FLASH: Foresighted Latency-Aware Scheduling Heuristic for Processors with Customized Datapaths

Manjunath Kudlur, Kevin Fan, Michael Chu, Rajiv Ravindran, Nathan Clark, Scott Mahlke

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

Abstract

Application-specific instruction set processors (ASIPs) have the potential to meet the challenging cost, performance, and power goals of future embedded processors by customizing the hardware to suit an application. A central problem is creating compilers that are capable of dealing with the heterogeneous and non-uniform hardware created by the customization process. The processor datapath provides an effective area to customize, but specialized datapaths often have non-uniform connectivity between the function units, making the effective latency of a function unit dependent on the consuming operation. Traditional instruction schedulers break down in this environment due to their locally greedy nature of binding the best choice for a single operation even though that choice may be poor due to a lack of communication paths. To effectively schedule with non-uniform connectivity, we propose a foresighted latency-aware scheduling heuristic (FLASH) that performs lookahead across future scheduling steps to estimate the effects of a potential binding. FLASH combines a set of lookahead heuristics to achieve effective foresight with low compile-time overhead.

1. Introduction

The next generation of embedded processors will need to perform computationally demanding processing of images, sound, video, and network packets while achieving both low cost and minimal power dissipation. Application-specific instruction set processors (ASIPs) provide an effective strategy to meet these challenging demands. With ASIPs, the hardware design is customized to specifically suit the computation needs of the application. All aspects of the processor architecture and microarchitecture can potentially be customized, including the memory system, fetch and issue logic, datapath, and control path. Through specialization, large design wins are achievable.

One area of work that has been overlooked by many researchers is compilation support for ASIPs. Compiler support is critical so that these devices do not require hand-coded assembly code. The central challenge is that the hardware customization process often creates heterogeneous ASIPs with non-standard hardware structures and irregular connectivity. Traditional compiler techniques break down in this heterogeneous environment. In the worst case, they fail to produce legal code. More commonly, they yield poor quality code that fails to take maximum advantage of the customizations.

For this paper, we focus on one area of customization, non-uniform function unit (FU) connectivity arising when a processor datapath is customized. Non-uniform interconnect can arise in both FU to FU as well as FU to register file connectivity. The latter issue has been investigated by other researchers in the context of multicluster processors [4, 11, 13, 17, 23] and DSPs [19, 20, 21]. This paper focuses on compilation with non-uniform FU-FU connectivity. FUs in a pipelined architecture are generally interconnected through a register bypass network. *Register bypasses*, or data forwarding paths, eliminate data hazards in pipelined processors. With bypassing, datapaths and control logic are provided so that an operation's result is available for subsequent operations before it has been written to an architectural register, thereby reducing the effective latency of operations. Bypassing was first introduced in the IBM Stretch and is considered a standard part of all modern pipelined processors [8].

A fully bypassed processor allows an FU to read its inputs from the outputs of any FU from any of the pipeline stages subsequent to execution. The cost of implementing a full bypass network is significant, both in terms of area and wire delay [24]. For instance, the Alpha 21064 has 45 separate bypass paths [22]. As the number of FUs increases or as pipeline depth grows, bypass cost increases substantially. Furthermore, architectural mechanisms such as predicated execution increase bypass complexity [15]. The cost of the bypass network is contrasted with the observation that many of the bypass paths are underutilized or not used at all for any specific application. Therefore, there is opportunity to construct an ASIP with a small fraction of the bypass paths that will have substantial cost savings and comparable performance to a processor with full bypass [3, 14].

The difficulty with a partial versus full bypass network is that operation latencies are no longer fixed. Rather, they are a function of the specific placement of the producing and consuming operations. With full bypass, the latency of a flow dependence edge (read after write) is the latency of the FU on which the producer is placed as the bypass network is equally capable of transferring the result to any consumer. With partial bypass, the latency varies based on the specific FU placement of both the producer and consumer operations. The maximum latency (which occurs when no bypass path is available) is the FU latency plus the time to transfer the result to the architectural register file. Note that the latency is not simply a choice of two values, i.e. the end points of this range. Rather, any value in the range may be possible since bypasses can be selectively removed from any pipeline stage following execution back to the FU inputs.

Variable dependence edge latencies cause difficulties with the instruction scheduler. These difficulties arise because instruction schedulers make locally greedy decisions for assigning operations to FUs. Each operation is scheduled on the first available FU such that its start time is minimized. With full bypass, greedy scheduling has proved quite effective. However with partial bypass, the greedy choice for the current operation may result in poor assignment choices for dependent operations as there may be no bypass paths available for those operations to communicate values. As a result, schedules are lengthened as communication is forced to go through the register file.

To deal with the variable latency problem, we propose a foresighted latency-aware scheduling heuristic, or FLASH. FLASH performs a lookahead across future scheduling steps to estimate how the current placement of an operation onto an FU$\times$time-slot affects future decisions. A full lookahead across the entire dataflow graph could be done, considering every possible operation$\times$FU$\times$time placement, however this is infeasible in terms of compile time for any nontrivial application. Thus, the scheduling foresight must be carried out judiciously. FLASH combines several effective heuristics for achieving foresight with low overhead. First, lookahead distance is constrained to a small amount. Second, the combinatorics are reduced by considering dependence paths independently and in isolation. Third, the average slack of a dependence path is used to weight the relative importance of one path over another. With FLASH, the scheduler is able to make more intelligent placement decisions to effectively reduce schedule lengths on processors with partial bypass while incurring only a modest compile-time overhead.

2. Background and Motivation

2.1. Hardware Overview

The architectural model used throughout this work is that of a statically-scheduled, pipelined VLIW processor, as shown in Figure 1(a). In such a hardware model, the latency of a given type of operation is typically fixed and does not depend on the specific resources used to execute it. For example, a simple arithmetic operation generally takes one cycle regardless of which ALU executes the operation. The scheduler for such an ideal architecture does not need to worry about which ALU an arithmetic operation is placed on, and only needs to find a time to execute it that would minimize the total schedule length (in itself a difficult problem).

However, real machines, especially those customized for specific application domains, may have effective latencies that vary depending on the specific resources used by an operation. These variable latencies can arise for several reasons. A partially-connected register bypass network, for example, will cause latencies between producer and consumer operations to depend on whether or not data bypasses exist between the FUs executing the operations. Another example is the use of variable-speed FUs – for example, a processor could have a fast multiplier as well as a slow multiplier in order to provide flexibility for performance and power considerations during scheduling. Yet another example of variable latency is in the use of narrow bitwidth FUs to perform both narrow and wide computations [27]. In this case, wide computations can either be broken up by the compiler or by the hardware; in the latter case, the effective FU latency is variable. In the above cases, scheduling operations now becomes a much more difficult problem, as a poor FU choice for an operation can have ramifications later in the schedule.

