Cost Sensitive Modulo Scheduling in a Loop Accelerator Synthesis System

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

Scheduling algorithms used in compilers traditionally focus on goals such as reducing schedule length and register pressure or producing compact code. In the context of a hardware synthesis system where the schedule is used to determine various components of the hardware, including datapath, storage, and interconnect, the goals of a scheduler change drastically. In addition to achieving the traditional goals, the scheduler must proactively make decisions to ensure efficient hardware is produced. This paper proposes two exact solutions for cost sensitive modulo scheduling, one based on an integer linear programming formulation and another based on branch-and-bound search. To achieve reasonable compilation times, decomposition techniques to break down the complex scheduling problem into phase ordered sub-problems are proposed. The decomposition techniques work either by partitioning the dataflow graph into smaller subgraphs and optimally scheduling the subgraphs, or by splitting the scheduling problem into two phases, time slot and resource assignment. The effectiveness of cost sensitive modulo scheduling in minimizing the costs of function units, register structures, and interconnection wires are evaluated within a fully automatic synthesis system for loop accelerators. The cost sensitive modulo scheduler increases the efficiency of the resulting hardware significantly compared to both traditional cost unaware and greedy cost aware modulo schedulers.

1. INTRODUCTION

The markets for cellular phones, portable digital assistants, digital cameras, and other special-purpose devices continue to grow explosively. The embedded computing systems that go into these devices must meet the demands of higher performance and greater energy efficiency to support new functionality, added capabilities, more flexibility, and higher bandwidth communication. To achieve these challenging goals, application-specific hardware in the form of loop accelerators is commonly used to execute the compute-intensive portions of applications that would run too slowly if implemented in software on a programmable processor. Low-cost, high-performance, systematic verification, and short time-to-market are all critical objectives for designing these accelerators. Automatic synthesis technology to build loop accelerators from high-level specifications is critical to achieving these objectives.

A key challenge with automatic synthesis is creating efficient designs. Efficiency can be defined along many axes, including performance, cost, and energy. For this work, the focus is on cost efficiency, thus the objective is to design the lowest cost accelerator that meets a specified performance level. Cost-efficient accelerators are synthesized by optimizing the design in a number of ways. First, hardware structures are sized just large enough to meet the precision requirements of the application. Second, storage structures (memories, registers, etc.) are given just enough entries to meet the worst-case requirements of the application. Third, hardware can be shared by time multiplexing hardware components when either the hardware is required under disjoint conditions or the performance of dedicated hardware is not necessary. In addition to the hardware components, interconnect can also be optimized using the same strategies. A manual designer is typically proactive in organizing the design to maximize the savings of these general approaches and balance tradeoffs between component and interconnect cost.

This work examines the construction of a loop accelerator synthesis system. The proposed system utilizes a compiler-directed approach for designing accelerators that was derived from the PICO-NPA (Program-In Chip-Out Non-Programmable Accelerator) system [28]. The inputs to the system are a target loop nest expressed in C, the desired throughput, and the available memory bandwidth. Synthesis is divided into three steps. First, a simple, single-cluster VLIW processor is designed to meet the throughput requirements of the application. The simple processor consists of a set of arbitrary function units, connected to a centralized register file with unlimited entries and an unbounded memory. It provides an abstract target to which the compiler can efficiently map algorithms. Next, modulo scheduling is performed to map the application onto the simple processor [27]. Finally, a stylized loop accelerator is synthesized from the resulting schedule.

The critical portion of the synthesis system is the modulo scheduler. A traditional modulo scheduler attempts to map a loop onto a fixed hardware configuration, optimizing the throughput, number of pipeline stages, and possibly the lifetimes of registers. In our system, the resulting schedule of operations is used to determine the complete architecture of the accelerator, including the control path, computation elements, storage structures, and interconnect. Thus, the scheduling objectives are completely changed. The scheduler must make binding decisions that lead to the most efficient design. Hence, *cost sensitive modulo scheduling* is proposed.

The objective of cost sensitive modulo scheduling is to create a schedule that not only achieves a specified throughput, but also yields the lowest cost accelerator design. To accomplish this objective, the accelerator design is modeled during scheduling, so the impact of binding decisions on cost can be assessed. Our first approach to this problem utilized a greedy strategy, wherein at each scheduling step, the alternative that produced the least cost increase to the current design was made. The greedy approach was generally better than the baseline cost insensitive scheduler, but not by a large amount. The scheduler got trapped in too many local minima and the overall quality did not improve much.

The central problem is that each portion of the accelerator architecture is not the result of an individual scheduling decision, but rather is determined by many inter-related scheduling decisions. Each decision for a single operation has cost implications on earlier and later decisions. Thus, a greedy approach inherently does not make sense as the cost saved by making one decision is often unrelated to the cost of the entire design. As a result, we decided to focus on two scheduling methods that provide exact solutions: branch-and-bound and integer linear programming. Our approach is to develop cost sensitive formulations of both methods.

As with most exact formulations, these methods break down for moderate to large problem sizes as the run-time and memory usage of these methods explode. Thus, the scheduling problem is decomposed into a set of more manageable subproblems, where each subproblem is solved in a phase-ordered manner. We utilize three techniques to break down the problem: graph partitioning, space-time decomposition, and time-space decomposition. Graph partitioning divides loop bodies into smaller subgraphs, optimally scheduling the subgraphs, while space-time and time-space decomposition split the scheduling process into two separate phases, time slot and resource assignment. These methods sacrifice optimality of the schedule and thus of the cost of the accelerator, but enable realistic problems to be solved in a reasonable amount of time, while achieving substantial cost savings.

1.1 Related Work

Datapath synthesis from high level descriptions has been researched widely by many researchers. Cathedral III represents a complete synthesis system developed at IMEC and illustrates one comprehensive approach to high-level synthesis [22]. It uses an applicative language for program specification and designs customized datapaths for signal processing applications from this specification. The Sehwa system automatically designs processing pipelines from behavioral specifications [25]. The PICO system synthesizes C loop nests into a synchronous array of customized processor datapaths [28]. The above systems produce standard cell based designs. 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 [10] that was subsequently productized as the NAPA system [11]. Other automatic compiler systems for FPGA-based platforms include Garp [5], PRISM [31], Cameron [14], Match [13], DEFACTO [4], and a SUIF-based system [2].

Cost sensitive scheduling in the context of data path synthesis has been studied for many years. Force-directed scheduling integrates resource allocation and scheduling into a common synthesis algorithm to minimize overall cost of synthesized datapaths [26]. Tradeoffs in allocating either low latency and expensive or high latency and inexpensive resources have been considered within an integrated scheduling and resource allocation algorithm [3]. [23] proposes a polynomial time scheduling algorithm based on heuristics that produces near optimal results. [17] presents an integer programming formulation for the scheduling problem in data path synthesis. Generation of more efficient designs by sharing hardware across basic blocks was recently proposed [21]. All of the above work handle only acyclic code regions. The optimization criteria usually is achieving shortest schedule length, or given a schedule length, achieving the least cost of data path. The focus of our work is cyclic scheduling. Though the components of the cost are the same, the optimization strategy is different because of the way in which function units are utilized in a cyclic schedule.

Heuristics that work as a preprocessing step to scheduling and try to minimize cost of the resulting hardware have also been studied. Clique-based partitioning algorithms were developed in the FACET project to jointly minimize function unit and inter-function unit communication costs [30]. Within the PICO system, width clustering is used to bind operations of narrow bitwidth to common resources to reduce datapath cost [20]. Assignment of scheduled operations to resources with the goal of increasing interconnect sharing has been proposed [24]. The advantage of preprocessing heuristics is that they are fast and usually achieve good results when used in conjunction with a traditional scheduling algorithm. Our work intertwines the cost minimization into the scheduling algorithm to achieve greater cost savings.

In the compiler domain, software pipelining is a technique to exploit instruction-level parallelism by overlapping the execution of successive loop iterations. Modulo scheduling is a class of software pipelining algorithms that achieve high quality solutions and have been implemented in production compilers [27]. A number of extensions to modulo scheduling have been proposed to increase the quality of the solution, including reducing register requirements [15, 9, 18] and code size [19]. Reducing register requirements is most closely related to accelerator cost reduction. Swing modulo scheduling changes the core modulo scheduler to reduce register requirements by considering operations in different orders and changing how time slots are chosen [18]. Conversely, stage scheduling is a post-processing to shift the pipeline stage of instructions to reduce register requirements [9]. While the application of these techniques can reduce the cost of loop accelerators, the affect is limited as traditional compiler-based measures, such as register lifetimes, do not reflect the structure of a loop accelerator. For instance, a long lifetime may be free in an accelerator if it is scheduled to share a register with a similar lifetime. Hardware sharing and all aspects of cost must be considered to create efficient loop accelerators.

