work

Datapath synthesis from behavioral specifications is a field that has been studied for many years. The basic techniques, including resource allocation and scheduling, have been well established [6]. Cathedral III represents a complete synthesis system developed at IMEC and illustrates one comprehensive approach to high-level synthesis [14]. Force-directed scheduling is used to synthesize datapaths for ASIC design [16]. The Sehwa system automatically designs processing pipelines from behavioral specifications [15]. Clique based partitioning algorithms were developed in the FACET project to jointly minimize FU and communication costs [22].

Automatic mapping of applications to FPGA-based and other reconfigurable systems has also been investigated. One of the first efforts to automatically map applications onto an FPGA was Splash [7], subsequently productized as the NAPA system [8]. Other automatic compiler systems for FPGA-based platforms include GARP [2], PRISM [24], and DEFACTO [1]. Modulo scheduling has been used [19, 12] to map critical loops onto reconfigurable coprocessors. Compilation for architectures consisting of predefined FUs and storage with reconfigurable interconnect have been investigated, including RaPiD [3] and PipeRench [9]. Generation of more efficient designs by sharing hardware across basic blocks was recently proposed [13]. Cost sensitive scheduling, used within the synthesis system to reduce hardware cost, has been studied in the context of storage and interconnect minimization in [20, 11, 5] and to improve resource sharing [23].

This paper extends prior work in an orthogonal direction by investigating multifunction accelerators. A single accelerator is designed that is capable of executing multiple algorithms. While the resulting designs could be implemented on an FPGA, our intent is to design standard cell implementations.

3. MULTIFUNCTION ACCELERATORS

Multifunction design refers to generalizing a loop accelerator to support two or more loop nests. One obvious approach to creating a multifunction accelerator is to separately design accelerators for the individual loops, and then place these loop accelerators side by side in silicon. The area of the final accelerator would be the sum of the areas of the individual accelerators. However, by creating an accelerator with a single datapath that can support multiple loops, more hardware sharing can be achieved while continuing to meet the throughput constraints of both loops.

The cost of a multifunction accelerator is affected by the individual functions in several ways. First, the execution resources required by the multifunction accelerator must be a superset of the resources required for each individual accelerator. Since the multiple functions will not be executing simultaneously, any resources common to the individual accelerators need only be instantiated once in the combined accelerator. Effectively, the multifunction accelerator should have the union of the FUs required by the individual accelerators. Second, the cost of the SRFs is sensitive to how the hardware is shared across functions. Since every FU has an SRF at its output, and the SRF has the bitwidth of its widest member and the depth of its value with the longest lifetime, there is a potential for careless sharing to result in large, underutilized SRFs. Third, one advantage of a customized ASIC is that there are few control signals that need to be distributed across the chip, since the datapath is hard-wired for a specific loop. When multiple loops come into play, not only must the datapath be able to support the computation and communication requirements of each loop, but the control path must be capable of directing the datapath according to which loop is being executed.

Two techniques are presented to increase hardware sharing: joint scheduling and the union of individually designed accelerators.

3.1 Joint Scheduling

Since the cost of the multifunction datapath depends on the combined schedules of all loops, an ideal scheduler should look at all loops simultaneously and schedule them to minimize the total hardware cost (while meeting their individual II constraints). This is referred to as *joint scheduling*; the scheduler is aware that all loops will execute on the same hardware, and is therefore able to make scheduling decisions that maximize hardware sharing across loops.

An ILP formulation for joint scheduling is used. This formulation is similar to the modulo scheduling formulation proposed in [4, 10], along with extensions to minimize accelerator cost as proposed in [5]. These formulations are extended to consider multiple loops simultaneously. For each loop a under consideration, integer variables to represent time and FU assignment are introduced. For every operation i in loop a, II_a mutually exclusive binary variables $X_{i,s,a}$ represent the time slot s in the modulo reservation table (MRT) that the operation is scheduled. The integer variables $k_{i,a}$ represent the stage in which operation i is scheduled. Binary variables $R_{i,f,a}$ represent the assignment of operation i in loop a to the FU f. The set of variables $X_{i,s,a}$, $k_{i,a}$, and $R_{i,f,a}$ represent complete modulo schedules for the loops. Other auxiliary variables are introduced to represent the cost of the hardware.

