Predicate-Aware Scheduling: A Technique for Reducing Resource Constraints

Mikhail Smelyanskiy Scott A. Mahlke Edward S. Davidson Hsien-Hsin S. Lee†

Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI 48109
{msmelyan, mahlke, davidson}@eecs.umich.edu

†School of ECE
Georgia Institute of Technology
Atlanta, GA 30332
leehs@ece.gatech.edu

ABSTRACT

Predicated execution enables the removal of branches wherein segments of branching code are converted into straight-line segments of conditional operations. An important, but generally ignored side effect of this transformation is that the compiler must assign distinct resources to all the predicated operations at a given time to ensure that those resources are available at run-time. However, a resource is only put to productive use when the predicates associated with its operations evaluate to True. We propose predicate-aware scheduling to reduce the superfluous commitment of resources to operations whose predicates evaluate to False at run-time. The central idea is to assign multiple operations to the same resource at the same time, thereby oversubscribing its use. This assignment is intelligently performed to ensure that no two operations simultaneously assigned to the same resource will have both of their predicates evaluate to True. Thus, no resource is dynamically oversubscribed. The overall effect of predicate aware scheduling is to use resources more efficiently, thereby increasing performance when resource constraints are a bottleneck.

Keywords: instruction scheduling, predicate analysis, predicated execution, resource utilization, software pipelining, VLIW processor

1. INTRODUCTION

Very long instruction word (VLIW) processors rely on an intelligent compiler for extracting, enhancing, and exposing sufficient instruction-level parallelism (ILP) to deliver high performance. To extract ILP more effectively in the presence of branches and reduce the overhead of branches, predicated (conditional) execution is often employed. With predicated execution, operations are augmented with an additional Boolean operand known as the guarding predicate. When the guarding predicate is True, the operation executes normally. Conversely, when it is False, the operation is nullified. Predicated execution can be exploited by compilers that use if-conversion to convert branching code into straight-line segments of predicated operations [2] [13]. As a result, many branches and the difficulties associated with them can be eliminated.

Though generally effective at dealing with branches, predicated execution introduces a serious overhead of its own. Predicated execution trades off sequential execution of conditional operations for increased resource requirements. If-conversion is additive with respect to resources across branches to which it is applied. For branches that are if-converted in a code segment, the resources of the *then* and *else* clauses are added to determine the overall resource requirements for the resultant sequence of predicated operations. Intuitively this makes sense, since to remove a branch both clauses must be scheduled with the appropriate one nullified at run-time. As a result, a compiler must apply if-conversion carefully to avoid oversaturation of the processor resources [12].

Compile-time assignment of resources (e.g., fetch slots, register ports, function units, memory ports) to predicated operations is traditionally handled in a conservative manner. The compiler assumes that any predicate may evaluate to True at run-time and accordingly ensures that all resources required by an operation are unconditionally available. However, this is not necessary. At run-time, operations require resources when their predicate evaluates to True. An operation with a False predicate only requires a subset of its resources. In particular, the resources from fetching the operation to determining that its predicate is False are necessary. All later resources assigned to a nullified operation are superfluous.

For a predicated architecture, processor resources can be broken down into two categories: must-use and may-use. A must-use resource is required by an operation regardless of its run-time predicate value. Conversely, a may-use resource is only required when an operation's predicate evaluates to True. The classification of resources into the two categories is based on the point in the processor pipeline where operations with False predicates are nullified. Resources before the nullification point are must-use; those after are may-use. Nullification later in the pipeline reduces the latency from predicate computations to uses of those predicates; nullification earlier in the pipeline minimizes the number of must-use resources.

To overcome the problem of superfluous resource utilization by nullified operations, we propose a technique referred to as **predicate aware scheduling**. The central idea of predicate aware scheduling is to allow static over-subscription of may-use resources wherein multiple operations are allowed to reserve the same resource at the same time. However, dynamic over-subscription of resources must not take place. Thus, the compiler must guarantee that no two operations that are assigned to the same resource at the same time will ever have their predicates both evaluate to True at run-time. The overall affect of predicate aware scheduling is to increase the utilization of may-use resources, thereby increasing processor performance. A secondary benefit is that with resource constraints lessened, more aggressive if-conversion can be applied to extract further benefit from branch elimination.

In order to accomplish predicate aware scheduling, predicate analysis is employed in the resource reservation process. Specifically, relational properties among the predicates used in a program segment are derived [12] [8] [17]. The most important property for this work is disjointness, wherein two predicates are disjoint if they can never evaluate to True at the same time. For instance with an *if-then-else* statement, the predicates controlling the *then* and *else* clauses are disjoint. Such predicate analysis is already used extensively in compilers to assist with dataflow analysis, optimization, and register allocation of predicated code [4] [5]. A set of operations with disjoint predicates is allowed to reserve a common resource as the compiler guarantees that at most one of these operations will be active at run-time.

One obvious alternative to predicate aware scheduling is to simply build a wider processor with more resources. When the number of resources is sufficiently large, the problem of resource contention goes away. However, this solution may have a high cost; additional function units, register file ports, busses, etc. may be necessary. For either cost- or power-sensitive environments, this may be unacceptable. Predicate-aware scheduling offers an approach to increase the utilization of existing processor resources and therefore increase application performance with a fixed set of resources.

