Compiler-directed Synthesis of 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

Complex algorithms and increased functionality are expanding the computation demands of embedded systems. Hardware accelerators are commonly used to meet these demands by executing critical application loop nests in custom logic, achieving performance requirements while minimizing hardware cost. Traditionally, these loop accelerators are designed in a single-function manner, wherein each loop nest is implemented as dedicated hardware. This papers focuses on hardware sharing across loop nests by creating multi-function loop accelerators, or accelerators capable of executing multiple algorithms. A compiler-based system for automatically synthesizing multi-function loop accelerator architectures from high level specifications is presented. We compare the effectiveness of three synthesis approaches with varying levels of complexity: unioned, phase-ordered, and integrated. Experiments show that intelligently designed multi-function accelerators achieve substantial hardware savings over their single-function counterparts on loop kernels taken from multimedia and signal processing domains.

1. INTRODUCTION

The markets for wireless handsets, PDAs, and other portable devices continue to grow explosively. The growth is fueled by new functionality, added capabilities, and higher bandwidth. These devices demand higher performance and more energy-efficient computer systems to satisfy the 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 hardware accelerators from high-level specifications has the potential to solve these problems.

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 functionality, 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 extremely inefficient design. Processor-based solutions are the obvious approach to creating multi-purpose designs due to their inherent programmability. However, processor-based solutions do not offer the performance and energy efficiency of accelerators as there is an inherent overhead to instruction-based execution.

In this paper, the focus is on automatic design of multi-function loop accelerators from high-level specifications. The goal is to maintain the efficiency of single-function accelerators, while exposing a large number of opportunities for hardware sharing across multiple algorithms. The inputs to the system are the target applications expressed in C, the desired throughput, and the available memory bandwidth. 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 [21]. Accelerators are synthesized by mapping the algorithm to a simple VLIW processor and then extracting a stylized accelerator architecture from the compiler mapping.

To accomplish multifunction design, the single-function system is extended using three alternate strategies. First, the simplest strategy is to create individual accelerators for each algorithm without any consideration for the other algorithms. The data and control paths for the individual accelerators are then unioned together to create a single design capable of all algorithms. The second approach is to iteratively synthesize accelerators for each algorithm in a phase-ordered manner. For the first algorithm, a single-function accelerator is synthesized. For each subsequent algorithm, synthesis accounts for with pre-existing hardware created for the prior algorithms and attempts to augment the design with as little additional hardware as possible to support the desired performance of that algorithm. Finally, the third approach is to perform integrated, cost-aware synthesis of all algorithms. We employ an integer linear programming formulation to find a solution with optimal estimated cost. Each successive strategy represents a more complex approach and hence the ability to extract more opportunities for sharing. But as a consequence, the successive strategies require more synthesis time and memory usage, which may become prohibitive for large algorithms.

2. RELATED WORK

Datapath synthesis is a field that has been studied for many years. The basic techniques have been well established [7]. Cathedral III represents a complete synthesis system developed at IMEC and illustrates one comprehensive approach to high-level synthesis [17]. Force-directed scheduling is used to synthesize datapaths for ASIC design [19]. The Sehwa system automatically designs processing pipelines from behavioral specifications [18]. Clique based partitioning algorithms were developed in the FACET project to

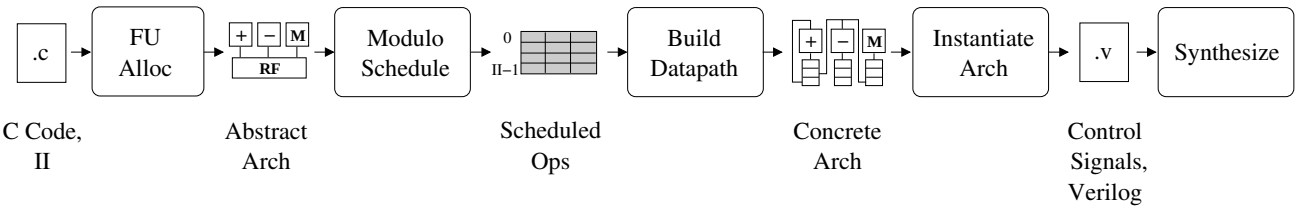

Figure 1: Compiler-directed design system flow.

jointly minimize function unit and inter-function unit communication costs [24].