Many techniques for optimal modulo scheduling have been proposed in the literature. [6] proposes and efficient integer programming formulation for optimal modulo scheduling. [1] proposes an enumeration based approach for optimal modulo scheduling. Both of these techniques focus primarily on achieving a valid schedule. Minimizing register requirements has been the main optimization criteria for many of the works published on optimal modulo scheduling. [12], [7], and [8] formulate the modulo scheduling with minimum register requirements as an integer linear programming problem. Our work uses the basic ILP formulation from [6] and builds upon it significantly by adding variables and constraints to represent the cost of hardware and uses the hardware cost as the objective function.

1.2 Contributions of this Work

The contributions of this work are two-fold:

- We present the formulation of two exact methods for cost sensitive modulo scheduling: branch-and-bound and integer linear programming. Each method can be applied to optimize for area, interconnect, or a simple combination of both. We compare the effectiveness of these methods to traditional cost insensitive and greedy cost sensitive modulo schedulers. criteria.
- To address the issue of problem size explosion common to exact scheduling methods, three methods for decomposing scheduling algorithms into phased solutions of simpler subproblems are utilized. They consist of graph partitioning, time-space decomposition, and space-time decomposition. The implementation details of each are presented along with analysis of the performance tradeoffs.

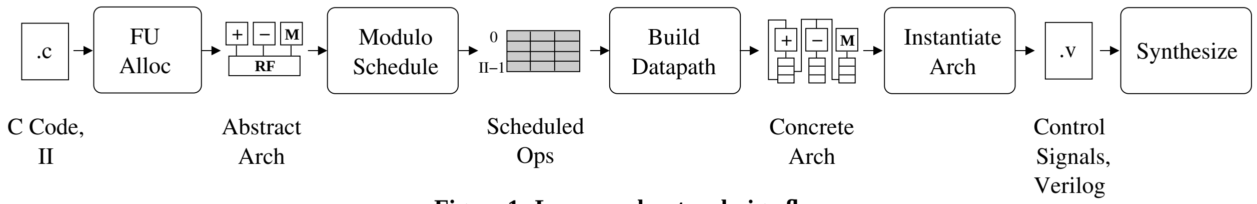


Figure 1: Loop accelerator design flow.

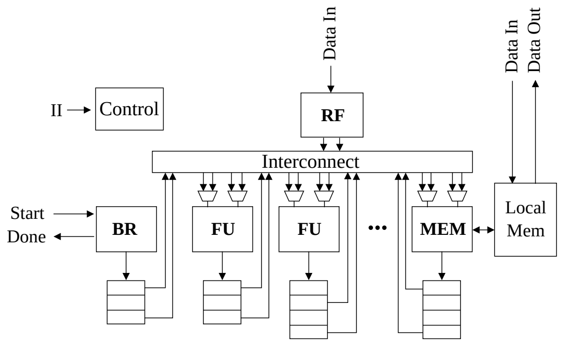


Figure 2: Loop accelerator schema.

2. SYNTHESIS SYSTEM OVERVIEW

Cost sensitive modulo scheduling is implemented as part of an automatic synthesis system to create stylized loop accelerators. In this section, we first describe the hardware schema of the loop accelerators and then explain each step of the system (Figure 1) with an example from `sobel`, an edge detection algorithm.

2.1 Hardware Schema

Figure 2 shows the hardware schema used in this system. It is designed to exploit the high degree of parallelism available in modulo scheduled loops with a large number of function units (FUs). Each FU writes to a dedicated shift register where entries in registers move down at every cycle. 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 multiplexor (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 registers.

Literals and static live-in register values cannot be stored in the shift register files. Therefore, these values are supplied by a central register file which is connected to the inputs of FUs that require literal or live-in operands. FUs that access memory are connected to a local memory structure such as a scratchpad, cache, or stream buffer. The loop accelerator begins execution when a *start* signal is asserted by the host processor. When the loop execution is complete, the branch FU asserts a *done* signal to the host processor.

2.2 System Flow

The overall flow of the synthesis system is presented in Figure 1. Each step of the flow is described in this section with an example from the sobel edge detection algorithm.

FU allocation. This step takes the inputs of the system and creates an abstract VLIW architecture that represents a high-level view of the accelerator's functionality. 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 (expressed as the initiation interval of the loop or II [27]), 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 throughput constraint. In this phase, all FUs are assumed to be full width for cost purposes. (Bitwidth specialization is performed after the cost sensitive scheduling, when operations have been assigned to specific FUs.) 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, such as both ADD and ADDSUB units being available. 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 supporting all of the operations. Figure 3(b) shows the result of FU allocation for `sobel` with II=4. There are 22 ADD and 2 SUB operations in the loop, which are covered by the 5 ADD and 1 ADDSUB units.

Modulo Scheduling. The loop is modulo scheduled to the abstract architecture created in the previous step. A cost-sensitive modulo scheduler, to be described in Sections 3 and 4, assigns operations to the resources and timeslots in the abstract architecture. At the completion of this phase, all of the loop operations are bound to resources and time, and the producer-consumer relationships between FUs have been determined. Figure 3(c) shows some operations from the modulo schedule for `sobel`, with edges indicating communication between operations. The number associated with each operation indicates its width; the width of each FU is set to the width of the largest operation assigned to it.

Datapath Construction. 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 a shift register entry at the output of a producing FU to the input of a consuming FU. The register entry that should be connected is determined from the difference in execution time between the producer and consumer, since register entries move down at every cycle. The bitwidths of FUs and register files are determined by the maximum bitwidth of operations that are mapped to the FU or contained in the register. The depth of a register file is set to the longest lifetime of the values produced by the corresponding FU. Figure 3(d) shows the shift register files and connections resulting from the scheduled operations in Figure 3(c).

Architecture Instantiation. Lastly, the architecture created in the previous step is lowered into a Verilog realization of the accelerator. Each module in the datapath is translated into a set of primitive modules that have pre-defined behavioral Verilog descriptions. To reduce global wiring of control signals, we employ a distributed hierarchical control scheme that consists of three levels of control logic: FU control activates the appropriate primitive FU with the proper functionality and sets any internal MUX selects; cluster control converts the II value to generate high-level FU opcodes and sets the input MUXes select signals; and, processor control generates the II counter value. A subset of the final lowered datapath for `sobel` is presented in Figure 3(e). Input MUXes are

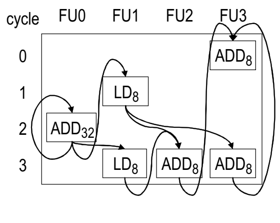
```
for (i = 0; i < N1; i++) {
  for (j = 0; j < N2; j++) {
    t00 = x[i  ][j  ];
    t01 = x[i  ][j+1];
    t02 = x[i  ][j+2];
    t10 = x[i+1][j  ];
    t12 = x[i+1][j+2];
    t20 = x[i+2][j  ];
    t21 = x[i+2][j+1];
    t22 = x[i+2][j+2];

    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;
  }
}
```

(a)

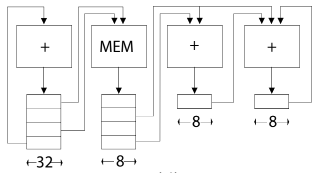
II = 4

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

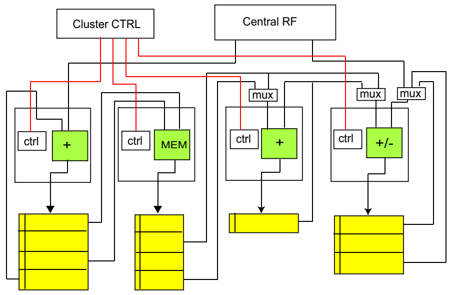
(b)



(c)



(d)



(e)

Figure 3: An example loop accelerator design. (a) `sobel` source code, (b) result of FU allocation with II = 4, (c) a portion of the `sobel` modulo scheduled loop, (d) datapath derived from the modulo schedule shown in (c), (e) lowered datapath.