The full ILP formulation for joint scheduling is shown in Figure 4. The formulation consists of basic constraints (Equations 1 through 4) that ensure a valid schedule, and auxiliary constraints (Equations 5 through 8) that are used to compute the cost of the resulting hardware. Note that Equations 3, 6, 7, and 8 have non-linear components; these may be linearized using standard techniques.

The schedule validity constraints for individual loops are totally independent and represented using disjoint variables. However, there is only one set of variables that represent the hardware cost. For example, the cost of an FU is represented by a single variable, but depends on FU assignment of operations in all loops. Similarly, SRF costs are modeled using a single set of variables.

3.2 Union of Accelerators

The joint scheduler considers the effects on hardware cost of the scheduling alternatives for operations in all loops, and selects combinations of alternatives to minimize cost. This is computationally complex, because the number of possible schedules grows exponentially as the number of loops increases (since the scheduling alternatives of operations in different loops are independent). As a result, joint scheduling with ILP is impractical for large loop bodies or high numbers of loops.

Instead, the multi-loop scheduling problem may be divided into two phases to reduce its complexity. First, loops are scheduled individually and a single-function accelerator is designed for each loop; then, the accelerator datapaths are unioned into one multifunction datapath that supports all loops. This phase ordering can result in high quality designs, as the single-function accelerator costs are first minimized, and then hardware sharing across loops is exploited during the accelerator union. Synthesis runtimes are reduced significantly as it is no longer necessary to consider all schedules simultaneously.

The union phase is accomplished by selecting an FU and its corresponding SRF from each single-function accelerator and combining them into a single FU and SRF in the resultant accelerator. The new FU has the bitwidth and functionality to execute all operations supported by the individual FUs being combined. Similarly, the new SRF has sufficient width and depth to meet the storage requirements of any of the SRFs being combined. This process is repeated for the remaining FUs and SRFs until all of them have been combined. At this point, the resulting accelerator supports all of the functionality of the individual accelerators.

The cost of the multifunction accelerator is affected by the specific FUs and SRFs that are combined. For FUs, the ideal case occurs when FUs with identical functionality and bitwidth from k individual accelerators are combined into a single FU. This FU in the multifunction accelerator represents a cost savings (by a factor of k) over the single-function accelerators due to hardware sharing. When FUs with differing functionality are combined, no cost savings is achieved in the FUs, but this may enable cost savings in the corresponding SRFs. In the case of SRFs, maximal sharing occurs when two or more SRFs with similar bitwidths and numbers of registers are combined; in this case, only a single SRF is required in the multifunction accelerator where several were needed by the single-function accelerators.

Positional Union. The most straightforward union method is a *positional union*, where the FUs in each accelerator are ordered by functionality (multiple FUs with the same functionality have no par-