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 [9] that was subsequently productized as the NAPA system [10]. Other automatic compiler systems for FPGA-based platforms include Garp [3], PRISM [25], Cameron [14], Match [13], DEFACTO [2], and a SUIF-based system [1]. Various programming models have been proposed to provide a more efficient FPGA implementation, including Transmogrifier C [8], Picasso [26], and Machines [22]. Compilation for architectures consisting of predefined function units and storage with reconfigurable interconnect have been investigated, including RaPiD [4] and PipeRench [11]. Generation of more efficient designs by sharing hardware across basic blocks was recently proposed [16]. Cost sensitive scheduling, used within the synthesis system to reduce hardware cost, has been studied in the context of interconnect minimization in [23, 15].

This paper extends prior work in an orthogonal direction by investigating multi-function 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. The compiler synthesis strategy has the most similarity to the PICO system [21]. But, PICO is focused on a single-function accelerators.

3. SINGLE FUNCTION DESIGN SYSTEM

The overall flow of the proposed design system is presented in Figure 1. The design system takes an application loop and a performance requirement, specified as the initiation interval (II) or the number of cycles between initiating successive loop iterations, as input. Additional constraints such as clock rate or bandwidth to memory may also be specified. Based on the number and types of operations in the loop and these cost and performance constraints, an abstract architecture for a hypothetical VLIW processor is created. This abstract architecture represents a high-level view of the accelerator's functionality, that can be effectively compiled to, and exposes resource sharing opportunities within the operations of a single loop. This abstract architecture also enables the compiler to generate a modulo schedule. From this modulo schedule, the accelerator datapath is filled in with function units (FUs), storage elements and interconnect. Finally, the control path is generated and the accelerator is instantiated in RTL and synthesized. Each of these steps will be discussed in detail in the following sections with an example from `sobel`, an edge-detection algorithm.

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); each cycle that the FU produces a new value, the contents of the registers shift downwards to the next register. The entries in an SRF therefore contain the values produced by the corresponding FU in the order

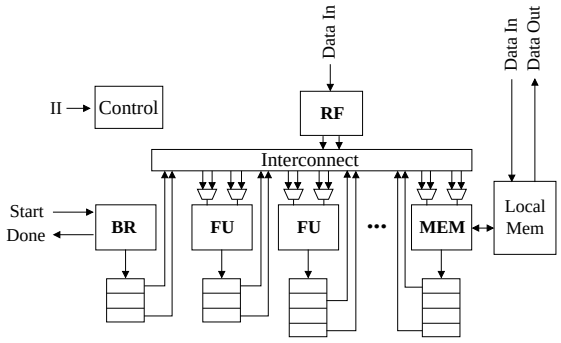

Figure 2: Loop accelerator schema.

they were computed. Wires from the registers back to the FU inputs allow data transfer from producer to consumer. 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. Other than this counter, no control signals are needed to address the SRFs.

Literals and static live-in register values cannot be stored in the SRFs. Therefore, literals are hard-wired to the appropriate FU inputs, and live-in values are supplied by a central register file which is connected to the inputs of FUs that require them. FUs which access memory are connected to a local memory structure such as a scratchpad, cache, or stream buffer. The host processor can let the loop accelerator begin execution by setting some control registers like the loop counter (*LC*) and asserting the *start* signal. When the loop execution is complete, the branch function unit asserts a *done* signal to the host processor.

3.1 Architecture Synthesis

The first step in the architecture synthesis process is the creation of the 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 problem of *FU allocation* is to come up with a mix of FUs that minimizes cost while providing enough resources to meet the performance constraint. In the simplest case where each operation can be executed by only one type of FU, $\lceil compatible\_ops/II \rceil$ instances of each FU type should be created.

However, operations can generally be executed by multiple types of FUs. For example, both ADD and ADDSUB units may be available, and the best choice of FUs depends on the number of ADD and SUB operations in the loop. In this case, the FU allocation problem becomes more complex and can be formulated as an integer linear program, minimizing the sum of the FU costs while

```
for (i = 0; i < N1; i++) {
  for (j = 0; j < N2; j++) {
    t00 = x[i  ][j  ];
    t01 = x[i  ][j+1];
    t02 = x[i  ][j+2];
    t10 = x[i+1][j  ];
    t12 = x[i+1][j+2];
    t20 = x[i+2][j  ];
    t21 = x[i+2][j+1];
    t22 = x[i+2][j+2];

    e1 = ((t00 + t01) + (t01 + t02)) -
       ((t20 + t21) + (t21 + t22));
    e2 = ((t00 + t10) + (t10 + t20)) -
       ((t02 + t12) + (t12 + t22));
    e12 = e1*e1;   e22 = e2*e2;
    e = e12 + e22;
    if (e > threshold) tmp = 1;
    else tmp = 0;
    edge[i][j] = tmp;
  }
}
```

II = 4

FU	#
ADD	5
ADDSUB	1
MEM	3
MPY	1
CMP	1

Figure 3: `sobel` source code and result of FU allocation with II=4.