For this paper, we present the necessary extensions to accommodate both predicate-aware acyclic scheduling and software pipelining. In the next section, a brief background on scheduling is presented followed by a motivational example to illustrate the potential benefit of predicate-aware scheduling. Section 3 contains a description of the compiler support used to accomplish predicate-aware acyclic scheduling and software pipelining. The effectiveness of the technique is evaluated experimentally on a sample predicate-aware processor in Section 4. The final two sections describe related work and present conclusions.

2. BACKGROUND AND MOTIVATION

Code scheduling refers to the process of binding operations to a time slot and a set of resources for execution. In this section, we briefly describe two common scheduling techniques: list scheduling (LS) [1] to schedule acyclic code regions and iterative modulo scheduling (IMS) [14] to schedule innermost loop regions. Each technique is applied to a basic block as a simple illustration.

The goal of LS is to find a valid schedule of minimum length for an acyclic code region. The minimum achievable schedule length is the maximum of two lower bounds. The resource-constrained lower bound is equal to the number of busy cycles required by the most heavily used resource during a single execution of the region. The latency-constrained lower bound is determined by the sum of the latencies along the longest path through the data dependence graph (i.e. the critical path) of the region.

The goal of IMS is to find a valid schedule for an innermost loop that can be overlapped with itself multiple times so that a constant interval between successive iterations (**Initiation Interval** (**II**)) is minimized. The II-cycle code region that achieves the maximum overlap between iterations is called the **kernel**. The scheduler chooses its initial II to be the maximum of two lower bounds. The resource-constrained lower bound (**ResMII**) is equal to the number of cycles that the most heavily used resource is busy during a single iteration of the loop. The recurrence-constrained lower bound (**RecMII**) is determined by the maximum ratio $\lceil D(C)/P(C) \rceil$

Function Unit	Operations	Mnemonics	Lat.
ALU (A)	Add	**add**	1
	Subtract	**sub**	1
	Multiply	**mult**	3
	Predicate Compare	**cmpp**	1,2
Memory (M)	Load	**load**	2
	Store	**store**	1
Branch (B)	Branch on condition	**if**	1

Table 1: Description of a sample processor

among all cycles C in the dependence graph, where $D(C)$ is the sum of the operation latencies over all edges of the cycle C, and $P(C)$ is the sum of all loop-carried dependence distances over those edges.

As the number of machine resources increases, recurrence and latency constraints begin to dominate the schedule length for LS and IMS techniques, respectively. In general, IMS is more resource-constraint bound than LS, because IMS can look for independent operations across loop iteration boundaries. Conversely, LS is limited to operations within a single execution of a code region. Note that loop unrolling in conjunction with LS can be applied to approximate the benefits of IMS for loops.

Both scheduling techniques schedule operations at particular cycles so that both data dependences and resource constraints are satisfied. To satisfy scheduling constraints, LS and IMS use a data structure known as the schedule reservation table (SRT). The SRT records the use of a particular resource by a specific operation at a given time [3] [14]. Scheduling at that time is permitted only if the resource usage does not result in a resource conflict, i.e. it does not attempt to reserve any resource at a time when some other operation already reserved that same resource, and no latency constraints of prior operations on which the operation being scheduled depends are violated. In addition, IMS uses a Modulo Reservation Table (MRT) to facilitate tracking the modulo constraints. The modulo constraint states that two operations that use the same resource may not be scheduled an integer multiple of II cycles apart from one another.

IMS is generally applied to single basic block innermost loops. In processors that support predicated execution, if-conversion [2] [20] is applied to broaden the class of loops that can be modulo scheduled. If-conversion can also be used in conjunction with LS on acyclic regions to increase the effectiveness of the scheduler.

2.1 Example Code Segment

To illustrate the application of conventional LS and IMS along with the potential benefits of making each predicate-aware, we consider a simple code example and processor model. The example processor can fetch and execute up to three operations per cycle has three fully pipelined function units as detailed in Table 1 [1]. The mnemonics for the various operations, binding of operations to units, and latency of operations are shown in the table.

The example code segment is a slightly modified loop extracted from the *unquantize_image()* function from the *epic* application in the Mediabench benchmark suite [11]. Figure 1(a) shows the C source for the loop. Figure 1(b) shows the assembly code for the loop after if-conversion. For conciseness of the example we

[1]To support predicate-aware scheduling, the cmpp latency must be increased by at least one cycle as discussed in Section 4.1. In this example, the cmpp latency is increased from 1 to 2 cycles.

```
for ( i = 0; i < im_size; i + +)
{
    prod = q_im[i] * bin_size;
    if (q_im[i] ≥ 1)
        res[i] = prod − correction;
    else
        res[i] = prod + correction;
}
```

(a) Source code

```
op1: t1 = load(i1, q_im) if p0;
op2: prod = mult(t1, tbs) if p0;
op3: p1,p2 = cmpp.lt.uu(t1, 1) if p0;
op4: t2 = sub(prod, tcor) if p1;
op5: t2 = add(prod, tcor) if p2;
op6: store(i1+ = 4, res, t2) if p0;
op7: if(i + + < im_size) goto op1 if p0;
```

(b) Assembly code after if-conversion (p0=True)