In this work, we focus on VLIW machines that have variable latency due to a partial bypass network. The bypass network logic has previously been shown to be a major factor in the delay in modern processors [24]. In addition, as issue width in current processors increases and pipelines become deeper, the cost of a full bypass network significantly increases. Given an issue width *i* and *n* pipeline stages after execution to bypass from, the full-bypass model on two-input FUs would require $(2 \times i^2 \times n)$ bypass paths, result-

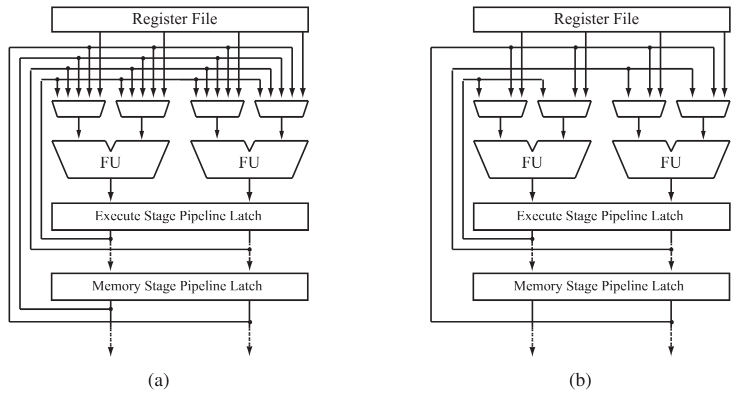

Register File
FU
FU
Execute Stage Pipeline Latch
Memory Stage Pipeline Latch
(a)
Register File
FU
FU
Execute Stage Pipeline Latch
Memory Stage Pipeline Latch
(b)

Figure 1. (a) A VLIW processor pipeline with full bypass (b) A pipeline with partial bypass.

ing in a significant cost on very wide machines. Figure 1(a) shows an example of a datapath with a full bypass network. Each stage has the ability to forward computed results back to the execute stage for other operations. Limiting the number of bypass paths between FUs has been shown to have good results in cutting cost while minimizing the effect on performance. In Figure 1(b), the same datapath is shown with some low-utilized bypass paths removed. In this example, each FU can forward data to itself after the execute stage and only one of the two FUs can forward data after the memory stage.

2.2. Motivation

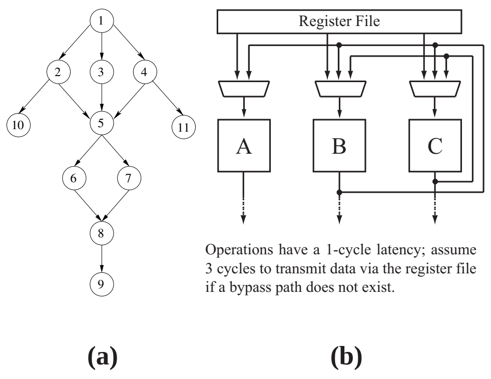
Given a machine with several bypass paths removed, intelligent compilation support becomes much more critical. With a full bypass machine, the choice of which specific FU to execute an operation on has no bearing on the latencies of future operations. In the case of a partial bypass machine, poor FU choices can have dramatic effects on the schedule length; the scheduler must consider the different delays involved with routing operands between two dependent operations through the network depending on the FUs they are placed on. A poor choice for an FU to execute on can keep dependent operations from executing sooner because of a lack of bypass paths, and can also preclude other dependent chains, which would prefer to use the scheduled operation's bypass path, from executing.

An intelligent scheduler not only must decide on when to execute each operation, it must decide which FU to execute it on and consider the effects of using that bypass path. Scheduling two dependent operations on FUs without a bypass path between them can dramatically change the schedule length. By the same token, placing operations that have significant slack or no consumers on the FUs with the most important bypass paths can unnecessarily constrain the schedule.

Consider the dataflow graph (DFG) in Figure 2(a), which we will use as an example throughout the rest of the paper. The datapath shown in Figure 2(b), where arrows indicate bypass paths between FUs, is an example of a machine with partial bypass. For the purposes of the example, assume each FU is universal with unit latency. A 3-cycle latency is assumed for the data to be written to the register file after execution. If the bypass path is used, the consumer can be scheduled in the cycle immediately following the producer; if not, then the consumer must wait 3 cycles for the result to be written back to the register file.

A traditional list scheduler considers operations in a prioritized manner. The scheduler checks the resources available (FUs are the only resource considered in this discussion) to find the earliest available cycle for the operation to execute and schedules it at that time. Thus, using the example from Figure 2(a), assume operations are numbered with their priority, so operation 1 would be the first one scheduled. The traditional scheduler has no notion of the bypass limitations of the processor. Its only objective is to select the FU that allows the current operation to be scheduled at the earliest cycle. In the case of a tie, an FU is selected arbitrarily. For this example, let us assume that the scheduler simply picks the FUs in alphabetical order when ties occur. Such an arbitrary selection is typical with most schedulers.

Using this strategy, operation 1 is assigned to FU *A*. The process continues for each operation creating the 16-cycle schedule as shown in Figure 2(c). It is easy to see that the uninformed decisions made by the scheduler can result in poor schedules. A better scheduler could have made different placement decisions resulting in the schedule shown in Figure 2(d) which is half the length. The more compact schedule was achieved by making effective use of the available bypass paths to reduce critical inter-operation communication latency. The central challenge of the scheduler is to place operations so that producer-consumer communication can be carried out across the available bypass paths.


Register File
A
B
C
Operations have a 1-cycle latency; assume 3 cycles to transmit data via the register file if a bypass path does not exist.
(a)
(b)

time	*A*	*B*	*C*
0	1		
1			
2			
3	2	3	4
4			
5			
6	5	10	11
7			
8			
9	6	7	
10			
11			
12	8		
13			
14			
15	9		

(c)

time	*A*	*B*	*C*
0		1	
1	4	2	3
2			
3			
4	10	5	11
5		6	7
6		8	
7	9		

(d)

Figure 2. (a) Example DFG, (b) Machine with partial bypass, Schedules generated by (c) naïve scheduler and (d) scheduler using FLASH.