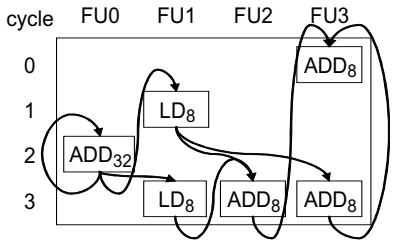

Figure 4: A portion of the `sobel` modulo scheduled loop. Edges represent dataflow between operations.

supporting all of the operations. Figure 3 shows the result of FU allocation for `sobel` with II=4. The operations in the loop include 22 ADD and 2 SUB operations, which are covered by the 5 ADD and 1 ADDSUB units.

Next, the loop is modulo scheduled to the abstract architecture. The scheduler is a backtracking modulo scheduler which assigns operations in priority order to the resources in the abstract architecture [20]. The scheduler is augmented with a hardware cost model, detailed in Section 3.2, in order to make scheduling decisions that will reduce the cost of the resulting hardware. At the completion of this phase, all of the loop operations are bound to FUs and time slots, and therefore the producer-consumer relationships between FUs have been determined. Figure 4 shows some operations from the modulo schedule for `sobel`. Note that the number associated with each operation indicates its width, and the width of a function unit is set to the width of the largest operation assigned to it.

Thus the virtual FUs of the abstract architecture, concretized by operation assignments, directly become the FUs of the loop accelerator. The rest of the accelerator datapath is derived from the producer-consumer relationships in the modulo schedule. Wires connect an SRF entry at the output of a producing FU to the input of a consuming FU. The SRF entry that should be connected is determined from the difference in execution time between the producer and consumer. More specifically, the register number that should be connected to transfer a value from producing operation p to consuming operation c is:

$$time(c) - time(p) + iteration\_distance(p,c) * II - latency(p)$$

The bitwidths of the FUs and SRFs are determined from the modulo schedule as well; they are the maximum bitwidth of the

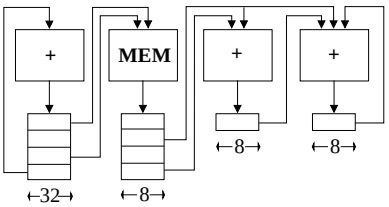

Figure 5: Datapath derived from the modulo schedule shown in Figure 4.

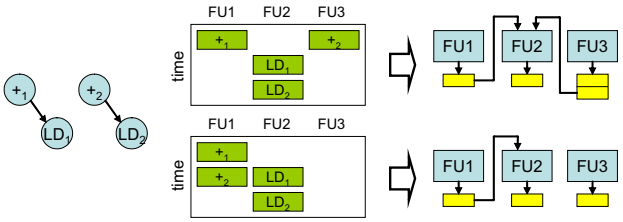

Figure 6: Effect of schedule on wire cost.

data that must be computed in the FU or contained in the SRF. The depth of an SRF is determined from the longest lifetime of the values produced by the corresponding FU. Figure 5 shows the SRFs and connections resulting from the scheduled operations in Figure 4. Note that three of the four SRFs can be sized to 8-bit width. Finally, the live-in register values are allocated to the central register file, and wires are created to connect the central register file with FUs that have live-in operands.

The result of architecture synthesis is a high-level representation of the accelerator architecture consisting of the major components (FUs, SRFs, and central register file) and the interconnections between them.

3.2 Cost Sensitive Scheduling

Typically, the goal of a scheduler is to maximize the performance of an application on a given machine. However, in the accelerator design system the application is scheduled on an abstract architecture, and then the accelerator datapath is synthesized based on this schedule. Therefore, it is important that the scheduler be aware of the impact of its decisions on the the cost of the resulting machine.

A modulo scheduler selects a scheduling alternative, or assignment of FU and time slot, for each operation. The choice of scheduling alternative for an operation has a significant impact on the cost of the resulting machine. A standard, cost-unaware scheduler chooses alternatives naïvely (for example, scheduling all operations as early as possible on an arbitrary free FU). In Figure 6, assume the two pairs of operations are 32 bits wide. A cost-unaware modulo scheduler might produce the upper schedule, which requires 64 wires, while the lower schedule would have required only 32.

3.2.1 Greedy Modulo Scheduler

The basic scheduling algorithm used in our system is based on [20]. The scheduler goes through the operations in the loop in a dependence height based priority order. For every operation, the scheduler chooses a FU and a time slot. Since the dependences could be cyclic, conflicts can occur while scheduling, i.e., the scheduler may fail to find a valid time slot and FU for an operation. Backtracking is used to resolve these conflicts. Some previously scheduled operations are unscheduled to make room for conflicting operations.