Figure 1: Example code segment

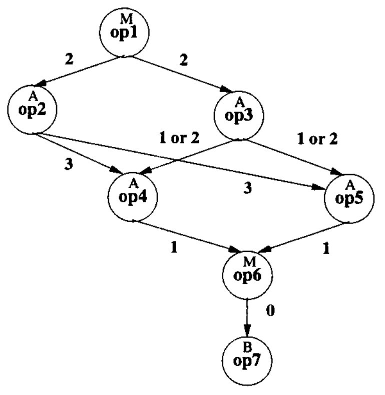


Figure 2: Data dependence graph for code segment

Time	A	M	B
0		op1	
1			
2	op2		
3	op3		
4			
5	op4		
6	op5		
7		op6	op7

(a) SchedLen = 8

Time	A		M	B
0			op1	
1				
2	op2			
3	op3			
4				
5	op4	op5		
6			op6	op7

(b) SchedLen = 7

Figure 3: LS schedule (a) versus PALS schedule (b)

assume that the instruction set supports post-increment load and store operations. In this example, the *if-then-else* statement is replaced by the corresponding predicate defining operation (p1,p2 = **cmpp.lt.uu**$(t1, 1)$ **if** $p0$). Predicate $p1$ is set to True and $p2$ is set to False when the *if* condition ($t1 < 1$) evaluates to True; whereas condition False sets $p1$ False and $p2$ True. The detailed semantics of the cmpp operations are described in [9].

The data dependence graph of the if-converted loop segment is presented in Figure 2. Each node is annotated with the type of the operation (A=ALU, M=memory, B=branch). Each edge is marked with the latency of that edge. Note that the edges in the graph are all flow dependences with the exception of the edge from op6 to op7 which is a control dependence.

2.2 Applying Predicate-Aware Scheduling

As stated above, the reservation table enforces resource constraints for both LS and IMS. That is, operations that use the same resource cannot be scheduled in the same cycle. Predicate-aware scheduling relaxes this constraint by allowing operations guarded by disjoint predicates (from here on referred to as disjoint operations) to reserve (or share) the same resource in the same clock cycle. The if-converted code shown in Figure 1(b) is used to illustrate LS and its counterpart predicate-aware list scheduling (PALS) along with IMS and its counterpart predicate-aware modulo scheduling (PAMS).

LS versus PALS

The application of LS to the example results in the 8 cycle schedule presented in Figure 3(a). This schedule is optimal for this machine model. *op4* is scheduled at cycle 5 which is the earliest time at which it can be scheduled. The earliest schedule time for *op5* is also cycle 5, but due to resource conflict with *op4*, it gets scheduled at the next cycle. Hence, the earliest schedule time for *op6*, which depends on both *op4* and *op5*, is cycle 7. Note that both *op4* and *op5* are executed conditionally but reserve the ALU unconditionally. In fact, only one of these operations is executed at run-time; the other is nullified. As a result, we effectively waste either cycle 5 or cycle 6 for each iteration of the loop because the ALU is not utilized during the cycle in which it executes a nullified operation.

With PALS, a 7 cycle schedule can be achieved, as shown in Figure 3(b). Operations *op4* and *op5* (from the *then* and *else* paths, respectively) can now be scheduled at their earliest schedule time; both operations may reserve the ALU in cycle 5 because they are provably disjoint, so only one will execute at run-time. In the SRT of Figure 3(b), each resource conceptually has two slots. This allows up to two disjoint operations to occupy the same resource at the same time. Two slots is not a restriction of this technique. Rather, for this example, there are only 2 control paths, thus we know that there can be at most two disjoint operations.

The overall result of the PALS schedule is that the ALU resource is always utilized in cycle 5 and the achieved schedule length is 7 cycles, a 14% speedup over the 8 cycle LS schedule shown in Figure 3(a). Note that for this basic block, a schedule of length 7 is optimal for any machine configuration. Since the latency-constrained lower bound (critical path length) is 7 cycles (in Figure 2).

IMS versus PAMS

The application of IMS to the example results in the II=4 schedule presented in the MRT shown in Figure 4(a). Since each of the four ALU operations (*op2, op3, op4, op5*) must reserve the ALU resource at a different cycle to avoid conflict, ResMII=4 and this schedule is optimal. Note that RecMII=1.

With PAMS, an II=3 schedule can be achieved as shown in Figure 4(b). Again, this improvement is achieved by enabling the provably disjoint operations, *op4* and *op5*, to reserve the ALU in the same cycle. Note that even though up to two disjoint operations can simultaneously reserve each function unit, the processor

Time	A	M	B
0	op5	op1	
1	op4	op6	
2	op2		
3	op3		op7

(a) II = 4

Time	A		M	B
0	op3		op1	
1	op4	op5		
2	op2		op6	op7

(b) II = 3

Figure 4: IMS kernel (a) versus PAMS kernel (b) schedule

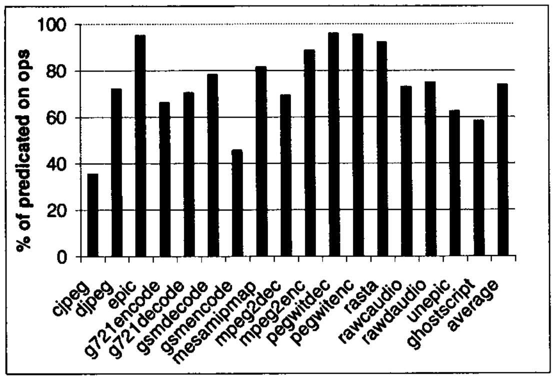


Figure 5: Ratio of operations predicated True

can only fetch a maximum of three operations per cycle. Thus, the scheduler must also ensure that the fetch width constraint is not violated. Overall, PAMS results in a 33% speedup over IMS for this example. Note, for this particular loop, an II=3 is optimal for any compiler strategy, because there is only one ALU and three operations that require it on each control path.