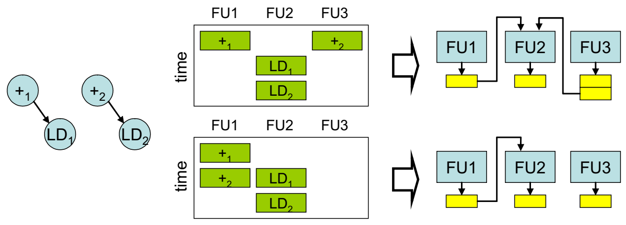


Figure 4: Effect of schedule on wire cost.

added when multiple wires share the same FU input port, and the control path is generated to direct the MUXes and FUs.

3. SCHEDULING TECHNIQUES

Cost sensitive modulo scheduling focuses on reducing the cost of three components of the hardware: FUs, register storage, and interconnect wires. These components were found to dominate the hardware cost of loop accelerators; other components such as multiplexers and control signals are less significant and are not specifically targeted for cost reduction in this work. By reducing the sizes of FUs and shift registers required to support execution of a given loop, the resulting hardware implementation will achieve the same throughput with fewer logic gates and less power. In addition, with high numbers of FUs and registers to support loop level parallelism, the interconnection network feeding values from registers to FU inputs can grow very large. Decreasing the number of wires required to support these data transfers reduces chip area from the wires themselves as well as from simplifying the placement and routing of other structures in the hardware layout.

FU and storage cost can be reduced by scheduling operations cognizant of their resource and communication requirements, such as bitwidth and register lifetimes; by maximizing hardware reuse, the total amount of hardware is reduced. Wire cost can be reduced by maximizing reuse of the same wires by different producer and consumer operations. Wires are reused if producers and consumers are scheduled on the same respective FUs, and the consumers read data from the same shift register entry (i.e., the time differences between producers and consumers are identical). In Figure 4, assume the two pairs of operations to be scheduled are 32 bits wide. An interconnect-unaware modulo scheduler might produce the upper schedule, which requires 64 wires, while the lower schedule would have required only 32.

The remainder of this section describes approaches for achieving these goals, assigning operations to FUs and time slots such that the cost of the hardware needed to support their execution is minimized.

3.1 Greedy Scheduling

The baseline (naïve) scheduler used in this work is the iterative modulo scheduler described in [27], with a stage scheduling postpass [9]. This scheduler arbitrarily selects an available scheduling alternative for each operation in order to meet a given II, and does not consider hardware cost. The stage scheduling postpass reduces register lifetimes, which may reduce hardware cost, but this is done without cognizance of the hardware.

A straightforward way to make the scheduler cost-aware is to augment the naïve modulo scheduler with a hardware cost model and a greedy heuristic to minimize cost. The cost aware scheduling framework is shown in Figure 5. The main component of this framework is the hardware cost modeler, explained in more detail in Section 3.2.1. 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. (This estimate is explained in more detail in Section 3.2.2.)

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 the best alternative and schedules the operation on that particular FU and time slot. This is done for all operations in priority order, backtracking as needed. After the completion of greedy scheduling, the stage scheduling postpass is performed to decrease register lifetimes.

3.2 Branch-and-Bound Solution

A second method of obtaining a modulo schedule that minimizes hardware cost is to utilize an optimal branch-and-bound (BNB) so-

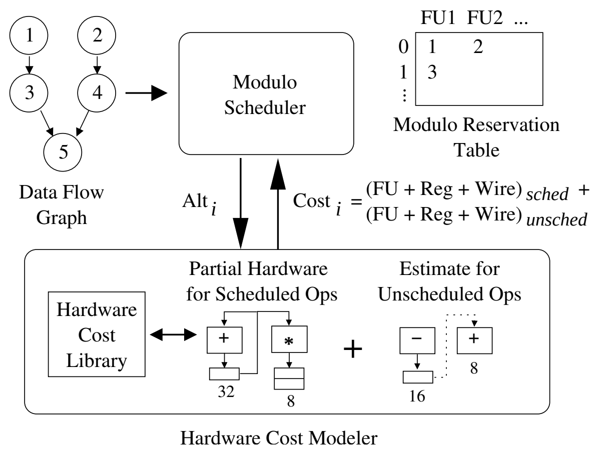

Data Flow Graph
Modulo Scheduler
FU1 FU2 ...
Modulo Reservation Table
Alt $_i$
Cost $_i$ = (FU + Reg + Wire) $_{sched}$ + (FU + Reg + Wire) $_{unsched}$
Partial Hardware for Scheduled Ops
Estimate for Unscheduled Ops
Hardware Cost Library
32
8
16
8
Hardware Cost Modeler

Figure 5: Cost sensitive scheduling framework used for greedy and branch-and-bound schedulers.

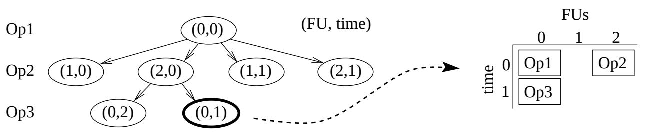


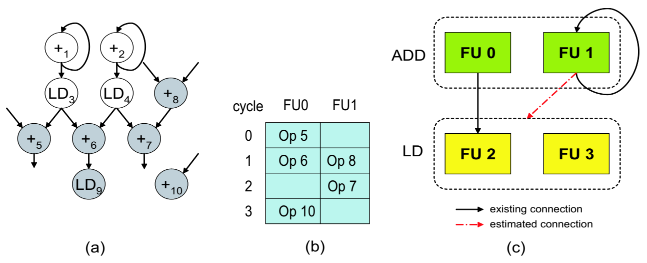
Figure 6: Branch-and-bound search. The highlighted state corresponds to the partial schedule shown.

lution. The goal is to search all possible schedules in order to find one that has the lowest hardware cost. The search is performed by scheduling each operation at all of its valid alternatives (FUs × time slots). (In a modulo schedule, each operation can be scheduled in at most II different time slots.) Operations are considered in order of least to most available alternatives; the order does not affect the algorithm's optimality, only its runtime. The search space can be represented by a tree as shown in Figure 6. Each node represents a partial schedule, or a state in which some operations have been assigned FUs and time slots. The children of a node are formed by scheduling the next operation at all of its valid alternatives, subject to resource and dependence constraints. Leaf nodes in the tree therefore represent full schedules, and the goal is to locate a leaf node whose schedule requires the minimum amount of hardware.

3.2.1 Hardware Modeling

The BNB scheduler uses a hardware model to estimate the cost of a machine supporting a given partial schedule. Three aspects of hardware cost are modeled: FUs, register storage, and interconnect wires. Function unit cost is determined by the capabilities of the FU as well as its width. In the loop accelerator synthesis system, FU capabilities are determined during the FU allocation phase described in Section 2, prior to scheduling. Therefore, the scheduler has influence only on the width of the FU – if only narrow bitwidth operations are scheduled on an FU, then its cost can be reduced. The FU cost for a given partial schedule can therefore be determined as a function of the maximum bitwidth operation scheduled on each FU.

The register storage cost is determined similarly. Each shift register must be wide enough to accommodate the maximum bitwidth operation scheduled on the corresponding FU, and deep enough to hold the value with the longest lifetime. Also, interconnect wires must connect specific registers with FU input ports. Given a partial schedule, the known producer-consumer relationships between operations is used to obtain the widths and depths of the shift registers, as well as the number of interconnect wires required.



cycle	FU0	FU1
0	Op 5	
1	Op 6	Op 8
2		Op 7
3	Op 10	

Figure 7: Wire estimation example: (a) DFG, (b) partial schedule, (c) connection diagram

The BNB algorithm requires a single metric to determine whether a given schedule is better or worse than previously explored schedules. Therefore, when the objective is to decrease overall hardware cost, the combined metric used is the sum of wires, storage bits, and FU cost. (The units of FU cost are scaled such that they are equivalent to storage bits in terms of the number of logic gates required for implementation.) The wire, storage, and FU metrics may also be used alone, for example, to obtain a schedule with the objective of minimizing only storage cost.