Note that the choice of FU and time slot affects the cost of the

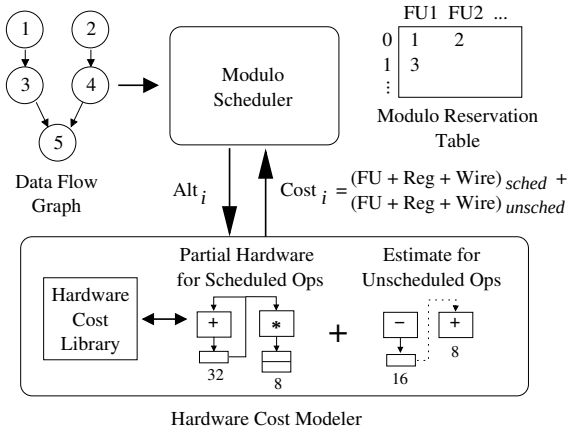

Figure 7: Cost aware scheduling framework.

resulting machine. Thus, a cost aware scheduling framework is used, shown in Figure 7. The main component of this framework is the hardware cost model. This hardware model represents the cost of FUs, SRFs, and interconnect wires. Note that the final cost of hardware can be computed only when all operations have been scheduled. However, the scheduler needs to know the cost impact of a scheduling alternative when it is in the middle of the scheduling process. For this reason, the hardware cost model is able to represent the cost of a partial machine, that is, the cost of hardware resources required to support execution of just the scheduled operations. In addition, the cost modeler can estimate the cost of hardware that would be required to support the remaining, unscheduled operations.

To choose the best local alternative, the greedy modulo scheduler makes queries about the machine cost to the hardware cost modeler. The cost modeler returns a cost estimate that includes both the partial machine cost as well as the estimated cost of unscheduled operations. Based on this cost, the scheduler chooses an alternative and schedules the operation on that particular FU and time slot. The scheduler informs the hardware cost modeler about this decision so that the partial machine can be updated. Scheduling an operation can change the width of some FU, and it can add some new connection between a register and the FU. When the modulo scheduler unplaces an operation during a backtracking step, it also informs the hardware cost modeler about the unplaced operation, so that the partial machine can be updated.

3.2.2 ILP-based Optimal Scheduler

Optimal modulo schedulers based on integer linear programming have been extensively studied in [12] and [5]. The objective functions in these formulations have been compiler-oriented in nature, for example minimum schedule length, minimum register requirement, etc. However, in our design system, since the schedule determines the cost of the resulting hardware, an objective function has to be formulated which reflects the FU cost, storage cost and wire cost. Due to space constraints, only an high level description of the formulation is provided here.

The software setup for ILP-based scheduler is similar to the one shown in figure 7. However, instead of using a detailed hardware cost modeler, different components of hardware costs are directly built into the ILP formulation. The ILP formulation employs a binary variable to represent the assignment of an operation to a particular row in the modulo reservation table. Besides these II binary variables, an integer variable representing the stage in which the operation is scheduled, completely describes the schedule time of an operation. The basic constraints which ensure a valid schedule are identical to the ones presented in [5]. In addition, our formulation employs binary variables to represent the assignment of an operation to a particular FU. Additional constraints are introduced to get a valid FU assignment. The FU widths can be derived from these binary variables, as the width of an FU is the maximum bitwidth of operations assigned to it. The depth of shift registers associated with an FU is derived from the difference in schedule times of operations assigned to the FU and their consumer operations. The cost of storage structures are derived from the width of FUs and depth of the shift register files. More detailed descriptions and equations can be found in [6].

3.3 Architecture Instantiation

The goal of this step is to generate a Verilog realization of the accelerator from the high-level architecture created in the previous section. This is done by lowering each module into primitive modules that have pre-defined behavioral Verilog descriptions. After the primitive modules are identified, connections between modules are made and MUXes are introduced for multiple-producer connections. For example, an FU that supports ADD, SUB and SHR is lowered into a set of primitive modules which includes an ADDSUB module, SHR module and a MUX combining two outputs.

Due to the diversity in width of the datapath and the use of separate storage elements for each FU, two types of customized MUXes are introduced in our design. One is the extension MUX that can do signed or unsigned extension on its inputs depending on which type of data is transferred through it. Having this extension MUX in the datapath enables removing most of the sign extension operations in the original program as extension is done implicitly. The other type of MUX is the data-merge MUX which solves the problem of multiple producers feeding a single consumer under disjoint predicates. In a conventional architecture with a centralized register file, this is not a problem as all the producers write into the same structure. However, in our dedicated shift register file scheme, multiple producers may conditionally write to different SRFs and depending on which FU executes the producer, the valid one is known only at run time. To address this problem, we extended the width of each SRF by one bit that indicates whether the data contained in the register entry is valid. Using this valid bit information, the data-merge MUX selects the valid data at runtime. It is legal for more than one entry to have its valid bit set to 1. In this case, the most recently generated valid value is selected.