However, not all communication requires the use of bypass paths. The key is that producer-consumer operations on the critical path are placed to enable low-latency communication through the bypass paths, such as operations 1-2 and 1-3. Conversely, non-critical operations often do not require the bypass paths (e.g., operations 2-10 and 4-11), so the scheduler needs to ensure they do not unnecessarily utilize the bypass paths, thereby precluding critical operations.

2.3. Related Work

Previous work has studied the positive effects of limiting bypass connections, but failed to resolve the scheduling issues that arise with varying latencies between operations. Ahuja et al. [3] performed the initial study on the impact of partial bypass, which focused on detailing the amount of performance lost due to interlock stalls from missing bypasses on a scalar processor. Other work [1, 9, 12] has found that limiting the connections of the bypass network can result in little impact on performance while greatly decreasing cost.

In our previous work [14], we started with a fully connected bypass network and systematically removed unnecessary bypass paths based on utilization statistics. Given a machine with partial bypass, code was sent through an FU prioritization phase which first annotated each operation with a list of preferred FU alternatives for assignment. This list was created by examining possible FU assignments and available bypass paths to forward results between them. Assignments were ranked in a way which best makes use of the resources. The annotated list was then used as a hint to help the scheduler decide which FU to execute the operation on. Two methods were used, one using a bottom-up greedy approach and the other using a linear programming technique.

Past work has studied the effects of scheduling in a lookahead manner In the realtime domain, where each operation has a set range of time in which it must execute in order to generate a valid schedule. Allan et al. [5] introduced the concept of foresighted instruction scheduling, which places operations only if their placement does not constrain the schedules of future operations into forcing an invalid schedule. This greatly increases the likelihood of generating valid schedules, but breaks down when a more constraining operation is scheduled after an unconstrained one, due to factors such as having a higher priority. Beaty introduced a technique called Lookahead Scheduling [6, 7], which not only looked at whether the placement of an operation would constrain future placements, but also placed those future operations so other unconstrained operations would not prevent a valid schedule from forming. Both the foresighted and lookahead techniques differ from our study in that they have a notion of valid and invalid schedules; their focus is mainly to place operations in any way possible to find a valid schedule.

Lookahead based scheduling schemes have also been studied in the context of job scheduling [26] to minimize average job waiting time and in the area of artificial intelligence for reducing the search space in solving constraint satisfaction problems [16]. Backtracking schedulers have the potential to generate better schedules at the expense of compile-time by unscheduling operations that have been scheduled to make room for the current operation. They have been evaluated for effectively filling branch delay slots in [2]. Also, Iterative Modulo Scheduling [25], an algorithm for software pipelining innermost loops, is based on backtracking.

Previous work in multi-cluster compilation has similarities to our work, as a scheduler for such architectures must consider the added latencies required for scheduling dependent operations on different clusters. Buss et al. [10] investigated a similar problem but was more focused on minimizing the amount of inter-cluster moves necessary in a clustered microarchitecture. Although the topic of their study was similar, they approached the problem from another angle. They first scheduled the code in a way where dependent operations had a preference to be scheduled to the same FU. Then, after scheduling, they decided on a grouping into clusters and removed unnecessary bypass paths. Our work approaches the problem from the standpoint of scheduling for an already fixed partial bypass network. Another related work is that of Özer et al. [23], whose Unified Assign and Schedule algorithm dealt with multiple latencies resulting from operations assigned to different clusters in a multi-cluster VLIW datapath. They used a modified list scheduler, which implicitly adjusted flow-dependency latencies if multiple predecessors were preassigned to different clusters, thus requiring an inter-cluster move to be inserted.

3. FLASH Technique

The FLASH technique is described in this section. Section 3.1 describes the naïve list scheduler, which is our baseline, and point out its shortcomings when scheduling for a machine with partial bypass. A scheduling technique using exhaustive search is described in Section 3.2 in order to provide some insights into the complexity of the problem of scheduling for a machine with partial bypass. In Section 3.3, some key observations are made that drove the design of the FLASH algorithm. The algorithm is described in Section 3.4.

3.1. Naïve List Scheduler

Algorithm 1 shows the operation of a regular list scheduling algorithm. The algorithm maintains an ordered list of operations that have yet to be scheduled, called the *ready list*. Typically, a height based prioritization is used to order the list, i.e. the higher an operation is in the data dependence graph, the higher its priority. This ensures that producers are scheduled before consumers, which is essential for one-pass scheduling. The algorithm picks an operation from the list and tries to schedule it at the earliest possible time (**stime**). Of course, there may be no free resource available to execute the operation at **stime**. As shown in Line 1, the algorithm checks for resource availability at a later time, looping until a time slot is found when a free resource is available to execute the operation. The operation is scheduled on that resource at that time slot.

The main drawback of Algorithm 1 when applied to a machine with partial bypass concerns Line 2 in the pseudo code. The scheduler greedily schedules an operation on a

```
   while (ReadyList is non-empty)
   do
       op ← Next unscheduled operation in priority order ;
       stime ← Earliest time when op can be scheduled ;
1      while (no free resource available to execute op at stime)
       do
           stime ← stime + 1 ;
       end
2      res ← Free resource capable of executing op ;
3      schedule(op, res, stime);
   end
```

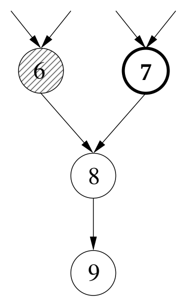
Algorithm 1. List scheduling algorithm.

resource that is available at the earliest time. Furthermore, when more than one resource is available to execute the operation, the scheduler picks one arbitrarily. These choices can result in good schedule quality on a machine with uniform/complete bypass, however, intelligent choices of resources and times for operations becomes crucial on a machine with partial bypass. As described in Section 2, the naïve scheduler achieves a 16 cycle schedule for the DFG shown in Figure 2(a) on a machine shown in Figure 2(b), which is 50% longer than necessary..

3.2. Exhaustive Scheduler

One way to improve the scheduling decision of an FU for an operation is to consider the effects of the current decision on all future decisions, accounting for all of the interactions between dependent operations, resources, and bypass paths. In other words, for each possible resource and time assignment for a given operation, the scheduler could look at possible resource and time assignments for future operations, taking into account bypass path availability, and choose the assignment that gives the shortest overall schedule length. Figure 3 shows a portion of the DFG shown earlier. Assume that operation 6 (shaded) has already been scheduled on FU C at time 0, and operation 7 (in bold) is currently being scheduled. The table in Figure 3 shows some of the possible resource and time assignments that can be given to operations 7 through 9, taking into account resource and bypass path availability. The goal is to minimize the start time of operation 9, which can be achieved by performing any of the assignments in the shaded rows. Therefore, the best choice would be to schedule operation 7 on FU B at time 0.