3.2.2 Hardware Cost Estimation

An effective bound function is a crucial element in any BNB algorithm in order to prune, as early as possible, search paths that will not yield optimal results. The search is bounded using an estimate of the hardware needed to support operations that have not yet been scheduled. Thus, for any partial schedule, when the cost of hardware required by scheduled operations plus hardware estimated for unscheduled operations exceeds the best solution found so far, that search path is pruned. As long as the estimate is conservative (i.e., never overestimates the actual hardware cost), optimality is preserved as no search paths will be erroneously pruned. Additionally, the more accurate the estimate, the more likely a wrong path will be pruned earlier, thus decreasing the run time of the search.

Wire estimation. For the wire estimation, FUs are placed into groups based on their functionality. An FU group is the basic unit for estimation, and estimated connections are made between FU groups. For a given pair of FU groups, we collect all compatible edges[1] whose producer or consumer ops are unscheduled. Then, we determine the minimum number of additional connections required to support those unscheduled edges based on the number of available slots on the FUs (each FU has II slots). We optimistically assume that empty slots in the FUs can be occupied by any compatible unscheduled producer, ignoring scheduling constraints. This assumption guarantees that the estimation is a lower bound for the wire cost. It is assumed that existing wires in an FU group with n free slots can be reused by n unscheduled operations with compatible edges. When there are more than n such operations, new estimated connections are made to support the remaining operations.

Figure 7 shows how the estimation is performed for the ADD FU group. The processor consists of four FUs, two in the ADD FU group (FU0, FU1) and two in the LD FU group (FU2, FU3). Assume the shaded operations are already scheduled and wires are being estimated for the unshaded operations. There are two types of edges originating from ADD operations: ADD→ADD and ADD→LD. As there is already a connection from FU 0 to FU 2 and FU 0 has one available slot, one of the two ADD→LD edges

[1]Multiple dataflow edges whose producer and consumer operations can execute on corresponding FU groups.

can potentially be scheduled without generating additional connections. This will make FU 0 fully occupied and the producer of the second edge must be placed on FU 1. Therefore, a new estimated connection between FU 1 and the LD FU group is created. Another estimation is performed independently for the ADD→ADD edges. Here, both can potentially be scheduled by placing the producers on FU 1, as it has two available slots. Thus, the ADD→ADD edges will not require any additional connections. As a result, the wire estimation for the ADD FU group is one.

Storage estimation. Estimating the incremental storage requirements for unscheduled operations is performed using an analogous method. First, the overall storage requirements for the unscheduled operations is determined; then, based on the number of available execution slots in the FUs and the existing register storage, the number of bits of new storage needed to support the unscheduled operations is estimated.

For each unscheduled operation, an estimate of the number of register bits needed to hold its result is obtained. This value depends on the width and depth of the output register; the width is simply the bitwidth of the operation, while the depth can be estimated from the estart/lstart[2] times of the operation and its consumers. More specifically, for operation *op* with consumers *cons*:

$$depth = \left(\max_{c \in cons} estart_c\right) - lstart_{op} - latency_{op} \qquad (1)$$

Once register requirements are approximated for the unscheduled operations, it is optimistically assumed that existing shift registers at the outputs of compatible FUs with available execution slots can be reused to satisfy these requirements. Any required register bits that cannot be satisfied by existing registers become part of the incremental storage estimation. Similarly to the wire estimation, this storage estimation does not take dependence constraints into consideration and is therefore conservative.

Function unit estimation. FU cost estimation is somewhat simpler than wire or storage estimation, since FU capabilities are fixed prior to scheduling and only the FU width varies depending on the schedule. First, unscheduled operations are grouped by type and their maximum bitwidth is determined. Next, existing FUs with free slots are used to satisfy these FU requirements. Finally, the additional cost of FUs needed to support the remaining operations (either by widening existing FUs or creating new FUs) is calculated.

For a given partial schedule, once the wire, storage, and FU costs have been estimated for the unscheduled operations, the search may be pruned. Once again, a single metric is needed for the hardware cost estimate, and this is obtained by the weighted sum of the wire, storage, and FU cost metrics. Note that these hardware estimations are performed at every step of the BNB search. Therefore, they are implemented in a computationally efficient way, using incremental updates to internal data structures in order to minimize their impact on the execution time of the search. Note also that it is worthwhile to spend some computation time obtaining an accurate estimate if it allows the search paths to be pruned earlier, since the number of states eliminated by pruning a node is exponential in the height of the node.

3.3 Integer Linear Programming Formulation

The third approach to the problem is an integer linear programming (ILP) formulation for achieving modulo schedules optimal with respect to the cost of hardware generated from the schedule. The basic structure of the formulation is identical to the one proposed in [12, 6]. The basic formulation described in these works do not perform FU assignment, but only ensure that a valid assignment is possible. FU assignment is crucial in determining cost of hardware derived from the schedule. In the formulation described in this section, additional variables and constraints to represent FU assignment for operations is added to the basic formulation. An objective function to represent hardware cost is derived from these variables and constraints.

3.3.1 Basic Formulation

The body of the loop under consideration is represented by a graph $G = \{V, E\}$, where V represents the set of operations in the loop body and E represents data dependence edges between operations. Each dependence edge has an associated latency $l_{i,j}$ which specifies the latency of the producer i, and a distance $d_{i,j}$, which specifies the difference in iterations between when the value is produced by i and when the value is consumed by j.

Consider a loop with $|V| = N$ operations. Let II be the initiation interval. The schedule for this loop is represented by $II \times N$ binary variables $X_{i,s}$. Operation $i \in \{0, N-1\}$ is scheduled in slot s, $0 \le s \le II-1$, if $X_{i,s} = 1$. The following constraint enforces a unique slot for every operation i.

$$\sum_{s=0}^{II-1} X_{i,s} = 1 \qquad \forall i \in \{0, N-1\} \qquad (2)$$

N integer variables $k_i, i \in \{0, N-1\}$ are introduced to represent the stage in which each operation is placed. $X_{i,s}$ and k_i uniquely identify the cycle in which an operation i is scheduled. In fact, the schedule time of an operation i is given by

$$t_i = \sum_{s=1}^{II-1} s \times X_{i,s} + II \times k_i \qquad (3)$$

Note that t_i is used as a shorthand to represent the schedule time of an operation i. In a real implementation, there is no need to introduce a new variable to represent the schedule time. Given the t_i's for all operations, the data dependences between operations can be enforced with the following set of constraints.

$$t_j + d_{i,j} \times II - t_i \ge l_{i,j} \qquad \forall (i,j) \in E \qquad (4)$$

The schedule times should satisfy the resource constraints, i.e., the number of operations scheduled in each slot should not exceed the available number of FUs for each FU type. Suppose I_f are the set of operations that require a FU of type f and M_f are the total number of FUs of type f available. Then, the following constraint enforces the resource constraints.

$$\sum_{i \in I_f} X_{i,s} \le M_f \qquad s \in \{0, II-1\} \qquad (5)$$

Note that the above constraint only ensures a valid FU assignment and does not actually perform the assignment.

3.3.2 Function Unit Assignment

The FU assignment for operations is represented by a set of binary variables $R_{i,j}$, $i \in \{0, N-1\}$, $j \in \{0, M_f - 1\}$, i.e., there are M_f binary variables for every op i, where M_f is the number of compatible FUs to which i can be assigned. The following constraint enforces a unique assignment.

$$\sum_{j=0}^{M_f-1} R_{i,j} = 1 \qquad \forall i \in \{0, N-1\} \qquad (6)$$

[2] $estart_{op}$: earliest start time of *op* ignoring resource constraints. $lstart_{op}$: latest start time of *op* without delaying exit operations.

The number of operations assigned to a particular FU cannot exceed II. The following constraint enforces this.

$$\sum_{i \in I_j} R_{i,j} \leq II \qquad i \in I_j \text{ can execute on } j \tag{7}$$

Even with the above constraint, an FU can be assigned to two operations in the same cycle. To prevent this from happening, the following constraint has to be enforced for every FU.

$$\sum_{i \in I_j} R_{i,j} \times X_{i,s} \leq 1,$$
$$\forall s \in \{0, II-1\}\ and\ i \in I_j \text{ can execute on FU } j \tag{8}$$

The above equation is a sum of products of two binary variables, and is non-linear. It can be easily linearized as follows. For every $R_{i,j}$ and $X_{i,s}$ appearing in the above set of equations, an auxiliary binary variable $Z_{i,j,s}$ is introduced and following set constraints are enforced on $Z_{i,j,s}$.

$$\begin{aligned} -R_{i,j} + Z_{i,j,s} &\leq 0 \\ -X_{i,s} + Z_{i,j,s} &\leq 0 \\ R_{i,j} + X_{i,s} - Z_{i,j,s} &\leq 1 \end{aligned} \tag{9}$$

Now the product terms in Equation 8 can be replaced with the corresponding $Z_{i,j,s}$'s. Solving equations 2 through 9 would yield a valid schedule and FU assignment for operations in a loop.