Based on the datapath that is instantiated, the control path is generated for each loop. To reduce global wiring of control signals, we employ a distributed control scheme wherein each high-level module has its own logic that internally generates control signals for all of the enclosed primitive modules. We currently utilize three levels of control logic: FU control activates the appropriate primitive FU with the proper functionality and sets any internal MUX selects; Cluster control (a cluster is defined as the set of tightly interconnected FUs and SRFs) converts the II value to generate high-level FU opcodes and sets the input MUXes select signals; and, the top level control which generates the II counter value.

A subset of the final lowered datapath for `sobel` is presented in Figure 8. The rightmost FU is realized with two primitive FUs, a MPY and a CMP. The first output of the CMP is shared with the output of the MPY. In addition, input MUXes are added when multiple wires share the same FU input port, as shown in the fourth FU from the left. Each shift register file contains valid bits as described above.

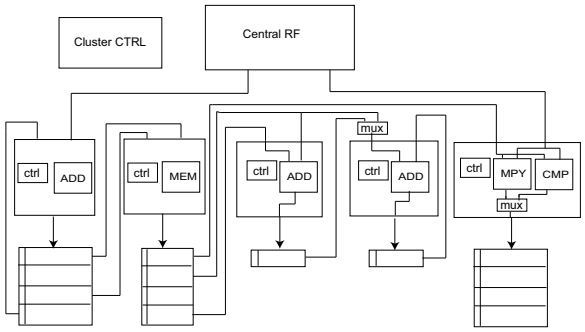

Figure 8: Lowered datapath for `sobel`.

II = 4

FU	#
AND	1
MEM	1
SHL	1
SHR	1
MOV	2

II = 4

FU	#
ADDSUB,SHL	1
ADD,MOV	1
AND,MPY	1
MEM	1
MEM,MOV	1
MEM,SHR	1
ADD	4
CMP	1

Figure 9: FU allocation for `fsed` (left) and FU mix supporting both `sobel` and `fsed` (right).

4. MULTIFUNCTION ACCELERATORS

Multifunction design refers to generalizing the loop accelerator design to support two or more separate loop nests. One obvious approach to creating a multifunction accelerator is to separately design accelerators for the individual loops, and then stamp out these loop accelerators side by side. While this may save packaging costs over creating completely separate accelerators, more hardware sharing can be achieved. By creating an accelerator with a single datapath that can support multiple loops, a large amount of hardware sharing is possible while continuing to meet throughput constraints for 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 is a superset of the resources required for the individual accelerators. 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 sharing is done 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.

Three methods to extend the single-function design system to create multifunction accelerators are described in the remainder of this section.

4.1 Union of Accelerators

One multifunction accelerator synthesis strategy is to first design a single-function accelerator for each loop as described in Section 3, and then to combine these accelerators to form the multifunction accelerator. The multifunction accelerator datapath is the union of the single-function datapaths, and therefore hardware cost savings over the naïve sum of accelerators can be realized by taking advantage of hardware sharing across loops.

The unioning phase is accomplished by selecting an FU and its corresponding shift register file (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 unioned. At this point, the resulting accelerator supports all of the functionality of the individual accelerators.

Note that the most sharing of SRFs occurs when two or more large SRFs with similar dimensions are combined; in this case, only a single SRF is required in the multifunction accelerator where several were needed by the single-function accelerators. Similarly, the most FU sharing occurs when FUs with similar functionality and bitwidth are combined. However, it can be advantageous to combine FUs with dissimilar functionality. In this case, hardware sharing in the FU does not improve, but the combination may enable more sharing in the corresponding SRFs.

The simplest unioning method is a *positional* union, where the FUs in each accelerator are ordered by functionality (multiple FUs with the same functionality are unordered), and FUs and SRFs in corresponding positions are selected for combination. For example, 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 unioning method yields good hardware sharing in the FUs. However, hardware sharing in the SRFs occurs by chance, i.e., if the dimensions of the SRFs being combined happen to be similar. Note that in the worst case, it is possible for the unioned SRF to have a higher cost than the sum of two individual SRFs: when one SRF is wide and has few entries and the other is narrow with many entries, the union will both be wide and have many entries, with greater total area than the sum of the two SRFs.