Note that there are quite a lot of scheduling possibilities. Each of the three operations can be assigned on any of FUs A, B, or C. Furthermore, for each of these operation-to-FU assignments, the operation can be scheduled in various time slots. Notice in Figure 3, for example, that when considering operation 7 on FU C, the overall schedule length is shorter when operation 7 is scheduled at time 2 rather than time 1. This is because operation 8 is constrained by the availability of bypasses from both operations 6 and 7. Thus, it is insufficient to simply try the earliest time that resources are available.



op 7		op 8		op 9	
FU	start	FU	start	FU	start
A	0	A	3	A	6
A	0	A	3	B	6
A	0	A	3	C	6
A	0	B	3	A	4
A	0	B	3	B	4
A	0	B	3	C	4
A	0	C	3	A	6
A	0	C	3	B	4
A	0	C	3	C	4
B	0	A	3	A	6
B	0	A	3	B	6
B	0	A	3	C	6
B	0	B	1	A	2
B	0	B	1	B	2
B	0	B	1	C	2
B	0	C	1	A	4
B	0	C	1	B	2
B	0	C	1	C	2

op 7		op 8		op 9	
FU	start	FU	start	FU	start
C	1	A	4	A	7
C	1	A	4	B	7
C	1	A	4	C	7
C	1	B	4	A	5
C	1	B	4	B	5
C	1	B	4	C	5
C	1	C	4	A	7
C	1	C	4	B	5
C	1	C	4	C	5
C	2	A	5	A	8
C	2	A	5	B	8
C	2	A	5	C	8
C	2	B	3	A	4
C	2	B	3	B	4
C	2	B	3	C	4
C	2	C	3	A	6
C	2	C	3	B	4
C	2	C	3	C	4

Figure 3. In the DFG, assume that operation 6 is scheduled at time 0 on FU C. Tables show some possible combinations of valid FU and start time assignments for operations 7, 8, 9. The shaded rows show assignments that give the earliest start time for operation 9.

In general, for a simple machine where all FUs can execute any operation, the number of possibilities that an exhaustive foresighted scheduler would need to consider is $(r \times t)^n$, where r is the number of FUs, t is the number of time slots that must be considered for each operation-to-FU assignment, and n is the number of operations from the current operation to the end of the scheduling region. Assuming that three time slots need to be checked for each possible operation-to-FU assignment in our mini-example[1], this yields $(3 \times 3)^3 = 729$ possibilities. Clearly, for a larger example the number of possibilities would grow out of hand.

3.3. FLASH Overview

In this section we describe FLASH, which enhances a regular list scheduler when scheduling for a machine with non-uniform resource latencies arising due to partial bypass interconnect. As seen in Section 3.2, the scheduler could resort to performing an exhaustive search, and decide on an FU and time slot for an operation depending on the best outcome from all possibilities. However, an exhaustive search is infeasible for real programs. We can intuitively see that a limited search can still be useful for deciding on an FU for an operation, though. This intuition is formalized in the foresighted latency-aware scheduling heuristic (FLASH). Concisely, FLASH performs a limited search of possibilities and evaluates these possibilities to decide on an FU for the current operation under consideration. We first provide an informal description of how FLASH tries to limit the search and provide a formal description of the heuristic in the next section. FLASH manages the search in four ways: limiting search depth, considering dependence chains independently, weighting chains by slack, and ignoring future resource constraints.

Limiting Search Depth. As seen in Section 3.2, the number of possibilities is dependent on the depth of the dependence chain of the DFG. Instead of considering all possibilities up to the last operation in a dependence chain, we can consider all possibilities for operations that are at a a certain depth of the chain. We parameterize FLASH by the depth in the dependence chain to which we perform the search. Intuitively, this is a good approximation as the placement decision of an operation has the largest effect on its immediate consumers. The next layer of consuming operations are affected, but indirectly by placement limitations on their producers caused by the placement of the original operation. Each layer adds another level of indirection, resulting in diluted effects further down the dependence chain. Therefore, most of the lookahead benefits can be extracted by examining a small window of future operations.

Considering Dependence Chains Independently. Even with a limited depth, the number of possibilities could be still large. In programs with high ILP, there can be a large number of dependence chains originating from a given operation. Suppose we are considering dependence chains of length l and there are n dependence chains originating from a given operation in the DFG. Assuming that all operations are uniform and there are r FUs in the machine to which these operations can be assigned, then there are $r^{n \times l+1}$ possible assignments of operations to FUs. However, FLASH considers each dependence chain independently, which gives only $n \times r^{l+1}$ possibilities.

The intuition behind this assumption is that latency-

1 Due to the maximum bypass latency being three cycles.

aware placement is critical in dependence height constrained code regions. Critical dependence chains need to be carefully placed onto FUs so that communication latency is minimized. The placement decision concerns a single operation, thus its effect on subsequent operations is the most important consideration. The interaction with other dependence paths is less important and will be accounted for when other operations are evaluated.

Weighting Dependence Chains. Dependence chains are considered independently, but intuitively each is not equally important. Delaying critical paths is generally much more costly than non-critical paths and optimizing the critical paths generally leads to a better schedule. To account for the differential importance, FLASH weights the dependence chains based on their criticality (inverse of slack), so that the choice of FU for the current operation is better with respect to the more critical dependence chains.

Ignoring Resource Constraints. To accurately evaluate the effect of an assignment of operations to FUs on the schedule time, we need to lookup the resource reservation table to see if an FU is free at various times depending on the assignment we are evaluating. This will mean $n \times r^{l+1}$ lookups to the reservation table are needed when an operation has n dependence chains originating from it and all l-deep dependence chains are evaluated. When evaluating a possible assignment in FLASH, however, we make the simplifying assumption that FUs are free to execute the operations. Thereby, we avoid the costly lookups into the reservation table.

The intuition behind this assumption is that latency-aware placement is less important in resource limited code regions. When the code is resource limited, increases in producer-consumer latencies due to poor FU placement will have a small affect. Conversely, in dependence limited code regions, efficient use of bypasses is critical to achieving a compact schedule. In these regions, sufficient resources are generally available. Thus, FLASH assumes sufficient resources are available and optimizes placement exclusively for latency. For resource constrained code regions these decisions will not be very effective, but they will not hurt performance. Conversely, in dependence constrained regions, good decisions will be made.