3.3.3 Cost Minimization

As described in Section 2, the hardware schema is a set of FUs writing to independent shift registers. The cost of the hardware includes cost of the FUs and cost of the shift registers and cost of wires used to connect shift registers to the input of FUs. In this section, we describe modeling of costs of FU and shift registers only. Modeling wire cost is left out due to space considerations.

Function unit cost. The cost of the FU depends on the set of operations assigned to it. For example, if 8-bit and 16-bit add operations are assigned to an add FU, then the cost of the add FU is the cost of a 16-bit adder. Suppose H_i is the cost of a FU required to execute operation i only. H_i is a constant and is a (possibly non-linear) function of the bitwidth of operation i. Now, the cost of a FU j will be at least H_i, if i is assigned to j. Since we have binary variables to represent the fact that operation i is assigned to FU j, the above fact be represented as follows.

$$C_j \geq R_{i,j} \times H_i$$
$$i \text{ can execute on FU } j\ and\ C_j \text{ is the cost of FU } j \tag{10}$$

The above constraint is introduced into the integer program for every operation i that can be assigned to FU j. Thus, C_j automatically gets set to the maximum cost of an FU that can execute any set of operations assigned to it. The total cost of FUs in the hardware can be calculated as follows.

$$\sum_{j \in FUs} C_j \tag{11}$$

Storage cost. As described in Section 2, the FUs write their output to the head of a shift register which shifts the values down every cycle. The shift register should have enough entries to hold the values until the consumer FU reads it in a later cycle. Consider an operation i_1 feeding another operation i_2. From Equation 3 we know that t_{i_1} and t_{i_2} are the schedule times of i_1 and i_2 respectively. The value produced by i_1 is read by i_2 after $t_{i_2} - t_{i_1} + II \times d_{i_1,i_2} - l_{i_1,i_2} + 1$ cycles. i_1 could have many consumers and the latest time a value produced by i_1 is live is the maximum of $t_{i_2} - t_{i_1} + II \times d_{i_1,i_2} - l_{i_1,i_2} + 1$ with respect to some consumer. A integer variable LT_i is introduced for every producer i operation in the loop body to indicate the maximum lifetime (measured in number of cycles) of the value produced by that operation.

$$LT_i \geq t_{i'} - t_i + II \times d_{i,i'} - l_{i,i'} + 1 \quad (i,i') \in E \tag{12}$$

Note that the lifetime indicates the lifetime in actual number of cycles. This is significantly different from the lifetime measure used in [7, 12] which is just the maximum number of values produced by an operation live at any instant. The maximum lifetimes of values produced by operations is used to calculate the depth D_j of the shift register associated with an FU j. A shift register should hold live values from all operations assigned to it. Therefore, D_j is the maximum of lifetimes of any operation assigned to it. This can be represented as follows.

$$D_j \geq R_{i,j} \times LT_i \qquad \forall i \text{ assigned to } j \tag{13}$$

The above equation is a product of a binary variable and an integer variable, and is non-linear. However, it can be linearized using an auxiliary variable TD_j as shown below.

$$\begin{aligned} TD_j &\geq 0 \\ TD_j &\leq P \times R_{i,j} \\ TD_j &\leq LT_i \\ TD_j &\geq LT_i - (1 - R_{i,j}) \times P \\ D_j &\geq TD_j \end{aligned} \tag{14}$$

where P is a suitably large constant. Note that TD_j is 0 when $R_{i,j}$ is 0 and is equal to LT_i when $R_{i,j}$ is 1. D_j thus gets the maximum of LT_i among all operation i assigned to FU j.

The cost of the shift register of an FU also depends on the bitwidth of the operations assigned to the FU. In fact, the width of the shift register has to be the maximum of the bitwidths of operations assigned to the FU. The width W_j of the shift register associated with FU j is calculated as follows.

$$W_j \geq R_{i,j} \times BW_i \qquad \forall\ i \text{ assigned to } j \tag{15}$$

where BW_i is a constant, indicating the bitwidth of the values produced by operation i. From D_j and W_j, the cost S_j of the shift register associated with FU j can be calculated as follows.

$$S_j = W_j \times D_j \tag{16}$$

The above equation is non-linear. However, it can be linearized using the observation that W_j can take only a small set of discrete values. Suppose W_j can take values w_1, w_2, ... w_k. Then, W_j can be represented as shown below.

$$W_j = \sum_{n=1}^{k} w_n \times b_{j,w_n}, \qquad \sum_{n=1}^{k} b_{j,w_n} = 1 \tag{17}$$

where b_{j,w_1}, b_{j,w_2}, ... b_{j,w_k} are binary variables. Now S_j can be expressed in linear form as follows.

$$\begin{aligned} S_j &\leq w_{max} \times D_j \\ S_j &\geq w_n \times D_j - (1 - b_{j,w_n}) \times Q, \quad \forall n \in \{1, n\} \end{aligned} \tag{18}$$

where w_{max} is the maximum among w_1, w_2, ... w_k and Q is a suitable large constant.

The objective function for minimizing the cost of data-path of the hardware can now be calculated from equations 11 and 18.

$$\sum_{j \in FUs} C_j + S_j \tag{19}$$

The overall ILP formulation for cost sensitive modulo scheduling can be stated as “minimize Equation 19, subject to constraints expressed in Equations 2 through 18”.

4. DECOMPOSITION METHODS

It is necessary to decompose the modulo scheduling problem described in the previous section because the number of possible schedules is too large for realistic loops. There are multiple ways in which the problem can be decomposed. One approach is to partition the dataflow graph into sets of operations and then schedule the sets one by one. Another approach is to perform scheduling in phases. In this case, all operations are considered at once, but only resource assignment is performed in the first phase, and time assignment is performed in the second phase. Alternatively, the two phases can be performed in reverse order.

4.1 Operation Partitioning

One natural way of simplifying the scheduling problem is to partition the operations into multiple disjoint sets. The size of each set is bounded (generally to 10-15 operations), and thus the space of possible schedules for the operations in a set can be reasonably explored using the branch-and-bound or ILP techniques described in Section 3.

The scheduler considers sets of operations in sequence. Within each set, an optimal assignment of operations to resources and time is obtained which minimizes the cost of the additional hardware required by this set. Once operations from a set are scheduled, their resource and time slot assignments are fixed, and subsequent sets will take these assignments into account when they are scheduled. Thus, for each set of operations, the scheduler attempts to utilize two forms of hardware sharing to minimize cost: intra-set sharing, where operations within a set reuse new hardware, and inter-set sharing, where operations reuse existing hardware from previously scheduled sets.

The partitioning scheduler therefore obtains an optimal solution for each set, and the combination of these solutions forms the final global schedule. This decomposition method loses some global optimality because only operations within the same set are considered together, and scheduling decisions made in earlier sets cannot be changed when scheduling later sets. However, in general this method is effective in producing low-cost schedules as both resource and time assignments are made jointly, and the decisions account for previously scheduled sets. An elegant tradeoff can be achieved between global optimality and running time of the scheduler. Larger sets are likely to give solutions closer to the globally optimal solution at the cost of increased search time. Smaller sets can be quickly searched to find locally optimal solutions.

Two issues have to be addressed in this scheduling scheme. First, a suitable partitioning method has to be devised. Second, a backtracking strategy has to be designed to ensure successful completion of the scheduler.

4.1.1 Partitioning Method

A simple way to partition the data flow graph is to consider the height based priority order of operations used in a typical scheduler, and place every n operations into a set (where n is the desired set size). Since the height based priority minimizes the instances where a consumer is scheduled before its producer operation, this partitioning method minimizes backtracking and ensures quick convergence to a schedule. A more sophisticated graph partitioning method could also be employed to form partitions. However, unlike traditional graph partitioning, the goal of partitioning the dataflow graph of the loop body is not to achieve min-cut of the edges. This is because we are not considering a traditional performance metric like schedule length. Instead, a good partition is one which exposes as many hardware sharing opportunities as possible within a set. Since exhaustive search is performed on each partition, all the sharing opportunities will be exploited and the combined global solution is improved.