An improved unioning method to increase hardware sharing should consider all permutations of FUs from the different loops, and union the permutation that results in minimal cost. This can be formulated as an ILP problem where binary variables are used to represent the possible pairings of FUs/SRFs from different loops. The objective function is set such that the set of pairings with minimal cost is chosen. Unlike the positional unioning, ILP unioning is able to actively improve both FU and SRF hardware sharing, rather than allowing SRF sharing to come about by chance. In addition, ILP unioning can combine dissimilar FUs (which increases FU cost relative to the positional union) if it results in significant SRF cost savings.

The disadvantage of the unioning strategy is that each loop is scheduled without knowledge of the other loops. Once the loops are scheduled, their individual FU and storage requirements are fixed, and the subsequent unioning phase cannot change the schedules to further improve hardware sharing across loops. Therefore, this strategy is unable to take advantage of some hardware sharing opportunities.

4.2 Phase Ordered Scheduling

The second approach to handling multiple loops is to use phase ordering. The loops are scheduled individually; however, they are scheduled in order, such that each loop can account for the hardware created by the schedule of the previous loop. Thus, the first loop is scheduled using the cost sensitive scheduler described in Section 3.2. When the second loop is scheduled, rather than starting with an empty virtual hardware model, the hardware resulting from the first loop is used. The cost sensitive scheduler therefore naturally attempts to reuse hardware from the first loop as it minimizes cost during scheduling of the second loop. This continues until all loops are scheduled. Currently, the greedy scheduling method (Section 3.2.1) is used as it can naturally account for the preexisting hardware.

One condition of the phase ordered approach is that all loops should be scheduled onto the same abstract architecture in order to allow the datapath to be incrementally updated as loops are scheduled. To generate the abstract architecture for the multifunction accelerator, FU allocation is first performed on each loop individually. As described in Section 3.1, FU allocation for each loop generates an FU mix capable of executing the individual loops. These virtual sets of FUs are unioned together to get the minimum set of FUs which can support execution of all the loops. Figure 9 shows the minimum set of FUs required to execute `sobel` and `fsed` (a halftoning algorithm).

Note that the FU allocation for individual loops can result in FUs which are mutually exclusive between two loops. For example, Figures 3 and 9 show that the MPY unit is used by `sobel` but not by `fsed`. Similarly, an AND unit is used by `fsed` and not by `sobel`. We take advantage of this mutual exclusivity to reduce the number of shift register files. Note that both MPY and AND FUs have to be present in the multifunction accelerator which can execute both `fsed` and `sobel`. But only one of the FUs will be active at any time. Therefore, it is enough to synthesize one SRF to hold the values produced by both of these FUs. These mutually exclusive virtual FUs are *pre-grouped* and treated as a single entity for later steps.

In this phase ordered scheduling approach, it is clear that the order in which loops are scheduled affects the cost of the final hardware. The modulo schedule for each loop accounts for the hardware from previously scheduled loops, but cannot account for loops not yet scheduled. In this system, loops are scheduled in order from largest hardware cost to smallest hardware cost (hardware cost can be approximated by synthesizing a single-function accelerator for the loop). It was found that this ordering gives good solutions because there is greater potential for hardware sharing earlier in the phase ordered scheduling process.

4.3 Integrated Scheduling

When scheduling multiple loops, it is necessary to consider the cost of the hardware resulting from the combined schedules of all loops. Another synthesis approach which accounts for this is to jointly schedule all loops simultaneously. This entails considering the effects on hardware cost of the scheduling alternatives for operations in all loops, and selecting combinations of alternatives to minimize cost. In general, this problem is quite 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). In our system, the integrated scheduling uses an ILP formulation, which is based on the one used for scheduling single loops, described in Section 3.2.2. 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 storage and wire costs are modeled using a single set of variables.

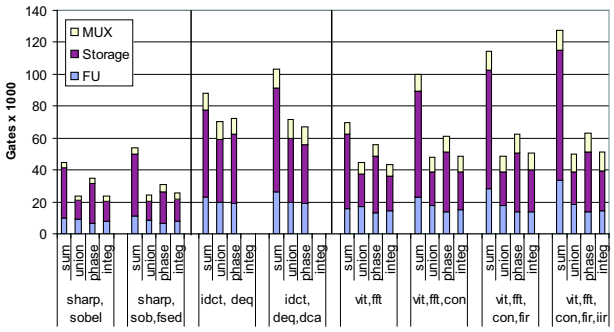

Figure 10: Gate cost of multifunction accelerators.