3.4. FLASH Algorithm

The pseudo code of the list scheduler using FLASH is shown in Algorithm 2. The main addition to the naïve list scheduling algorithm is in Line 1 of Algorithm 2. For every possible resource **r**$_{op}$ that the current operation **op** can be scheduled on, we compute the FLASH_RANK, which is an estimate of the earliest schedule time of all the **k**-th successors of **op**. We then choose the resource **res** which

```
  Data      : k, the evaluation depth
  begin
      while (ReadyList is non-empty)
      do
          op ← Next unscheduled operation in priority order ;
          stime ← Earliest time when op can be scheduled ;
          while (no free resource available to execute
                  op at stime)
          do
              stime ← stime + 1 ;
          end
          res_set ← Set of all resources that can execute op ;
1         res ← $r_{op}$ ∈ res_set such that
                  FLASH_RANK(op, $r_{op}$, k) is minimum;
2         while (res is not free at stime)
          do
3             stime ← stime + 1;
          end
          schedule(op, res, stime);
      end
  end
```

Algorithm 2. List scheduling algorithm using FLASH.

has the minimum FLASH_RANK, i.e. the resource which makes the schedule time of **k**-th successor of **op** as early as possible. Lines 2-3 of Algorithm 2 ensure that the scheduler schedules **op** on **res**, even if it is not available early. This is markedly different from the naïve scheduler which greedily picks whichever resource is available at the earliest time.

```
  Algorithm: FLASH_RANK
  Data      : a, the operation to be scheduled, $r_a$, the resource on
              which it can be scheduled, k, the evaluation depth.
  RANK = -MAXINT ;
  for ($S_i \in \{S_1^k, S_2^k, ...S_p^k\}$, all dependence chains
      originating from a)
  do
      $(a_{i1}, a_{i2}, ...a_{ik}) \leftarrow S_i$ ;
1     best_sched_time ← MAXINT ;
2     for (($r_2 \in R_2, r_3 \in R_3, ... , r_k \in R_k$),
          $R_j$ is the set of FUs that can execute $a_{ij}$)
      do
3         sched_time ← CST ($a_{i1}, ...a_{ik}, r_a, r_2, ... r_k$) ;
4         if sched_time < best_sched_time then
5             best_sched_time ← sched_time ;
          end
      end
6     rank_of_S ← $\frac{1}{slack(a_{ik})+1}$ × best_sched_time;
7     if rank_of_S > RANK then
8         RANK ← rank_of_S ;
      end
  end
  return RANK;
```

Algorithm 3. Function to compute FLASH_RANK of an operation.

Algorithm 3 shows the pseudo code for the FLASH_RANK function. Given an operation **a** and a resource **r**$_a$ that the operation can be scheduled on, this function estimates the schedule length of all **k**-deep dependence chains originating from **a**. Note that we parameterize FLASH_RANK by **k**, the depth to which we limit our evaluation in the data flow graph. Assume the **k**-deep dependence chains of operation **a** are named $\{S_1^k, S_2^k, ...S_p^k\}$. Note that each dependence chain has **k** operations and there are **p** dependence chains. Each of S_i^k is an ordered list of operations, $S_i^k = \{a_{i1}, a_{i2}, ...a_{ik}\}$, and the first element of each of these ordered lists is the operation **a**, i.e. a_{ij} = **a** for $j = 1$ and $i = 1, ..., p$. For each dependence chain S_i^k, Lines 1-5 calculate the best schedule length possible, by considering all possible assignments of the operations $a_{i1}, a_{i2}, ...a_{ik}$ to FUs. Note that the best schedule length of S_i^k is the earliest schedule time of operation a_{ik}. Let R_1, R_2, ... , R_k be the sets of FUs which can execute operations a_{i1}, a_{i2}, ... , a_{ik} respectively. Note that some of these sets could be equal if they correspond to similar operations. The number of possible ways of scheduling these **k** operations is $|R_1| \times |R_2| \times ... \times |R_k|$. Given the operations a_{i1}, a_{i2}, ... , a_{ik} and the FUs $r_1 \in R_1$, $r_2 \in R_2$, ... , $r_k \in R_k$, Algorithm 4 calculates the schedule time of a_{ik} on r_k. As mentioned before, this function does not consider whether the resources are free, thus avoiding lookups to the reservation table. Thus, we get only an estimate of the schedule time of a_{ik}.

The rank of the current dependence chain is calculated in Line 6 of Algorithm 3 which is the schedule length of the dependence chain weighted by $\frac{1}{slack(a_{ik})+1}$. This multiplicative factor ensures that we give less weight to dependence chains which are not on the critical path. Lines 7-8 make sure we return the rank which is maximum of the ranks of the dependence chains $\{S_1^k, S_2^k, ...S_p^k\}$. Thus, the function FLASH_RANK essentially computes:
MAX(weighted-min-sched-length(S_1^k), weighted-min-sched-length(S_2^k), ..., weighted-min-sched-length(S_p^k)),
given that the operation **a** is scheduled on FU $\mathbf{r}_a$. The MAX function ensures that we get the (best) schedule length of the dependence chain that is most constrained by latency. Also, since we are weighting these schedule lengths by the criticality of the dependence chains, we see that FLASH_RANK is a very good estimate of how early a critical operation dependent on the current operation **a** can start executing, if we schedule it on $\mathbf{r}_a$. By choosing an FU for **a** with the minimum FLASH_RANK, we ensure that a critical operation dependent on **a** can start early.

3.5. FLASH Example

To illustrate the application of the FLASH algorithm, we return to the example in Figure 2(a) and walk through the scheduling steps for some operations in the DFG. Consider operation 1. Let us limit our lookahead depth to 1 for simplicity. There are three first successors for operation 1, namely operations 2, 3, and 4. Each of these operations can be executed on any of the FUs A, B, or C. Also, since all the operations are on the longest path to the exit operation 9, their slack is 0. Table 1 lists the possible schedule times for operations 1 and 2. Note that operation 1 can be scheduled at a later cycle, but these choices will only increase the schedule time of operation 2, and are not shown in the table. The best schedule time for operation 2 is cycle 1 and can be achieved by scheduling operation 1 on either FU B or FU C. The possible schedule times for the operation pairs 1–3 and 1–4 look similar to Table 1, and the best schedule times for operations 3 and 4 is achieved by scheduling operation 1 on either FU B or FU C. Thus, Algorithm 3 will return the FLASH_RANK of 1 for both FUs B and C, and the scheduler can assign operation 1 to either FU B or C. Assume the scheduler chooses function units in alphabetical order and assigns operation 1 to FU B.