This example shows that by allowing disjoint operations to reserve the same resource in the same time-slot, the resource requirement for a code segment can be reduced. For code that is resource constrained, this results in a tighter schedule and hence performance improvement. Of course, if resources are not a limiting factor, the benefit of predicate-aware scheduling is lessened. If the example processor had two ALUs instead of one, both LS and IMS would achieve the optimal schedules for this example, 7 and 3 cycles, respectively.

2.3 Characteristics of Predicated Code

The previous section showed that an isolated example can derive benefit from predicate-aware scheduling. The central issue to motivate further discussion of this technique is whether applications in general have the properties that make them amenable to the approach. There are 3 interrelated questions to address: the number of predicated operations that are nullified at run-time, the fraction of time spent in regions with disjoint operations, and the potential to combine disjoint operations. For details related to the experimental methodology, the reader is referred to Section 4.3.

Figure 5 presents the percent of dynamic operations whose predicates evaluate to True during the program execution. Thus, 100 minus the height of the bar is the percent of nullified operations. On average, 26% of all dynamic operations are nullified. This means that 26% of the time that a function unit is reserved, it does no useful work.

Figure 6 addresses the last two questions. The left bar shows

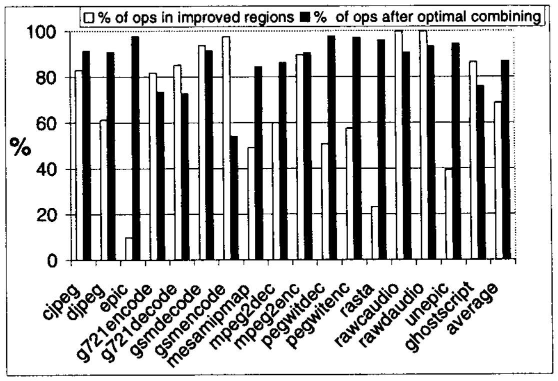


Figure 6: Optimal predicate-aware scheduling

the percent of dynamic operations that lie a regions with at least two disjoint operations. On average, 69% of the operations are from such regions. Of course, the overall benefit of the predicate-aware technique depends not only on the frequency of the improved region, but also on the number of disjoint operations in the region which indicates the potential for the improvement.

The right bar shows the dynamic combining potential. To derive the right bar, we optimistically assume maximal combining of the predicated code (regardless of operation type and latency) and count each group of combined operations as one operation. The bar shows the dynamic operation count after combining as a percentage of the count without combining. On average, optimal combining can reduce the total operation count by 13%. Note that this is not an upper bound on performance with predicate-aware scheduling; the actual performance benefits can be higher or lower as shown in the prior example.

3. PREDICATE-AWARE SCHEDULING

In this section, the details of the two predicate-aware scheduling algorithms are presented. Predicate-aware list scheduling (PALS) and predicate-aware modulo scheduling (PAMS) are extensions of conventional LS and IMS, respectively. As discussed in the previous section, both techniques aim to decrease schedule length by relaxing resource constraints, specifically by allowing disjoint operations to reserve the same resources in the same cycle.

LS and IMS share much of the same underlying scheduling infrastructure. Thus, we begin this section with a unified discussion of both algorithms, referred to as unified scheduling or simply scheduling. The term reservation table (RT) is used in a generic sense to represent either an SRT for LS or an MRT for IMS.

3.1 Baseline Unified Scheduling Algorithm

The heart of typical instruction scheduling algorithms employs two important functions to identify a conflict-free time for each operation to be scheduled. The central data structure used to identify resource conflicts is the RT. The general realization of an RT (similar to Figure 3(a)) is a two-dimensional matrix in which columns correspond to resources and rows correspond to schedule slots.

In our implementation, the scheduler selects an operation from the pool of unscheduled operations and calls the *FindTimeSlot* function (see pseudo code shown in Figure 7(a)). This function scans forward from *MinTime* to *MaxTime* looking for the first conflict free slot in RT to schedule the operation. *MinTime* is the ear-

```
FindTimeSlot(Operation, MinTime, MaxTime) {
    /* Successively try each time in the range */
    for ( CurrTime = MinTime; CurrTime ≤ MaxTime;
        CurrTime + +) {
        while ( there are remaining resource alternatives) do {
            resource_alt = next resource alternative for Operation
            if ( ResourceConflict(resource_alt, Operation,
                CurrTime) == FALSE )
                return CurrTime;
        }
        ...
}
```

(a) FindTimeSlot() function

```
ResourceConflict(resource_alt, Operation, CurrTime) {
    while (there are remaining resources in resource_alt) do {
        resource = next resource from resource_alt;
        if ( IS_EMPTY(ReservationTable [CurrTime][resource])
            == FALSE)
            return TRUE;
    }
    return FALSE;
}
```

(b) ResourceConflict() function

Figure 7: Baseline scheduling functions

liest start time that the operation can have as constrained by its scheduled predecessors. *MaxTime* (which can be infinity) is the latest time at which the scheduler will try to schedule the operation before giving up. Frequently, an operation may execute on any of multiple function units; in this case, the operation is said to have multiple alternatives. All operation alternatives are tried inside the *while* loop, and for each alternative, the function *ResourceConflict* is called.