5. EXPERIMENTAL RESULTS

Kernels from three different application domains are used to evaluate the loop accelerator designs. From image processing, `sharp`, `sobel`, and `fsed` algorithms are examined. `sharp` performs image sharpening, `sobel` performs edge detection, and `fsed` performs halftoning. `idct`, `dequant` and `dcacrecon` are computationally intensive loops extracted from the MPEG-4 application. From the signal processing domain, the following kernels are evaluated: `viterbi`, `fft`, `convolve`, `fir` and `iir`. These kernels are commonly used in applications such as wireless networking.

For each machine configuration, we use the compiler-directed architecture synthesis system described in this paper to design the loop accelerator and generate RTL. The resulting Verilog is synthesized using the Synopsys design compiler in 0.18μ technology. A 200-MHz clock rate is assumed. 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 10 shows the cost in gates of multifunction loop accelerators designed using the methods described in this paper. Each group of four bars represents a benchmark combination, showing, from left to right, the sum of individual accelerators, the intelligent union of independently designed accelerators (Section 4.1), the phase-ordered design (Section 4.2), and the integrated ILP solution (Section 4.3). Each bar is vertically divided into three segments, representing the contribution of FUs, storage, and MUXes to the overall cost. Since the integrated solution relies on the NP-complete ILP formulation, it did not complete for some benchmark groups. Thus, the cost of the integrated solution is not shown for the MPEG-4 group of benchmarks.

The first bar of each set represents current state-of-the-art multi-loop accelerator design methodologies, i.e. creating single-function accelerators for each loop. Thus, the difference between this bar and the other three bars in each group represent the savings available by synthesizing multifunction designs. As the figure shows, this savings is significant, especially as the number of loops increases.

One thing to notice is that the FU cost in the multifunction accelerator increases very little as more loops are mapped onto the same

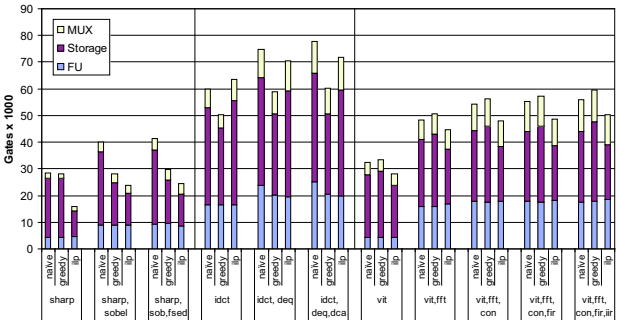

Figure 11: Effect of different cost sensitive schedulers.

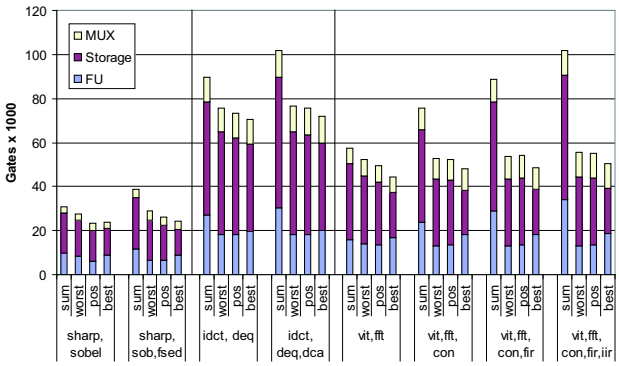

Figure 12: Effect of different unioning methods.

hardware. As the hardware supports more loops, chances are high that the FU requirements of a new loop can be met by FUs already present in the accelerator.

The cost of the storage area in the multifunction accelerator is also reduced through sharing. It can be seen that in all of the multifunction accelerators, the cost of the storage is significantly less than the sum of the storage required in the individual accelerators. In the signal processing domain, for example, the accelerator supporting all five benchmarks has a storage cost of about 25K gates, while the total cost of the storage without sharing is about 82K gates.

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 select among a potentially larger set of data locations, as more operations execute on each FU.

An important observation is that the union of independent accelerators has virtually the same cost as the accelerator designed with the integrated ILP solution. Hence, intelligently unioning single-function accelerators can give a nearly optimal solution. This is significant because the integrated solution does not scale with the number of loops combined in a multifunction accelerator. Another point is that generally the union of independent accelerators has lower cost than the accelerator designed with phase ordering. This is because the independent accelerators are scheduled optimally, while a greedy scheduling algorithm is used for the phase ordered accelerator.