Constraints:	Time slots:	$\sum_{s=0}^{II_a-1} X_{i,s,a} = 1$	$\forall i \in \{1, N\}$, for each loop a	**(1)**
	Resources:	$\sum_{f=1}^{M_f} R_{i,f,a} = 1$	$\forall i \in \{1, N\}$	**(2)**
		$\sum_{i \in I_f} R_{i,f,a} \times X_{i,s,a} \leq 1$		**(3)**
	Dependences:	$t_{j,a} + d_{i,j,a} \times II_a - t_{i,a} \geq l_{i,j,a}$	$\forall (i,j,a) \in E_a$	**(4)**
	SRF Depth:	$LT_{i,a} \geq t_{i',a} - t_{i,a} + II_a \times d_{i,i',a} - l_{i,i',a} + 1$	$(i, i', a) \in E_a$	**(5)**
		$D_f \geq R_{i,f,a} \times LT_{i,a}$	$\forall i$ assigned to f	**(6)**
	FU/SRF Width:	$W_f \geq R_{i,f,a} \times BW_{i,a}$	$\forall i$ assigned to f	**(7)**
Objective:	Cost:	$Cost = \sum_f D_f \times W_f + fu\_cost_f \times W_f$		**(8)**
Definitions:	$t_{i,a} = \sum_{s=1}^{II_a-1} s \times X_{i,s,a} + II_a \times k_{i,a}$		$l_{i,j,a}$ - latency on edge (i,j)	
	$d_{i,j,a}$ - iteration distance on edge (i,j)		M_f - number of FUs of type f	

Figure 4: ILP formulation for joint scheduling.

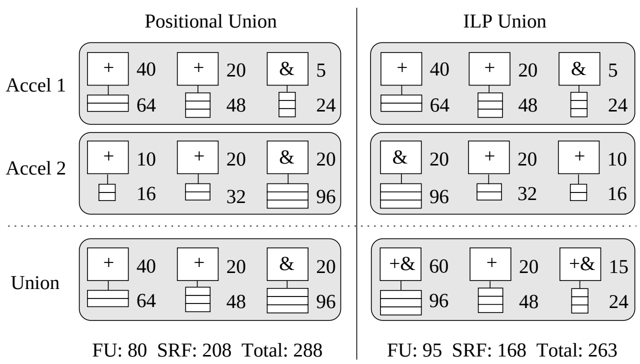

Figure 5: Example of union techniques. Two single-function accelerators, each with three FUs, are combined using positional (left) and ILP (right) methods. The cost of each FU and SRF is shown on its right.

ticular order), and FUs and SRFs in corresponding positions are selected for combination. The first FU and SRF in accelerator 1 are combined with the first FU and SRF in accelerator 2 to form the first FU and SRF in the multifunction accelerator, and so on. This union method yields good hardware sharing in the FUs, as FUs with identical functionality are combined, and the number of unique FUs in the resultant accelerator is therefore minimized. However, it does not account for FU width, nor does it attempt to improve hardware sharing in the SRFs. Sharing in the SRFs occurs by chance, if the dimensions of the SRFs being combined happen to be similar.

In Figure 5, an example of positional union is shown on the left. Here, each single-function accelerator has two ADD FUs and an AND FU. The FUs and SRFs have varying widths and depths (and thus varying costs), as shown to the right of each FU and SRF. The FUs of the two accelerators are combined according to functionality, and the resulting accelerator is shown on the lower left of the figure. Each FU and SRF in the unioned accelerator is sized to accomodate the corresponding FUs and SRFs from the single-function accelerators directly above them.

ILP Union of Accelerators. An improved union method to increase hardware sharing should consider all permutations of FUs (and corresponding SRFs) from the different loops, and select the permutation that results in minimal cost, considering both FU and SRF costs. This can be formulated as an ILP problem where binary variables are used to represent the possible pairings of FUs and SRFs from different loops. In this section, the combination of two loops to form a multifunction accelerator will be examined. Unions of more than two loops will be considered in the next section.

Assume that both single-function accelerators have N FUs. (If one accelerator has fewer FUs than the other, zero-width FUs may be added to make the number of FUs equal.) Then, N^2 binary variables x_{ij} may be used to represent the combination of FU i from the first loop with FU j from the second loop (along with their corresponding SRFs). For example, if $x_{11} = 1$, the first FUs in both accelerators will be combined in the multifunction accelerator. In addition, the following equations ensure that each FU is selected exactly once for combination with another FU:

$$\sum_{1 \leq j \leq N} x_{ij} = 1 \quad \forall i, \qquad \sum_{1 \leq i \leq N} x_{ij} = 1 \quad \forall j \tag{9}$$