Now, let us consider operation 4. The scheduler has already scheduled operations 1 and 2 on FU B and operation 3 on FU C. One of the first successors of operation 4 is operation 5 and both operations have slack 0. Table 2 shows the possible schedule times for operations 4 and 5. The earliest time that operation 4 could be scheduled is 1 and only FU A

```
Algorithm: CST
Data        : $(a_1, a_2, ..., a_k, r_1, r_2, ...r_k)$
stime ← earliest time $a_1$ can be scheduled on $r_1$
for (i ← 2 to k)
do
    etime ← MAX(stime + latency of operation $a_{i-1}$,
            earliest start time of $a_i$);
    while (no bypass path exists between $r_i$ and $r_{i-1}$
            at cycle (etime - stime))
    do
        etime ← etime + 1;
    end
    stime ← etime ;
end
return stime ;
```

Algorithm 4. CST : (COMPUTE_SCHED_TIME) : Function to find schedule length of a dependence chain, given an assignment of FUs to all operations in the chain.

Operation 1		Operation 2	
FU	stime	FU	stime
A	0	A	3
A	0	B	3
A	0	C	3
B	0	A	1
B	0	B	1
B	0	C	1
C	0	A	3
C	0	B	1
C	0	C	1

Table 1. Estimating schedule time for operation 1.

Operation 4		Operation 5	
FU	stime	FU	stime
A	1	A	4
A	1	B	4
A	1	C	4
A	2	A	5
A	2	B	5
A	2	C	5
B	2	A	5
B	2	B	5
B	2	C	5
C	2	A	5
C	2	B	5
C	2	C	5

Table 2. Estimating schedule time for operation 4.

is free at cycle 1. If operation 4 is scheduled on FU A at cycle 1, then the earliest time operation 5 can be scheduled is cycle 4, because FU A does not have any bypass paths from its outputs. Note that operation 4 can also be scheduled at cycle 2 in any of the FUs. When operation 4 is scheduled at cycle 2, the best schedule time for operation 5 is cycle 5. Operation 4 also has another first successor, namely operation 11, which has a slack of 3. The weighting factor in line 6 of the Algorithm 3 for operation 11 is $\frac{1}{3+1} = 0.25$. Consequently, the values of **rank_of_S** corresponding to operation 5 will dominate those of operation 11 and eventually Algorithm 3 will return the FLASH_RANK of 4 for operation 4. Therefore, the scheduler schedules operation 4 on FU A at cycle 1.

The schedule that follows is shown in Figure 2(d). Note that the length of this schedule is only 8 cycles as compared to the length of the schedule generated by a naïve scheduler, which is 16 cycles. Also, we see that there is better utilization of the bypass paths in the schedule generated by scheduler using FLASH. The next section shows the performance of FLASH on some real benchmarks.

4. Experimental Results

The proposed system was implemented using the Trimaran toolset [28], a retargetable compiler framework for VLIW/EPIC processors. Experiments were run on a subset[2] of the SPECint2000 and MediaBench [18] benchmark suites. The machine model used was a VLIW machine capable of issuing 4 integer, 2 floating point, 2 memory and 1 branch instructions per cycle. The FU latencies are similar to those of the Intel Itanium. A perfect memory system with 2-cycle load latency is assumed.

Two configurations of the bypass network were used for the experiments. The first configuration is specific to individual benchmarks and is the machine which can sustain 95% of the performance of a fully bypassed configuration. This is termed the application-specific bypass configuration. In our previous work [14], we found that on average, only 40% of the cost of nonzero-utilized bypass paths is required to sustain 95% of the performance, so these machines represent designs that might reasonably be obtained with application-specific customization. Detailed discussion on how this bypass configuration is determined is given in our previous work. In the second configuration, FUs are divided into two groups, each group having 2 integer, 1 floating point and 1 memory unit. The FUs in a group bypass data among themselves without any extra latency. One integer unit from each of the groups can bypass to the branch unit as well. A pictorial representation of the machine is shown in Figure 4. In the experiments, we compare FLASH with the naïve list scheduling algorithm and also with another scheduling heuristic [14] which assigns resource preferences to operations during a prescheduling phase. This heuristic is referred to as Greedy Resource Preference (GRP) and is based on the Bottom-Up Greedy method proposed in [13].

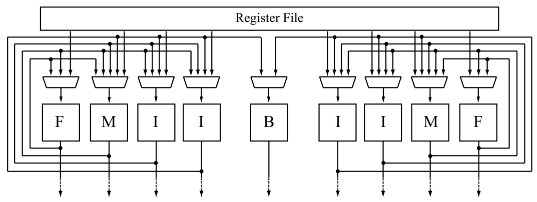


Figure 4. Bypass configuration of the two-cluster-like machine.

Figure 5 shows the speedup achieved by FLASH over the naïve list scheduler on a machine with application specific bypass configuration. Speedups achieved by the GRP scheduling heuristic are also shown. The labels in the figure refer to the depth FLASH was limited to for evaluation of all possibilities. For example, LA1 refers to a lookahead depth of 1, where only the next operation in the dependence chain is considered for deciding FU assignment.

Figure 5 shows that on average, LA1 performs 24% better than the naïve scheduler, with a maximum speedup of 55% for the *rawdaudio* benchmark. The scheduler using FLASH achieves an average of 4% more speedup than a scheduler using GRP. Increasing the lookahead depth of FLASH from 1 to 2, yields 1% more speedup. Speedup does not improve for lookaheads beyond 3, however. As explained in Section 3.3, the placement decisions for an operation has maximum effect mostly on its immediate consumers. Thus, considering operations further down the dependence chain has little effect on placement decisions for many benchmarks. For some benchmarks, the speedup achieved using a smaller lookahead is actually better than