A simple heuristic is used to form partitions with high hardware sharing opportunities. First, a similarity metric is calculated between every pair of operations in the dataflow graph. Then the operations are partitioned into sets by taking operation pairs in order of descending similarity and placing every n operations into a set.

The similarity metric has two components, one based on potential for sharing interconnect wires and one based on potential for sharing register storage. The wire similarity metric is a count of the number of wires (in bits) that can potentially be shared between two operations and their producers/consumers, determined by counting compatible edges. To estimate the storage similarity metric, register requirements are first estimated for each operation using the method from Section 3.2.2. Then, the metric is calculated as the number of bits of register storage the two operations have in common. This figure accounts for the dimensions of the register files, so that a wide, shallow register file has little similarity with a narrow, deep file even if the total number of storage bits is similar.

This storage similarity metric can be augmented to account for ”register waste,” that is, unused bits of storage that would result if the two operations shared storage. This gives preference to combining an operation with small register requirements with another similar operation, rather than one with large register requirements, even if the bits of common storage would be the same.

Figure 7(a) shows an example DFG. Consider the two operations $+_1$ and $+_2$. Both of them have an incoming edge from an add operation and an outgoing edge to a load operation; thus, the wire similarity metric is 64 (assuming 32-bit operations). Similarly, both operations will require the shift registers to hold their results for II cycles as there is an inter-iteration dependence from each add to itself; assuming $II = 4$, this translates to a storage similarity metric of 128. Thus the overall similarity between the two operations is 192. Assuming these are the most similar operations in the DFG, they will be added to the same operation set and scheduled together.

4.1.2 Backtracking

During modulo scheduling, it is possible that a set of operations cannot be scheduled due to conflicts with previously scheduled operations. In such a situation, it is necessary to use backtracking in order to maintain forward progress. When a conflict arises during traditional modulo scheduling, the operation is forcibly scheduled and conflicting operations are unscheduled and placed in the queue to be rescheduled later. The method of backtracking used in this scheduler is similar, but at the granularity of operation sets rather than individual operations. When a set cannot be scheduled, first it is determined which scheduled operation(s) is causing the conflict. Then, all operations in the same set as the conflicting operation are unscheduled. Finally the current set of operations is scheduled, and the unscheduled set is later rescheduled.

In general, backtracking has an adverse effect on the solution quality. This is because each set is optimally scheduled given the previously scheduled sets. If some of these previous sets are later unscheduled, the current set is no longer optimal. In addition, the sets are effectively scheduled out of priority order, which can potentially decrease the amount of hardware sharing that is achieved.

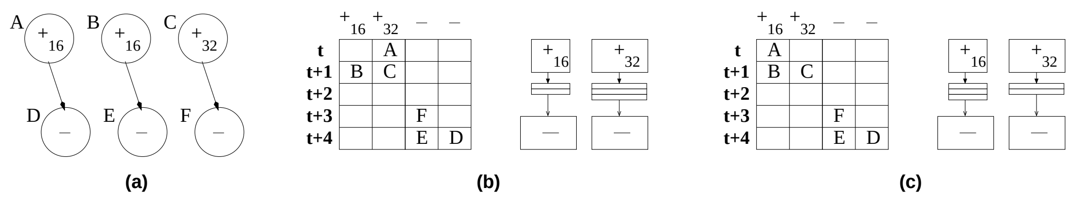


Figure 8: Effect of space-time decomposition.

4.2 Time and Space Decomposition

The job of a scheduler is to assign both a schedule time and an FU (e.g., “space”) to every operation in the loop body. In the context of the schedulers described in Section 3, assigning both time and space for an operation in a single pass has a multiplicative effect on the search combinatorics. For example, in the branch-and-bound scheduler, the number of possibilities for an operation to be explored by the scheduler is the product of number of time slots possible for the operation and the number of FUs to which the operation can be assigned. Similarly, in the ILP scheduler, the number of variables introduced by Equation 9 is equal to II times number of FUs for every operation in the loop body.

The problem of scheduling can be decomposed into its two constituent phases: (1) assigning a time slot to every operation, (2) fixing the operations in space. Note that the first phase still has to honor resource restrictions, i.e., it cannot assign more operations to a time slot than there are FUs available to execute those operations. The second phase of assigning operations to FUs should ensure that it does not assign two operations scheduled in the same time slot to the same FU. Such an assignment is always made possible by enforcing the resource restrictions in the first phase. The number of possibilities for every operation is reduced from $O(II \times \#FUs)$ in the combined solution to $O(II) + O(\#FUs)$ in the decomposed solution. The decomposed scheduler phases still have to be cost sensitive. Due to the nature of decomposition, some optimizations may not be possible in a particular phase. In time-space decomposition, optimizing for FU cost and width of the shift register file is not possible in the first phase. This is because the cost of FUs and width of registers depend on the assignment of operations to FUs. However the time assignment phase can optimize the depth of the shift register files.

In the ILP scheduler, assigning valid time slots to operations can be enforced using the constraints given by Equations (2) through (5). Note that time slot assignment is sufficient to calculate the lifetime of the value produced by an operation i, given by Equation (12). Since the lifetimes LT_i directly affect the register depth, minimizing lifetimes is important. Therefore, $\sum_{i=0}^{N-1} BW_i \times LT_i$ is used as the objective function in the formulation. Note that the lifetimes of operations are weighted by their bitwidths BW_i. This is to ensure that lifetimes of narrow operations are not minimized at the cost of wide operations. Solving the set of constraints described above gives a time slot assignment for all operations in the loop. Now the space (resource) assignment can be performed by forming a new ILP problem which includes all equations described in Section 3.3. The objective function remains the same, given by Equation (19). However, the values of time slots $X_{i,s}$ and stages k_i obtained from time assignment phase are explicitly specified to the ILP problem formed in the space assignment phase. Thus the second phase problem size is reduced greatly, because only resource assignments have to be computed.

4.3 Space and Time Decomposition

In this decomposition, the scheduling problem solved in two phases, namely, FU assignment phase followed by time assignment phase. This has the effect of optimizing the FU cost and shift registers' width before optimizing the depth of shift register files.

In the ILP scheduler, the formulation for space assignment consists of Equations (6) and (7). The objective function used in this phase is $\sum_{j=0}^{N-1} W_j$, where the W_j's are given by Equation (15). Thus, the FU assignment phase reduces the sum of widths of the FUs. Note that this minimizes both the FU cost and the width of shift register files. Now, the time assignment can be performed by forming an ILP problem which includes all equations described in Section 3.3, and explicitly specifying the values for $R_{i,j}$'s obtained from the FU assignment phase.

Figure 8 illustrates a negative effect of phase ordering the scheduling problem into FU assignment followed by time assignment. Figure 8(a) shows part of the dataflow graph of a loop. There are two 16-bit adds feeding subtract operations and a 32-bit add feeding a subtract operation. Suppose the machine has a budget for 2 adders and 2 subtractors and let the subtract operations be identical in width. The goal of FU assignment phase is to minimize the FU costs. Figures 8(b) and 8(c) show two possible assignments which result in the same FU cost of a 32-bit adder, a 16-bit adder and two subtractors. The crucial difference however is that operation A is assigned to the 32-bit adder in Figure 8(b) and the 16-bit adder in figure 8(c). Note that both these assignments result in the same FU cost, and there is no way for the FU assignment phase to differentiate between these two solutions. Now consider the time assignment phase. Suppose that, due to other data dependencies, the only possible time assignment is as shown in either of the Figures 8(b) or (c). The separation of the operations due to the schedule in Figure 8(b) results in $16 \times 2 + 32 \times 3 = 128$ register bits. However, the schedule in Figure 8(c) results only in $16 \times 3 + 32 \times 2 = 112$ register bits. Thus, phase ordering could result in some sub-optimality.