Next, the objective function is defined so that the overall cost of the multifunction accelerator is minimized. This cost consists of two components: FU cost and SRF cost. Define variables F_{ij} as the cost of the FU resulting from the combination of FU i from loop 1 and FU j from loop 2. Depending on the functionality and bitwidth of these FUs, this cost can vary from the maximum cost of the two FUs up to their sum. Also, define variables R_{ij} as the cost of the SRF resulting from the combination of the SRFs corresponding to these two FUs. Then, the objective function is the minimization of the following:

$$Cost = \sum_{\forall i,j} (F_{ij} + R_{ij}) \times x_{ij} \tag{10}$$

By minimizing (10) subject to constraints (9), a combination of the FUs and SRFs of two loops is chosen that minimizes the cost of the multifunction accelerator.

The right side of Figure 5 shows an example of the ILP union. The single-function accelerators contain the same FUs and SRFs as in the positional union case, but they are combined differently. The resulting FU cost is higher than the FU cost from the positional union, because dissimilar FUs were combined and thus less hardware sharing in the FUs is achieved. However, the overall cost is lower as the SRF hardware is shared more intelligently.

Union of Multiple Accelerators. In the case where more than two loops are being combined, two strategies may be applied to extend the union technique. The first strategy is referred to as *pairwise union* and consists of first combining two accelerators to form a (temporary) multifunction accelerator. This temporary accelerator is then combined with the third single-function accelerator to form a new multifunction accelerator that supports all three loops. This process is continued, combining the new temporary accelerator with remaining single-function accelerators, until all desired loops have been combined into one multifunction accelerator.

The second method is referred to as *full union* and extends the ILP formulation given in the previous section. Given k loops, there are N^k binary variables $x_{i_1 \ldots i_k}$ that represent the combination of FUs i_1, ..., i_k from accelerators 1, ..., k, respectively. Constraints (9) and objective (10) are extended to reflect the additional loops. The solution consists of the N variables set to 1 which represent the specific combinations of FUs and SRFs which minimize the final hardware cost.

The advantage of full union is that it simultaneously considers all single-function accelerators together, and determines the best permutation of FUs to minimize the overall FU and SRF cost. How-

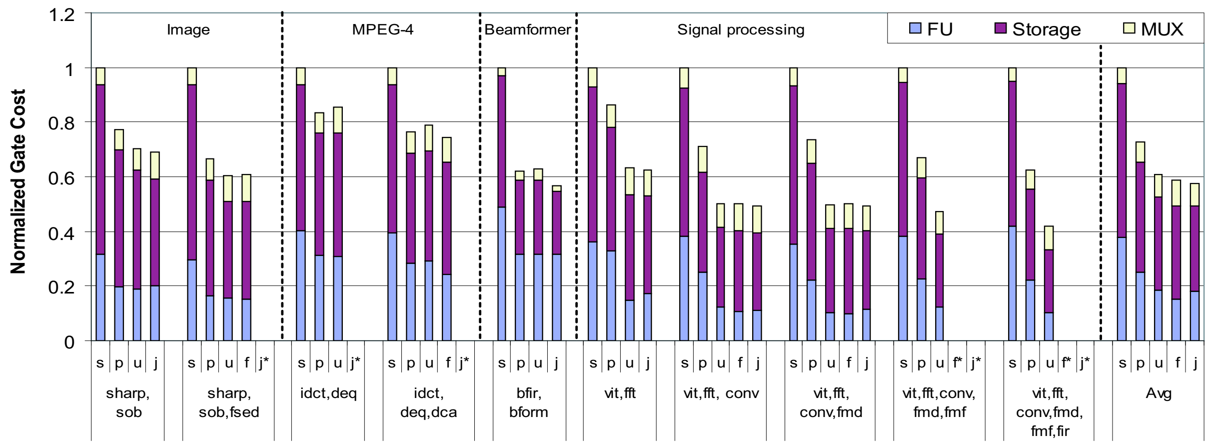

Figure 6: Gate cost of multifunction accelerators designed using sum (s), positional union (p), pairwise union (u), full union (f) (not shown for 2-loop combinations), and joint scheduling (j). * indicates the synthesis did not complete due to problem complexity.