The *ResourceConflict* function, shown in Figure 7(b), checks if the operation can be scheduled without conflict on the resource *resource_alt* at time *CurrTime*. Each *resource_alt* is a set of resources that one particular realization of the operation needs during execution. Therefore, each corresponding entry in the *ReservationTable* must be checked. If there are any conflicts, then the operation cannot be scheduled on this resource alternative at this time; otherwise, it can. Scheduling is accomplished at this level by reserving the appropriate entries in the RT.

3.2 Predicate-aware Extensions

Predicate-aware scheduling is accomplished by using the Predicate Query System (PQS) [8] to determine the disjointness of two operations based on their predicates. The PQS analyzes operations to determine relations between predicate values. These relations (or facts) are stored as Boolean expressions which can be efficiently manipulated. For a set of predicates, the Boolean expression essentially represents the disjunction of all the paths on which these predicates evaluate to True. For example, the predicate expression that represents *p0* in from Figure 1 is True, since the predicate evaluates to True on all the paths. To check if a predicate is disjoint from another predicate, the corresponding predicate expressions are ANDed. If the result is False, the predicates are disjoint, meaning that regardless of the execution path at most one of the predicates will be True at any given time. Otherwise, the predicates are not disjoint.

Predicate-aware scheduling uses a predicate-aware RT, as shown in Figure 8. Each entry in the predicate aware RT has two fields: a list of disjoint operations which have already reserved the entry, and a predicate expression (*pred_expr*), which represents the union of the predicates of the reserving operations.

In predicate-aware scheduling, *FindTimeSlot* calls the predicate-

Time	Res 1 ^{may}		Res 2 ^{may}				Res m ^{must}	
0	op1 op2	pred_expr01			...	...	op3	true
1					...	...	op4	true
...	...	...	...	...	...	...	...	...
n			op5	pred_exprn2	...	...		

Figure 8: Predicate-aware reservation table

```
ResourceConflict(resource_alt, Operation, CurrTime) {
    pred = get_predicate (Operation);
    while (there are remaining resources in resource_alt) do {
        resource = next resource from resource_alt;
        rt_entry = ReservationTable[CurrTime][resource]
        if ( IS_DISJOINT (rt_entry.pred_expr, pred) == FALSE)
            return TRUE;
    }
    return FALSE;
}
```

Figure 9: Predicate-aware ResourceConflict() function

aware *ResourceConflict* function (see Figure 9) which does the following. First, the operation's guarding predicate *pred* is obtained. For each entry *ReservationTable*[*CurrTime*][*resource*] of the predicate-aware RT, a call is made to the *IS_DISJOINT* function. This function takes two arguments: *pred* and the predicate expression *pred_expr* for this entry in the RT. If the conjunction of the two arguments is FALSE, the value returned is TRUE, then the operation is disjoint from every other operation in the list, and therefore it can also reserve the resource *resource* at time *CurrTime*. Otherwise the value returned is FALSE and the operation is not disjoint from one or more operations currently in the list. Therefore, this operation cannot reserve this resource at $CurrTime$.

If there are no resource conflicts, the operation is placed into the operation list of the corresponding entry in the RT. The operation's predicate is ORed into the current *pred_expr* in the RT entry to reflect the new condition under which the resource is busy.

The predicate-aware scheduler divides machine resources into two categories: may-use and must-use. May-use resources can be reserved in the same cycle by disjoint operations. Must-use resources can only be reserved by one particular operation in a given cycle, as on the baseline machine. The categorization rule is that every resource that is after the predicate nullification point in the pipeline is may-use. May-use resources can be reserved by disjoint operations because the operations whose predicates evaluate to False are discarded before those resources are used. Conversely, resources before the nullification point are must-use, and only one operation can reserve them at any time as these resources are used regardless of the operations's predicate value.

For our implementation, we add a pseudo must-use resource called the *fetch width* (or *FW*) resource. This resource limits the maximum number of operations that can be fetched in a given clock cycle. Note that in general the fetch width can differ from the execution width, which is the number of operations that can be simultaneously issued to function units in a given clock cycle. On all our processor models, these width are the same.

3.3 Additional Extension for PAMS

Up to this point, the predicate-aware extensions are common to both PALS and PAMS. However, to support PAMS, we must compute ResMII in a predicate-aware manner. IMS computes the resource-constrained lower bound (ResMII) by adding the number of times that each operation uses a particular type of resource to that resource's usage count. The cumulative usage count for the

Time	A^{may}		M^{may}		B^{may}		$FW1^{must}$	$FW2^{must}$	$FW3^{must}$
0			op1	p1 | p2			▨		
1									
2	op2	p1 | p2					▨		
3	op3	p1 | p2					▨		
4									
5									
6									
7	op4	p1					▨		
8									

(a) Partial schedule reservation table

Time	A^{may}		M^{may}	B^{may}	$FW1^{must}$	$FW2^{must}$	$FW3^{must}$
0	op3		op1		▨		
1	op4	op5			▨	▨	
2	op2		op6	op7	▨	▨	▨

(b) Final modulo reservation table

Figure 10: PAMS scheduling of the example in Figure 1

most heavily used resource determines ResMII for IMS.