5. EXPERIMENTAL RESULTS

Loop kernels from several application domains are used to evaluate cost sensitive modulo scheduling. idct, dequant and dcacrecon are loops from MPEG-4; fsed, sobel, and sharp are image processing loops; blowfish and sha are used in encryption applications; lyapunov is a mathematical kernel; and viterbi, fft, fir, and iir are commonly used in signal processing. The sizes of the loops range from 24 operations for iir up to 120 operations for idct. In general, loops for these applications can have intra loop code and may not be perfectly nested. For the experiments, we manually convert the loop kernels to a single perfectly nested `for` loop. Only the innermost loop is considered for modulo scheduling. The numbers reported below correspond to hardware generated for the innermost loop only.

For each benchmark, we use the compiler-directed loop accelerator synthesis system described in Section 2. After FU allocation, various cost sensitive scheduling algorithms are evaluated. From the resulting schedules, the hardware datapath and control path is generated and the resulting RTL is synthesized to obtain gate counts. Synthesis is performed with the Synopsys design compiler in 0.18μ technology. A 200-MHz clock rate is assumed.

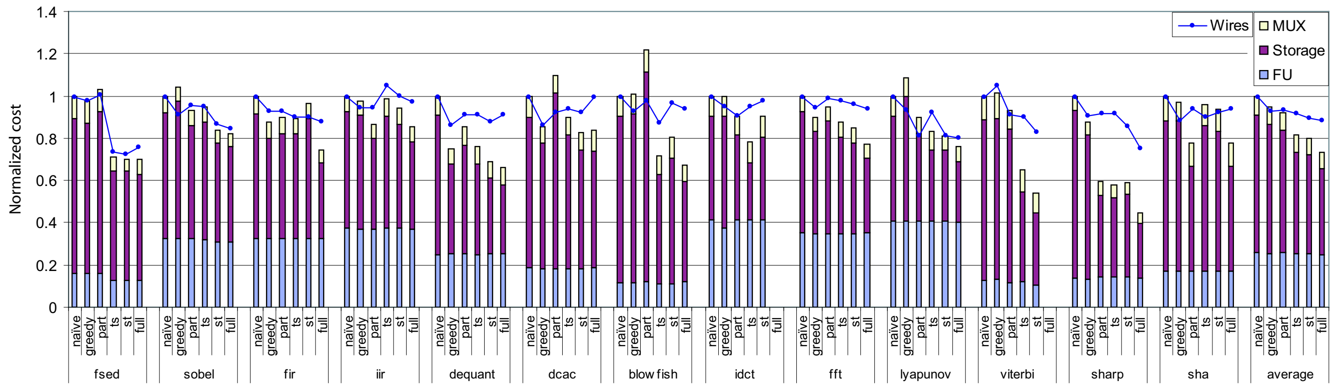


Figure 9: Hardware cost breakdown of loop accelerators synthesized using various scheduling techniques, relative to naïve scheduler.

The ILP scheduler is used for most experiments; however, the BNB scheduler is used for the experiments that vary the cost objectives or partitioning method. The two schedulers are both exact solutions; thus, we do not compare them with each other. Their use in certain experiments is a software engineering decision as some experiments are more amenable to one formulation or the other.

The first experiment, shown in Figure 9, evaluates the effectiveness of the different decomposition methods in reducing hardware cost. The baseline in this experiment is the hardware resulting from the naïve, iterative modulo scheduler [27] followed by a stage scheduling postpass [9] as implemented in the Trimaran compiler framework [29], and is represented by 1.0. This baseline result is shown by the first bar of each benchmark; all bars are divided into three segments, representing the contribution of MUXes, storage, and FUs to the overall cost. The remaining bars are as follows: the second bar shows the greedy algorithm described in Section 3.1; the third is the partitioned scheduler described in Section 4.1, using the priority-based partitioning method with a set size of 16 operations; the fourth is the time-space decomposition described in Section 4.2; the fifth is the space-time decomposition described in Section 4.3; and the sixth bar shows the optimal solution. Note that for some large benchmarks (idct and viterbi) this value could not be obtained due to the problem complexity, emphasizing the need for problem decomposition. The number of interconnect wires relative to the naïve scheduler is also shown in this figure as lines superimposed on the bars.

In this graph, FU cost does not differ significantly across schedulers. This is because FU capabilities are fixed prior to scheduling and most schedules result in the same or similar FU cost. The overall gate savings is significant in many benchmarks. The time-space decomposition scheduling achieves gate savings of 42% for sharp.

The greedy scheduler achieves only 5% gate savings on average and sometimes performs worse than the naïve scheduler. This is because it considers only one operation at a time and can be trapped in local minima. The average gate savings achieved by the partitioned, time-space and space-time scheduling methods are 8%, 19% and 20% respectively. In general, the time-space and space-time decomposition methods perform well as they are able to consider all operations at once. This is an advantage because the final machine cost is due to the combined effects of all operations rather than individual scheduling decisions. The partitioned cost sensitive scheduler results in slightly more gates than the naïve scheduler for some benchmarks like fsed, dcacrecon, and blowfish. This is due to the locally greedy nature of the decomposition. Also, for large benchmarks, the fixed-sized operation sets make up smaller fractions of the whole loop and thus the algorithm becomes greedier as it “sees” less of the loop at once.

The optimal scheduler achieves 27% savings over the naïve scheduler. For some benchmarks (iir, sha) the partitioned scheduler performs near optimal. The time-space and space-time decomposed schedulers are able to achieve near optimal for many benchmarks while only requiring a fraction of the runtime. Both time-space and space-time schedulers produce high quality solutions and can practically handle large problem sizes. Thus, we believe these methods to be the best choices for accelerator synthesis. The two perform differently according to the application characteristics: space-time performs better for loops with more bitwidth variation (sobel, viterbi) while time-space performs better for loops with more register lifetime variation (blowfish, idct).

Generally the number of wires decreases as gate count decreases. On average, the wire savings achieved for the three decomposed scheduling methods are 7%, 8%, and 10% for partitioned, time-space, and space-time respectively. In many cases, the wire cost of the optimal solution is higher than the wire cost for one of the decomposed solutions; this is because the optimal scheduling formulation does not account for wire cost.

The next experiment shows the effect of changing the hardware cost objective. The objective discussed thus far has been minimizing the sum of logic (storage and FUs) and wires. The weights of these components can be modified; for example, if interconnect cost is a dominating factor, the weight of the interconnect wires can be increased as a fraction of overall cost. The BNB scheduler can naturally accommodate these varying cost components. Figure 10 shows the breakdown of FU, storage, MUX, and wire costs relative to the baseline which optimizes the sum of these components. Each curve represents the machine resulting from scheduling with a certain cost objective; optimizing wires alone and optimizing logic gates (storage + FU) alone are presented. In Figure 10(a), iir is shown; when optimizing for wires, the cost of storage increases while wire cost decreases slightly. Conversely, optimizing for gates reduces the storage cost but increases wires slightly. Figure 10(b) shows the sha benchmark; interestingly, optimizing for gates reduces the number of wires. This is a product of the problem decomposition, which is imperfect – the baseline scheduler is unable to exploit wire sharing even though the gate optimizing scheduler happens to do so successfully. Figure 10(c) shows the average across all benchmarks; note that the wire optimizing scheduler did not save wires on average (though the wire count remains low). This is because jointly optimizing both gates and wires naturally results in good wire sharing (as fewer connections are made between fewer logic gates), and optimizing only wires does not improve on this for most benchmarks.

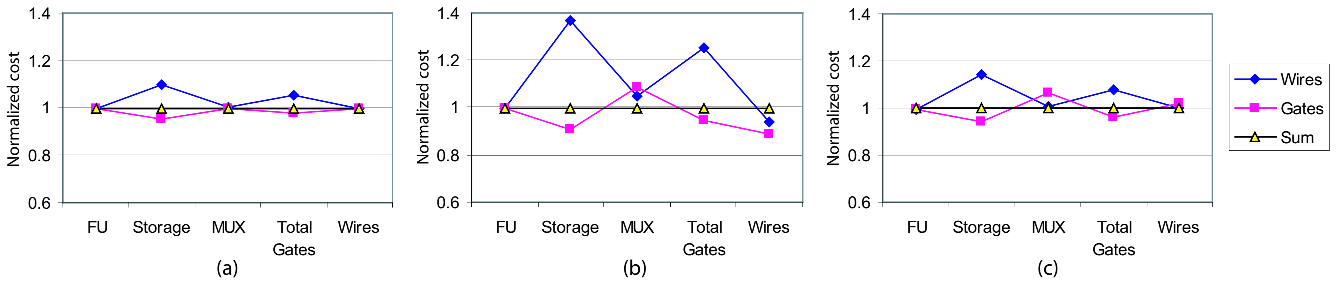


Figure 10: Effect of different cost objectives on (a) iir, (b) sha, and (c) average across all benchmarks.