ever, the downside is that the number of variables is exponential in the number of loops. Therefore, the full union quickly becomes infeasible for higher numbers of loops. Conversely, the pairwise union method may become trapped in local minima as it only considers two accelerators at a time during combining. We find experimentally that the pairwise union performs nearly as well as the full union in terms of final hardware cost, and its runtime is significantly faster due to its lower complexity.

4. EXPERIMENTAL RESULTS

Kernels from four different application domains are used to evaluate the loop accelerator designs. `Sharp`, `sobel`, and `fsed` are image processing algorithms. `Idct`, `dequant` and `dcacrecon` are computationally intensive loops extracted from the MPEG-4 application. `Bffir` and `bfform` are loops from the beamformer benchmark of the StreamIt suite [21]. `Viterbi`, `fft`, `convolve`, `fmdemodulator`, `fmfilter`, and `fir` are loops from the signal processing domain. To evaluate multifunction designs, loops from within the same application domain are combined, as they would likely be part of the same larger application accelerator.

For each machine configuration, we use the synthesis system described in this paper to design loop accelerators and generate RTL. The resulting Verilog is synthesized using the Synopsys design compiler in 0.18μ technology. All designs were synthesized with a 200-MHz clock rate. For all experiments, performance is held constant and is specified by the II value. A typical II is selected for each benchmark (for example, II=4 for `sobel` and II=8 for `idct`), and multifunction hardware is synthesized for combinations of benchmarks within the same domain. Gate counts are used to measure the cost of each accelerator configuration.

Figure 6 shows the cost in gates of multifunction loop accelerators designed using various scheduling methods. Each group of bars represents a benchmark combination, showing, from left to right, the sum of individual accelerators (s), the positional union of individual accelerators (p), the pairwise union (u), the full union (f), and the joint solution (j). When only two accelerators are combined, the full union is not shown as it is identical to the pairwise union. The bars are normalized to the sum of the cost of individual accelerators for that benchmark group. In addition, each bar is divided vertically into three segments, representing the contribution of FUs, storage, and MUXes to the overall cost. Since the joint solution relies on an ILP formulation with a large number of variables and constraints, it did not complete for some benchmark groups (labeled j$*$). Also, for groups containing more than four benchmarks, the full union becomes infeasible (labeled f$*$).

The first bar of each set represents current state-of-the-art multifunction accelerator design methodologies, i.e., creating single-function accelerators for each loop. Each single-function accelerator is designed using a cost-aware scheduler to minimize cost [5]. Thus, the difference between this bar and the other bars in each group represents the savings obtained by hardware sharing in multifunction designs. Since II is fixed for each benchmark, all multifunction designs in a group have the same performance, and hardware savings is essentially free. (However, note that additional multiplexers may increase critical path delay; this is discussed later in this section.) As the graph shows, the hardware savings is significant and increases with the number of loops. Up to 58% savings is achieved for the signal processing benchmark group, and 43% savings is achieved on average across all groups. Some groups (e.g. `idct` and `dequant`) exhibit less multifunction savings because the sizes of the two loops differ significantly, decreasing the amount of potential sharing.

On average, the pairwise and full union methods yield significantly lower-cost hardware than the positional union and are very close to the cost obtained with joint scheduling. However, in a few cases (most notably the benchmark groups containing `idct`), the positional union yields a lower cost than the more intelligent unions. This is due to two factors: first, MUX cost is not considered during the union phase and can affect the final cost; and second, the FU costs being minimized in the union phase are estimates, and actual FU costs may differ slightly when the design is synthesized into gates. In most benchmark groups, the pairwise union yields hardware that is equivalent in cost to the full union. Thus, pairwise union is an effective and tractable method of combining accelerators.