For PAMS, a similar calculation is done, except that whenever the current resource usage being considered is disjoint from some previously considered usage of that resource, this current usage is combined with that previous usage: the usage count of that resource is not incremented, but the predicate for that previous usage is updated to reflect being combined with the current usage.

3.4 Example of Applying PAMS

To illustrate predicate-aware scheduling, PAMS is applied to the example in Section 2.1 (see Figure 1). The machine model in Table 1 is assumed. Further, the machine is assumed to have a cmpp latency of 2 cycles and a fetch width equal to the number of function units (3). The increased latency for cmpp operations is discussed in Section 4.1.

As each operation is scheduled at some time slot, the appropriate resource is marked at that time slot in both the SRT and MRT. In addition, the predicate expression of the corresponding SRT entry is updated.

Figure 10(a) and Figure 10(b) show the partial (up to *op4*) SRT and the final MRT after the PAMS algorithm is applied to this loop. *op1* is scheduled at cycle 0 of SRT and the MRT (there is no MRT conflict) reserving resource M, and the predicate expression is updated to $p1 \mid p2$ since this operation occurs on both control paths. Here and in the rest of the example, one *FW* resource is reserved for each scheduled operation. Arithmetic operation *op2* is also on both control paths and is data dependent on two cycle *op1*. Therefore, *A* resource is reserved at cycle 2 in both SRT and MRT, with its predicate expression set to $p1 \mid p2$ in the corresponding SRT entry. *op3*'s earliest scheduling time is cycle 2, but it has a resource *A* conflict with the currently scheduled *op2* and is, therefore, scheduled at the next cycle, 3, of the SRT (cycle 0 of MRT), also with predicate expression $p1 \mid p2$.

The earliest scheduling time for *op4* is cycle 5 (since it is dependent on the two-cycle predicate defining operation *op3*), but *op2* uses resource *A* at cycle 2, which causes resource conflicts with cycle 5 since both map to cycle 2 of the MRT (i.e. 2 and 5 are congruent modulo the *II* of the MRT, which is 3). By the same token, *op4* cannot be scheduled at cycle 6 because of the conflict with *op3* currently scheduled at cycle 3. So, *op4* gets scheduled at cycle 7 of the SRT (cycle 1 of the MRT) which has no conflicts. It reserves resource *A* and the predicate expression is set to *p*1 (*op4*'s guarding predicate) to reflect the condition under which the operation will reserve the resource.

The rest of the schedule is not shown in Figure 10(a) but is shown in Figure 10(b). For example, the earliest scheduling time for *op5* is also at cycle 5. But similarly to *op4* it cannot be scheduled until cycle 7. It gets scheduled at cycle 7 of the SRT (cycle 1 of the MRT), which is at the same time with its disjoint operation *op4*. *op5* also reserves the *A* resource and updates the predicate expression to $p1 \mid p2$, which means the resource is now reserved on both control paths: either *op4* will utilize the resource on the False path (*p*1 is True), or *op5* will utilize the resource on the True path (*p*1 is False, *p*2 is True). Next, the operation *op6* is scheduled at cycle 8 of the SRT (cycle 2 of the MRT). Finally, as is customary in IMS (and hence also in PAMS), the region ending branch is replaced with a special control operation (called *brtop* in [15]) that is scheduled within the first II rows of the SRT. *op7* at row 2 of the MRT in Figure 10(b) represents this operation. The result is a successful modulo schedule with II=3.

4. PERFORMANCE EVALUATION

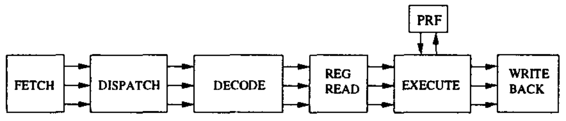
4.1 Predicate-Aware Architecture

As we discussed in Section 3, the generic predicate-aware architecture has two categories of resources: may-use and must-use. Every resource used after the value of the guarding predicate becomes known can be may-use, i.e., it can be reserved by disjoint operations at a time. All the remaining resources are must-use and can only be reserved by a single operation in a given time. Therefore, the earlier the predicates are read, the more resources can be may-use, which can lead to shorter schedules. On the other hand, accessing the predicate register file earlier in the processor pipeline increases the latency of the predicate defining operation. This can be problematic if many of predicate defining operations lie on the critical path of the application.

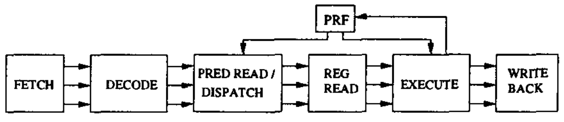
In our experiments, we evaluate the baseline pipeline datapath shown in Figure 11(a). This architecture is similar to the TI 'C6x architecture [6], except that the unified register read and execution stages are separated. The baseline processor pipeline has 6 stages: **fetch, dispatch, decode, register read, execute** and **write back.**The predicates are only read during the execution stage. Thus, resources in the execute stage and the preceding stages are must-use. Only the resources in the write-back stage are may-use.