2 Benchmarks were omitted from the results because they could not be successfully compiled with Trimaran.

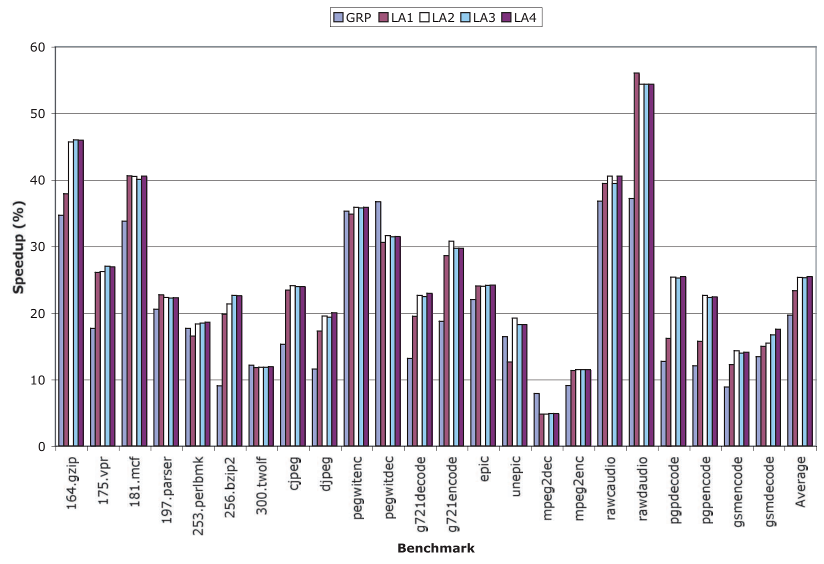


Figure 5. Speedup with GRP and FLASH compared to the traditional list scheduler on a machine with application-specific bypass configuration.

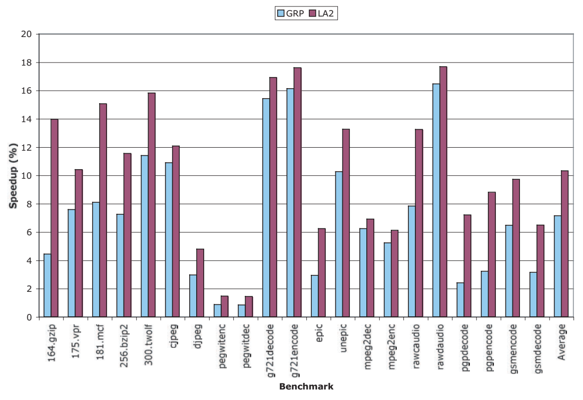


Figure 6. Speedup with GRP and FLASH compared to the traditional list scheduler on a machine with a two-cluster-like configuration.

the speedups when using a larger lookahead. This is mostly due to the simplifying assumptions in the heuristic dealing with evaluation of FU assignments. Dependence chains are evaluated independently in the heuristic and no reservation table lookups are performed to check whether the FUs are free. This sometimes leads to imperfections in the FLASH_RANK for an operation.

Note that, even though FLASH performs better than GRP on average, GRP performs better than FLASH in some benchmarks like `253.perlbmk`, `300.twolf`, `unepic`, and `mpeg2dec`, as seen in Figure 5. As described in [14], GRP takes a global view of the scheduling region when assigning resource preferences. The global view enables better decisions and hence better schedules in some cases compared to the local decisions made by FLASH. However, since GRP is done as a pre-scheduling phase, it is not adaptive when the actual scheduler decisions differ from the suggested resource preferences. This behavior can occur because of the greedy nature of the scheduler or the cross-interaction of dependence paths that the GRP heuristic ignores. In fact, the resource preferences for later operations tend to be noticeably less useful than for earlier operations due to this behavior. On the other hand, since FLASH is applied during scheduling, it is able to better adapt to each successive scheduler decision. An adaptive GRP algorithm that changes resource preferences dynamically can potentially yield better schedules, and could be the subject of future research.

Figure 6 shows the speedup achieved by FLASH scheduler on a machine with two-cluster like configuration. The FLASH scheduler achieves a speedup of 11%, which is 4% more than the speedup achieved by the GRP scheduler. Since this machine has more bypass paths than the previous configuration, the naïve scheduler itself achieves a good schedule. Therefore, the headroom available for the FLASH scheduler is much less compared to the previous configuration. This explains why the average speedup in these experiments is 11% as compared to 24% for the previous configuration.

Figure 7 compares the schedule lengths achieved by the FLASH scheduler on a machine with two-cluster like con-

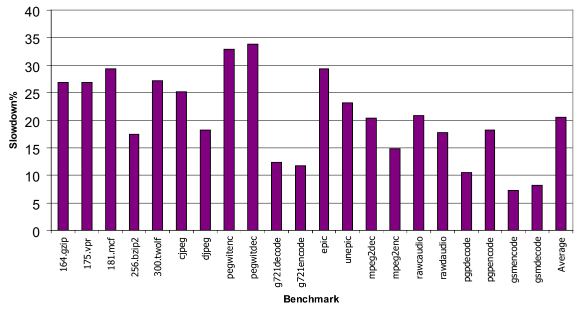


Figure 7. Slowdown with FLASH on a two-cluster-like machine compared to the traditional list scheduler on a machine with full bypass.

figuration to the schedule lengths achieved by the naïve scheduler on a machine with complete bypass. Lookahead depth of 2 was used for this experiment. The schedule lengths achieved on a machine with full bypass is almost like an upper-bound on the performance of the FLASH scheduler, because schedule length on a machine with partial bypass cannot be shorter than the schedule length on a machine with complete bypass. The average slowdown across the MediaBench and SPECint2000 benchmarks is 21%. In other words, the FLASH scheduler is able to achieve performance within 21% of the upperbound on an average. `pegwitenc` and `pegwitdec` have more than 30% slowdown. Since the configuration is similar to a 2-cluster machine latency-wise, a global partitioning-based approach is better suited than FLASH in this case. However, FLASH performs well on configurations that have irregular latencies at a fine grain level.

Figure 8 shows the utilizations of the bypass paths on the machines with application-specific bypass paths when the application was scheduled with the naïve scheduler and FLASH scheduler. The y-axis shows the average number of bypass paths used per cycle. The figure shows that bypass utilization is 1.5 paths per cycle when the application is scheduled using FLASH as compared to 1.1 paths per cycle when the naïve scheduler is used. This demonstrates that the FLASH scheduler chooses FUs that make use of bypass paths more often than the naïve scheduler. Thus, the intelligent choice of FUs for operations increases the utilization of the bypass network, consequently improving the performance of the application.

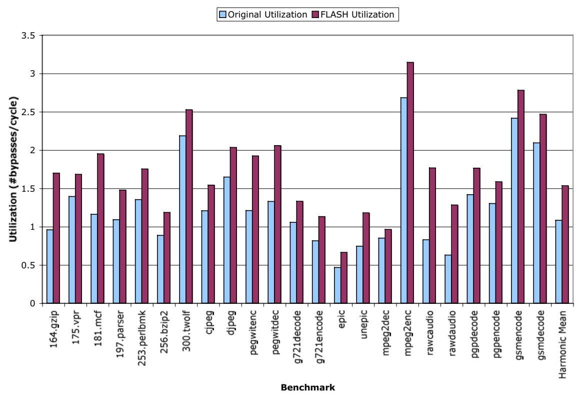


Figure 8. Number of bypass paths utilized per cycle when scheduled with naïve and FLASH schedulers.