An area in which the multifunction accelerator does not improve on the individual accelerators is in the MUX cost. Although the multifunction accelerator has fewer FUs (and thus fewer MUXes) than the sum of individual accelerators, each MUX must potentially select from more inputs, as more operations execute on each FU.

Figure 7 shows the amount of hardware sharing in each of the multifunction accelerators synthesized in Figure 6. Each accelerator is represented by a bar which is divided vertically to show the fraction of gates used by 1 loop, 2 loops, etc. In general, lower cost accelerators have a higher fraction of gates used by multiple loops. Some interesting points to note are when sharing across loops increases, but the corresponding hardware cost does not decrease much (e.g. `vit-fft` when moving from union to joint). This occurs because, even though the joint scheduler is better able to share hardware across loops, the union method often has better hardware sharing within each loop (since the single-function accel-

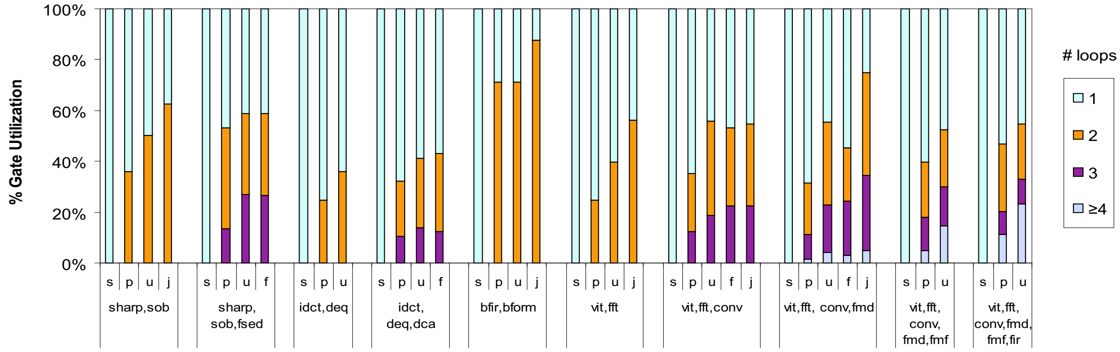

Figure 7: Degree of sharing of multifunction accelerator gates across loops.

erators are designed separately). Thus, hardware sharing still occurs in the union case, and the cost remains low.

Overall, runtimes for the synthesis system ranged from 20 minutes up to several hours on Pentium 4 class machines. The runtimes were dominated by the first step, generation of cost-efficient single-function accelerators; the runtime of the union phase was negligible for positional and pairwise unions, and up to 1 hour for the full union. The joint scheduler was allowed to run for several days; the bars missing from Figure 6 took longer than 5 days to run.

A side effect of multifunction designs is that additional interconnect is necessary to accomplish sharing in the datapath. The additional interconnect consists mostly of wider MUXes at the inputs of FUs. This can affect critical paths through the accelerator datapath and hence the maximal clock rate of the design. On average, the critical path delay in multifunction designs increased by 4% over the single-function designs. The largest critical path increase occurred in the signal processing group due to the increased resource sharing among the six loops. In this group the length of the critical path increased by 12% over that of the single-function accelerator. All of the multifunction designs were able to meet the target clock rate of 200 MHz.

5. CONCLUSION

This paper presents an automated, compiler-directed system for synthesizing accelerators for multiple loops. The synthesis system builds an abstract architecture based on the compute requirements of the loops, modulo schedules the loops, and then derives the datapath and control path for the accelerator. Cost savings is achieved by sharing hardware across loops while meeting the performance requirements of each loop. Union methods are presented to reduce the complexity of the scheduling problem. It is shown that intelligently unioning single-function accelerators yields multifunction accelerators that are nearly optimal in cost. By evaluating accelerators designed for various application domains, average hardware savings of 43% are realized due to sharing of execution resources and storage between loops, with individual savings of up to 58%.