In order to make the baseline pipeline datapath predicate-aware, four issues must be addressed. First, nullification should be performed earlier in the pipeline to make more may-use resources available. Second, the disjoint operations should be easily identifiable. Third, the cmpp latency should be kept as small as possible. Fourth, the pipeline complexity should not be increased substantially to compromise the cycle time. To this end, we make two main changes in the baseline pipeline to make it predicate-aware as shown in Figure 11(b).

The first change is to move predicate register file (PRF) read to the dispatch stage. This allows nullification to occur at the end of the dispatch stage. As a result, all the resources in subsequent stages (general / floating-point register ports, function units, etc.) are may-use.



(a) Baseline Machine



(b) Predicate-aware Machine

Figure 11: Baseline vs. Predicate-aware machine models

During the dispatch stage, the PRF is accessed early in the cycle to read predicates for all operations. Then, the dispatch logic nullifies the disjoint operations guarded under False and assigns the rest of the operations to their corresponding function units. To identify the disjoint operations, each operation is augmented with a **mutex** bit. If the mutex bit is '0', the operation is not disjoint from any previous operations. If the mutex bit is '1', the operation is disjoint from a previous operation.

The second change, is to reverse the order of the decode and dispatch stages. This change is made to delay predicate read by one stage, so as to reduce cmpp latency by one cycle. The result of this reordering is that decode occurs before dispatch. As a result, the complexity of the decode logic is increased. In the worst case, FW general purpose decoders, one per operation in the instruction word, are required. In the alternative, where decode follows dispatch, less expensive special purpose decoders can be used.

Notice that in the predicate-aware machine, the 1-bit wide PRF can be simultaneously accessed in two pipeline stages: predicate read / dispatch and execute. Thus, twice as many PRF read ports are required. If this poses a problem, the PRF could be replicated in each of these two stages.

The primary negative of our design is that since predicate read has been moved earlier in the pipeline, the distance between cmpp operations and their disjoint consumers must be increased. Without any further changes, the cmpp operation latency is two cycles. This assumes that full forwarding of predicate values is possible. However, since the predicates are accessed early in the dispatch stage and computed later in the execute stage, forwarding to the dispatch stage is not feasible. Without such forwarding, the predicate-aware scheduler needs to separate cmpp operations from their disjoint consumers by at least three cycles.

Finally, it is possible to have more pipeline stages between predicate read / dispatch and execute as the result of increased processor frequency (for example, register read or comparator logic may be split into multiple stages). As the results in Section 4.3 indicate, higher cmpp latency will degrade the performance of PALS because cmpp operations are often on the critical paths of acyclic regions. However, it will not have a significant impact on PAMS because cmpp operations are rarely on the critical paths of cyclic regions.

4.2 Evaluation Methodology

We use an existing VLIW compiler technology, **Trimaran** [19], to evaluate the effectiveness of our technique. This compiler system is capable of performing if-conversion with hyperblock formation [12], scalar and modulo scheduling, and predicate analysis, among other back-end optimizations. We implemented the bulk of our optimizations within the resource management module of **ELCOR** (Trimaran's back-end compiler). We also use the Predicate Query system [8] to analyze predicated code and construct the relationship among predicates in particular the disjointness relationship.

We use the notation (F,E,I,FP,M,B,C) to represent the processor in this study. **F** is fetch width, **E** - execution width, **I** - number integer units, **FP** - number of floating-point units, **M** - number of memory units, **B** - number of branch units, and **C** - latency of the predicate defining operation (cmpp). We use two base processors in our study: (4,4,2,1,1,1,1) and (6,6,4,2,1,1,1) called $P_{ba}(4)$ and $P_{ba}(6)$, respectively.

Each baseline processor $P_{ba}(i)$ is compared with three corresponding predicate-aware processors $P_{pa}(i,1)$, $P_{pa}(i,2)$ and $P_{pa}(i,3)$ with the same number of resources as baseline processor, but cmpp latency of one, two and three, respectively. Although maintaining a cmpp latency of one cycle in a predicate-aware architecture is almost impossible, the results are indicative of the schedule upper bound.

We evaluated the set of 17 MediaBench [11] applications, applying predicate-aware scheduling optimizations to the entire code. Clearly, the predicate-aware scheduling can only benefit if-converted regions of code that contain at least one *if-then-else* clause (we call these regions **pa-ready**), and will be ineffective for other code regions due to their lack of disjoint operations. We assume that those other regions will execute with predicate-aware support turned off. In addition, in this study we assume that all predicates in the predicate-aware scheduled region are read early in the predicate read / dispatch pipeline stage regardless of whether the operations guarded under these predicates share may-use resources. This assumption unnecessarily increases the latency of all cmpp operations in the region. In our future work, only those cmpp operations that define predicates for the operation in a group of disjoint operations will have their latency increased.

4.3 Evaluation Results

The goal of the predicate-aware scheduler is to reduce the length of the resource constrained baseline schedule on the baseline machine of fixed width. The predicate-aware scheduler takes advantage of the gap between the upper-bound defined by the length of the baseline schedule and the lower bound which is the maximum of the resource-constrained and latency-constrained schedule lengths of the predicate-aware processor. The resource-constrained schedule length is computed ignoring all data dependencies. The latency-constrained schedule length is the length of critical path. The gap between these lower and upper bounds constitutes the headroom for the predicate-aware scheduler.