Table 3 shows the average slowdown in scheduling times with the FLASH scheduler. The first row shows the average slowdown over all benchmarks. For some benchmarks the compilation time was very small, so even though the absolute slowdown was only a couple of seconds, the slowdown ratio was more than 100%. These outliers were ignored when computing the average slowdowns. The table shows that LA1 has a modest slowdown of 26%, whereas LA4 has 59% slowdown, once the outliers are accounted for. However, since scheduling is only one phase in the entire compilation run, we expect that the overall slowdown will be only modest. Thus, FLASH can be practically included in compiler software.

5. Conclusion

In this paper, we have developed a foresighted latency-aware scheduling heuristic (FLASH) to intelligently schedule operations on application-specific processor datapaths. Customized datapaths often result in non-uniform latency between the function units. Traditional instruction schedulers break down in this environment due to their locally greedy nature of binding the best choice for a single operation even though that choice may lead to poor choices for dependent operations. FLASH combines a set of simple lookahead heuristics to achieve effective foresight with low compile-time overhead. For a set of application-specific datapaths, FLASH achieves an average performance improvement of 23% to 25% as the lookahead depth is varied from one to four operations over a traditional scheduler. This performance gain is achieved by putting bypass paths to more effective use. FLASH increases bypass utilization by 41%.

6. Acknowledgments

We thank the anonymous referees for their helpful comments and suggestions. This research was supported in part by NSF ITR grant CCR-0325898, ARM Limited, the DARPA/MARCO C2S2 Research Center, and equipment donated by Intel Corporation.

	GRP	LA1	LA2	LA3	LA4
Average	22%	30%	58%	58%	103%
Average w/o outliers	22%	26%	52%	52%	59%

Table 3. Average slowdown in scheduling times.

References

[1] A. Abnous and N. Bagherzadeh. Pipelining and Bypassing in a VLIW processor. *IEEE Transactions on Parallel and Distributed Systems*, 5(6), June 1995.

[2] S. Abraham. Efficient Backtracking Instruction Schedulers. Technical Report HPL-2000-56, Hewlett-Packard Laboratories, May 2000.

[3] P. Ahuja, D. Clark, and A. Rogers. The Performance Impact of Incomplete Bypassing in Processor Pipelines. In *Proc. of Micro-28*, Dec. 1995.

[4] A. Aletà, J. Codina, J. Sánchez, and A. González. Graph-Partitioning Based Instruction Scheduling for Clustered Processors. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, Dec. 2001.

[5] V. Allan et al. Foresighted Instruction Scheduling Under Timing Constraints. *IEEE Transactions on Computers*, 41(9), Sept. 1992.

[6] S. J. Beaty. Lookahead Scheduling. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, Dec. 1–4, 1992.

[7] S. J. Beaty. List Scheduling: Alone, with Foresight, and with Lookahead. In *Proc. of 1st International Conference on Massively Parallel Computing Systems*, May 1994.

[8] E. Bloch. The Engineering Design of the Stretch Computer. In *Proc. of the Easter Joint Computer Conf.*, 1959.

[9] M. D. Brown and Y. Patt. Using Internal Redundant Representations and Limited Bypass to Support Pipelined Adders and Register Files. In *Proc. of HPCA-8*, Feb 2001.

[10] M. Buss, R. Azevedo, P. Centoducatte, and G. Araujo. Tailoring Pipeline Bypassing and Functional Unit Mapping to Application in Clustered VLIW Architectures. In *Proc. of CASES*, Nov. 2001.

[11] M. Chu, K. Fan, and S. Mahlke. Region-based Hierarchical Operation Partitioning for Multicluster Processors. In *Proc. 2003 Conference on Programming Language Design and Implementation*, pages 300–311, June 2003.

[12] R. Cohn, T. Gross, M. Lam, and P. Tseng. Architecture and Compiler Tradeoffs for a Long Instrction Word Microprocessor. In *Proc. of ASPLOS-3*, April 1989.

[13] J. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, Cambridge, MA, 1985.

[14] K. Fan et al. Systematic Register Bypass Customization for Application-Specific Processors. In *Proc. of 14th International Conference on Application-specific Systems, Architectures and Processors*, 2003.

[15] E. Fetzer and J. Orton. A Fully-Bypassed 6-Issue Integer Datapath and Register File on an Itanium Microprocessor. In *ISSCC Digest of Technical Papers*, pages 420–421, Feb. 2002.

[16] S. Grant and B. Smith. The Phase Transition Behavior of Maintaining Arc Consistency. In *European Conference on Artifical Intelligence*, 1995.

[17] K. Kailas, K. Ebcioğlu, and A. Agrawala. CARS: A New Code Generation Framework for Clustered ILP Processors. In *Proceeding of the 2001 International Conference on High Performance Computer Architecture*, pages 133–142, 2001.

[18] C. Lee, M. Potkonjak, and W. Mangione-Smith. Media-Bench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proc. of Micro-30*, Dec. 1997.

[19] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, Boston, MA, 1997.

[20] R. Leupers. *Code Optimization Techniques for Embedded Processors - Methods, Algorithms, and Tools*. Kluwer Academic Publishers, Boston, MA, 2000.

[21] P. Marwedel and G. Goossens. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, Boston, MA, 1995.

[22] E. McLellan. The Alpha AXP Architcture and 21064 processor. *IEEE Micro*, 13(3):36–47, June 1993.

[23] E. Özer, S. Banerjia, and T. Conte. Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures. In *Proceedings of the 31th Annual International Symposium on Microarchitecture*, pages 308–315, Nov. 1998.

[24] S. Palacharla. *Complexity-Effective Superscalar Processors*. PhD thesis, Wisconsin, Madison, 1998.

[25] B. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proc of Micro-27*, Dec. 1994.

[26] A. Rifkin. The Utility of Foresight in Single Server Scheduling. In *Proc. of 30th annual Southeast regional conference*, pages 253–260, Apr. 1992.

[27] B. Shackleford et al. Memory-CPU Size Optimization for Embedded System Designs. In *Proc. of 34th Design Automation Conference*, June 1997.

[28] Trimaran. An Infrastructure for Research in ILP. http://www.trimaran.org.