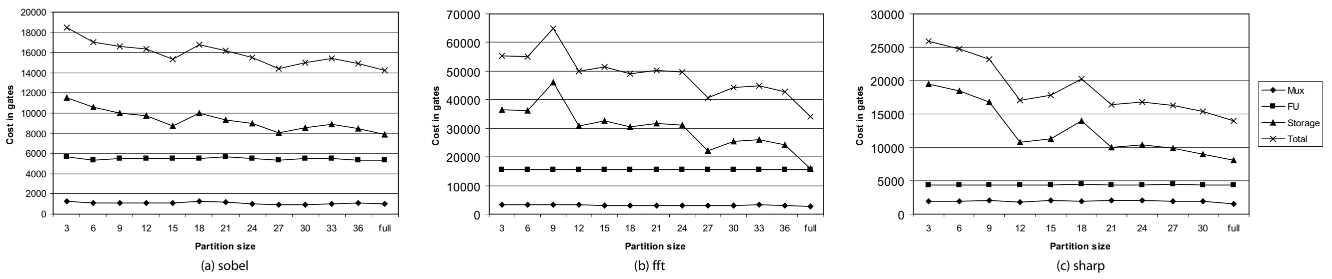


Figure 11: Effect of partition size on hardware cost.

To show the effect of set size on the partitioned scheduler discussed in Section 4.1, Figure 11 shows the hardware cost of scheduling the sobel, fft, and sharp benchmarks with varying set sizes. The set size is varied from 3 operations per set up to "full", where all operations are in one set. Each graph shows four lines, representing the FU, MUX, storage, and total gate costs at each set size. First, note that for these benchmarks, FU cost remains largely constant as there is little bitwidth variation among the data values, and no width specialization is performed. Second, the storage cost is where the scheduler is able to take the most advantage of larger set sizes. Third, the hardware cost decreases as set size increases, closely tracking the storage cost decrease. As expected, with larger set sizes, the scheduler is able to exploit hardware sharing across more operations at once. However, the overall cost generally nears optimal before the partition size becomes very large. For example, for sobel, a partition size of 15 gives a gate cost within 6% of optimal. Thus, the scheduler is often able to obtain good results while partitioning the operations into small sets.

In the ILP scheduler, CPLEX was used to solve the ILP formulations. A timelimit of 6 hours was enforced for the ILP formulations leading to fully optimal solutions. CPLEX reports the best solution seen so far when the timelimit expires. Thus the numbers reported for the optimal solution in Figure 9 corresponds to this best solution. For the time-space and space-time decompositions, CPLEX runtimes were between 30 seconds for the smaller benchmarks like `fir` to 2 hours for the larger benchmarks like `idct` and `viterbi`. The CPLEX runtimes for partitioned ILP formulation were less than a second for smaller partitions sizes and a maximum of 80 minutes for the bigger partition sizes, irrespective of the benchmarks. Note that the time taken by the rest of the compiler phases is non-significant (less than a minute) compared to the CPLEX runs.

For the partitioned scheduler, various partitioning strategies are investigated as discussed in Section 4.1.1. Figure 12 shows the resulting hardware cost of various partitioning methods for select benchmarks and the overall average. A partition size of 8 is used in these experiments. rand refers to random partitioning and usually performs worse than the naïve, cost unaware scheduler. prio (priority-based) is the best on average, and is the method used in other experiments involving the partitioned scheduler. mincut refers to a standard graph partitioner which attempts to minimize edge cuts; we use the Metis [16] partitioner. This method can perform poorly as it does not account for hardware cost. sim (similarity-based) and sim-w (similarity-based with waste accounting) perform better than mincut but are hampered by backtracking effects as discussed in Section 4.1.2. Note that not all benchmarks could be scheduled using all partition methods (due to backtracking effects), so the cost average includes only benchmarks that could be scheduled using all methods. Thus, the average cost of prio is not the same as in Figure 9.

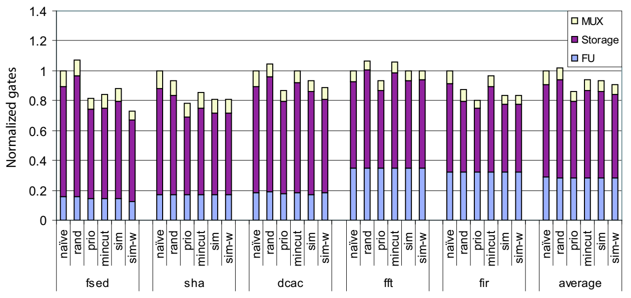


Figure 12: Cost breakdown for various partitioning methods.

6. CONCLUSION

This paper addresses the problem of cost sensitive modulo scheduling in a loop accelerator synthesis system. Scheduling decisions must be made with the goal of decreasing the cost of hardware that is generated from the final schedule. Traditional modulo schedulers are not suitable in this context as they are unaware of the effect of scheduling decisions on hardware cost. Two exact solutions, branch-and-bound and ILP, are presented to solve this problem. In addition, three methods of decomposing the problem are presented which allow the algorithm to solve realistic problems. The decomposition techniques work either by partitioning the dataflow graph into smaller subgraphs and optimally scheduling the subgraphs, or by splitting the scheduling problem into two phases, time slot and resource assignment. All decomposition methods were successful at making increasing problem sizes tractable, and depending on the application, different decomposition methods performed better than others. Since the final cost depends on the combined effects of all operations, the time-space and space-time methods, which consider all operations together, worked best. Overall, cost sensitive modulo scheduling increases hardware efficiency of automatically synthesized loop accelerators by an average of 8–20%, with individual savings of up to 42% over a naïve scheduler.

7. ACKNOWLEDGMENTS

Thanks to Nathan Clark and 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.

8. REFERENCES

[1] E. R. Altman and G. A. Gao. Optimal modulo scheduling through enumeration. *International Journal of Parallel Programming*, 26(3):313–344, 1998.

[2] 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.

[3] S. Bakshi and D. Gajski. Components selection for high performance pipelines. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(2):182–194, June 1996.

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

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

[6] 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.

[7] A. E. Eichenberger, E. Davidson, and S. G. Abraham. Minimum register requirements for a modulo schedule. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 75–84, Nov. 1994.

[8] A. E. Eichenberger, E. Davidson, and S. G. Abraham. Optimum modulo schedules for minimum register requirements. In *Proc. of the 1995 International Conference on Supercomputing*, pages 31–40, July 1995.

[9] A. E. Eichenberger and E. S. Davidson. Stage scheduling: A technique to reduce the register requirements of a modulo schedule. In *Proc. of the 28th Annual International Symposium on Microarchitecture*, pages 338–349, Nov. 1995.

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

[11] 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.

[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] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 258–267, June 1993.

[16] G. Karypis and V. Kumar. *Metis: A Software Package for Paritioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparce Matrices*. University of Minnesota, Sept. 1998.

[17] J. Lee, Y. Hsu, and Y. Lin. A new integer linear programming formulation for the scheduling problem in data-path synthesis. In *Proc. of the 1989 International Conference on Computer Aided Design*, pages 20–23, 1989.

[18] J. Llosa et al. Swing modulo scheduling: A lifetime-sensitive approach. In *Proc. of the 5th International Conference on Parallel Architectures and Compilation Techniques*, pages 80–86, 1996.

[19] J. Llosa and S. Freudenberger. Reduced code size modulo scheduling in the absence of hardware support. In *Proc. of the 35th Annual International Symposium on Microarchitecture*, pages 99–110, 2002.

[20] S. Mahlke et al. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1355–1371, Nov. 2001.

[21] 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.

[22] 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.

[23] I.-C. Park and C.-M. Kyung. Fast and near optimal scheduling in automatic data path synthesis. In *Proc. of the 28th Design Automation Conference*, pages 680–685, 1991.

[24] N. Park and F. Kurdahi. Module assignment and interconnect sharing in register-transfer synthesis of pipelined data paths. In *Proc. of the 1989 International Conference on Computer Aided Design*, pages 16–19, Nov. 1989.

[25] 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.

[26] 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.

[27] 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.

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

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

[30] 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.

[31] 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.