6. ACKNOWLEDGMENTS

Thanks to the anonymous referees who provided excellent feedback. This research was supported in part by ARM Limited, the National Science Foundation grants CCR-0325898 and CCF-0347411, and equipment donated by Hewlett-Packard and Intel Corporation.

7. REFERENCES

[1] K. Bondalapati et al. DEFACTO: A design environment for adaptive computing technology. In *Proc. of the Reconfigurable Architectures Workshop*, pages 570–578, Apr. 1999.

[2] T. Callahan, J. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *IEEE Computer*, 33(4):62–69, Apr. 2000.

[3] C. Ebeling et al. Mapping applications to the RaPiD confi gurable architecture. In *Proc. of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 106–115, Apr. 1997.

[4] A. E. Eichenberger and E. Davidson. Effi cient formulation for optimal modulo schedulers. In *Proc. of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 194–205, June 1997.

[5] K. Fan, M. Kudlur, H. Park, and S. Mahlke. Cost sensitive modulo scheduling in a loop accelerator synthesis system. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 219–230, Nov. 2005.

[6] D. D. Gajski et al. *High-level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[7] M. Gokhale and B. Schott. Data-parallel C on a reconfi gurable logic array. *Journal of Supercomputing*, 9(3):291–313, Sept. 1995.

[8] M. Gokhale and J. Stone. NAPA C: Compiler for a hybrid RISC/FPGA architecture. In *Proc. of the 6th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 126–137, Apr. 1998.

[9] S. Goldstein et al. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proc. of the 26th Annual International Symposium on Computer Architecture*, pages 28–39, June 1999.

[10] R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 85–94, Nov. 1994.

[11] D. Herrmann and R. Ernst. Improved interconnect sharing by identity operation insertion. In *Proc. of the 1999 International Conference on Computer Aided Design*, pages 489–493, 1999.

[12] Z. Huang, S. Malik, N. Moreano, and G. Araujo. The design of dynamically reconfi gurable datapath coprocessors. *ACM Transactions on Embedded Computing Systems*, 3(2):361–384, 2004.

[13] S. Memik et al. Global resource sharing for synthesis of control data fbw graphs on FPGAs. In *Proc. of the 40th Design Automation Conference*, pages 604–609, June 2003.

[14] S. Note, W. Geurts, F. Catthoor, and H. D. Man. Cathedral-III: Architecture-driven high-level synthesis for high throughput DSP applications. In *Proc. of the 28th Design Automation Conference*, pages 597–602, June 1991.

[15] N. Park and A. C. Parker. Sehwa: A software package for synthesis of pipelines from behavioral specifi cations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(3):356–370, Mar. 1988.

[16] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavorial synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–679, June 1989.

[17] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, Nov. 1994.

[18] R. Schreiber et al. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing*, 31(2):127–142, 2002.

[19] G. Snider. Performance-constrained pipelining of software loops onto reconfi gurable hardware. In *Proc. of the 10th ACM Symposium on Field Programmable Gate Arrays*, pages 177–186, 2002.

[20] L. Stok. Interconnect optimisation during data path allocation. In *Proc. of the 1990 European Design Automation Conference*, pages 141–145, 1990.

[21] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of the 2002 International Conference on Compiler Construction*, pages 179–196, 2002.

[22] C. Tseng and D. P. Siewiorek. FACET: A procedure for automated synthesis of digital systems. In *Proc. of the 20th Design Automation Conference*, pages 566–572, June 1983.

[23] K. Wakabayashi and T. Yoshimura. A resource sharing and control synthesis method for conditional branches. In *Proc. of the 1989 International Conference on Computer Aided Design*, pages 62–65, 1989.

[24] M. Wazlowski et al. PRISM-II compiler and architecture. In *Proc. of the 1st IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 9–16, Apr. 1993.