Figure 11 shows the effectiveness of the different cost-sensitive schedulers discussed in Section 3.2. The benchmark groups are the same as those in the previous figure. For each group, accelerators are designed independently using naïve, greedy, and ILP schedulers, and then unioned intelligently as discussed in Section 4.1. The first set of benchmark groups (from the image processing domain) shows the expected result: the greedy cost-sensitive scheduler reduces hardware cost from the naïve scheduler, and the ILP solution further reduces the cost beyond the greedy scheduler. The next set of benchmarks (from MPEG-4) shows the anomaly that the ILP solution is very poor. This is because the problem is NP-complete by nature, and the ILP solver did not complete when scheduling the `idct` benchmark even when running for several days. An intermediate, non-optimal solution was taken for `idct`, which negatively impacted the hardware cost of accelerators for groups of benchmarks which included `idct`. Finally, the third set of benchmarks (signal processing) show that the greedy scheduler can perform worse than the naïve scheduler. This is because currently the greedy scheduler can inaccurately estimate the hardware cost of unscheduled operations, which can lead to bad scheduling decisions as in the case of the `viterbi` benchmark. Thus, the benchmark groups containing `viterbi` have higher hardware cost. Improving the greedy scheduler is an area of future work.

Figure 12 shows the effect of the unioning method when combining independently designed accelerators. The first bar for each benchmark group shows the cost of hardware obtained by just summing up the accelerators of individual benchmark. Clearly, it has the worst cost because no hardware is shared across accelerators. The second bar shows the hardware cost when the data paths are unioned in the worst possible manner. This is achieved by maximizing the objective function in the ILP formulation of unioning described in Section 4.1. This result is included to highlight the ineffectiveness of positional unioning (third bar). For example, the hardware cost for positional unioning and worst unioning is same for the `vit-fft-con-fir` group of benchmarks. For other benchmark groups, positional unioning results in hardware cost lower than that achieved by worst unioning. But as described in Section 4.1, this is more by chance than by design. The last bar shows the hardware cost achieved by ILP based smart unioning, and it consistently gets better cost compared with other forms of unioning.

A side-effect in multifunction designs is that additional interconnect is necessary to accomplish sharing in the data path. The additional interconnect consists mostly of wider MUXes at the inputs of FUs. This can affect the critical path through the accelerator data path and hence the maximal clock rate of the design. The additional interconnect increased the critical path from 2.3% to 6.8% with an average of 4.3% over the longest critical path of the single-function designs. However, all multifunction designs were able to meet the target clock rate of 200 MHz.

6. CONCLUSION

This paper presents an automated, compiler-directed system for synthesizing accelerators for modulo scheduled loops. The synthesis system builds an abstract architecture based on the compute requirements of the loop, modulo schedules the loop, and then derives the datapath and control path for the accelerator. The system can be used for synthesizing custom accelerators that can run multiple loops, utilizing hardware sharing in order to realize cost savings over synthesizing individual loop accelerators while meeting the performance requirements of each loop. Three methods of synthesizing multifunction accelerators are presented: unioning, phase ordering, and integrated. It is shown that intelligently unioning single-function accelerators yields multifunction accelerators that

are nearly optimal in cost. By evaluating multifunction accelerators designed for various application domains, hardware savings of up to 60% are realized due to sharing of resources and storage between loops.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] J. Babb et al. Parallelizing applications into silicon. In *Proc. of the 7th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 70–80, Apr. 1999.

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

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

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

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

[6] K. Fan, M. Kudlur, H. Park, and S. Mahlke. Cost sensitive modulo scheduling. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, Nov. 2005.

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

[8] D. Galloway. The transmogrifier C hardware description language and compiler for FPGAs. In *Proc. of the 3rd IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 136–144, Apr. 1995.

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

[10] 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.

[11] 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.

[12] 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.

[13] M. Haldar et al. A system for synthesizing optimized FPGA hardware from Matlab. In *Proc. of the 2001 International Conference on Computer Aided Design*, pages 314–319, Nov. 2001.

[14] J. Hammes et al. Cameron: High-level language compilation for reconfigurable systems. In *Proc. of the 8th International Conference on Parallel Architectures and Compilation Techniques*, pages 236–244, Oct. 1999.

[15] D. Herrmann and R. Ernst. Improved interconnect sharing by identity operation insertion. pages 489–493, 1999.

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

[17] 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.

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

[19] 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.

[20] 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.

[21] R. Schreiber et al. PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators. *Journal of VLSI Signal Processing*, 31(2):127–142, 2002.

[22] G. Snider, B. Shackleford, and R. Carter. Attacking the semantic gap between application programming languages and configurable hardware. In *Proc. of the 9th ACM Symposium on Field Programmable Gate Arrays*, pages 115–124, Feb. 2001.

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

[24] 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.

[25] 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.

[26] X. Zhu and B. Lin. Hardware compilation for FPGA-based configurable computing machines. In *Proc. of the 36th Design Automation Conference*, pages 697–702, June 1999.