Table 2 shows an estimate of the predicate-aware scheduler headroom on acyclic (columns 2-7) and cyclic (columns 8-13) pa-ready regions. The data presented is averaged over all benchmarks. Column 1 shows the two machine models. Column 2 shows the length

Machine	Acyclic Scheduling						Cyclic Scheduling					
	SL_{ba}	CPL1	CPL2	CPL3	$ResMSL_{ba}$	$ResMSL_{pa}$	II_{ba}	RecMII1	RecMII2	RecMII3	$ResMII_{ba}$	$ResMII_{pa}$
P(4)	29.44	22.37	25.44	28.75	23.65	19.73	29.36	3.73	4.34	4.94	29.26	23.78
P(6)	24.87	22.37	25.44	28.75	12.08	11.29	15.55	3.74	4.34	4.94	14.97	13.13

Table 2: Data to estimate scheduling headroom for predicate-aware scheduler

	PALS								PAMS							
	P(4)				P(6)				P(4)				P(6)			
Benchmark	%Time	SP1	SP2	SP3	%Time	SP1	SP2	SP3	%Time	SP1	SP2	SP3	%Time	SP1	SP2	SP3
cjpeg	77.96	1.07	1.00	1.00	80.78	1.00	1.00	1.00	2.02	1.29	1.29	1.27	1.43	1.23	1.23	1.21
djpeg	25.13	1.04	1.00	1.00	33.33	1.01	1.00	1.00	37.37	1.16	1.09	1.09	30.31	1.05	1.05	1.05
epic	4.82	1.03	1.01	1.06	4.74	1.01	1.01	1.01	0.86	1.20	1.20	1.20	0.62	1.00	1.00	1.00
g721encode	38.18	1.13	1.09	1.04	42.84	1.04	1.01	1.01	39.7	1.18	1.16	1.25	33.7	1.07	1.12	1.08
g721decode	41.62	1.12	1.08	1.04	45.95	1.05	1.01	1.01	39.98	1.18	1.16	1.25	33.5	1.07	1.12	1.08
gsmdecode	4.52	1.05	1.00	1.00	7.47	1.00	1.00	1.00	89.97	1.07	1.09	1.09	83.76	1.06	1.05	1.05
mesamipmap	22.1	1.07	1.01	1.00	25.31	1.02	1.00	1.00	26.63	1.34	1.31	1.34	21.46	1.15	1.16	1.15
mpeg2dec	23.81	1.20	1.11	1.02	24.94	1.05	1.00	1.00	32.31	1.10	1.10	1.09	22.83	1.07	1.07	1.07
mpeg2enc	18.69	1.02	1.01	1.00	26.26	1.00	1.00	1.00	67.76	1.24	1.07	1.24	55.01	1.11	1.11	1.07
pegwitdec	42.52	1.01	1.00	1.00	51.34	1.00	1.00	1.00	15.03	1.04	1.04	1.03	12.03	1.06	1.06	1.06
pegwitenc	46.99	1.01	1.01	1.00	55.6	1.00	1.00	1.00	16.07	1.04	1.04	1.03	12.73	1.06	1.06	1.06
rasta	23.27	1.01	1.01	1.00	27.06	1.00	1.00	1.00	2.09	1.14	1.14	1.14	1.29	1.00	1.00	1.00
rawcaudio	0.10	1.18	1.08	1.00	0.11	1.00	1.00	1.00	99.81	1.04	1.00	1.00	99.8	1.00	1.00	1.00
rawdaudio	0.07	1.00	1.00	1.00	0.12	1.00	1.00	1.00	99.84	1.11	1.11	1.11	99.71	1.11	1.00	1.00
unepic	45.45	1.01	1.00	1.00	54.28	1.00	1.00	1.00	6.47	1.20	1.20	1.20	5.23	1.00	1.00	1.00
ghostscript	67.89	1.13	1.05	1.01	73.05	1.04	1.00	1.00	14.73	1.32	1.31	1.28	9.94	1.15	1.15	1.15
average	28.57	1.07	1.03	1.01	32.76	1.01	1.00	1.00	40.32	1.18	1.16	1.18	36.24	1.08	1.08	1.07

Table 3: Speedup breakdown

of the baseline acyclic schedule. Columns 3-5 show the critical path length for cmpp latencies 1, 2 and 3, respectively. Column 6 and 7 show the resource-constrained schedule length for baseline processor, and the predicate-aware processor, respectively. Since the resource-constrained schedule ignores all data dependencies, cmpp latency has no effect here. Columns 8-13 show similar data for the cyclic regions; resource-constrained schedule length is defined by ResMII, and latency-constrained schedule length is defined by RecMII for cyclic regions. Relative to the pa-read acyclic region schedule length on the baseline 4(6)-wide machine (with a cmpp latency of 1 cycle), we see from Table 2 that on average the critical path length for cmpp latencies of 1, 2 and 3 cycles respectively is 24%(10.1%), 13.6% (-2.6%) and 2.3% (-15.6%) shorter. Thus, as the latency of the cmpp operation increases, the latency-constrained lower bound approaches closely and eventually exceeds the length of the baseline schedule. Therefore, cmpps are often on the critical path of the acyclic region leaving little headroom for PALS. On the other hand, cmpp latency is not limiting in cyclic regions as seen by the fact that $ResMII_{pa}$ is much larger than *RecMII1*, *RecMII2*, and *RecMII3*.

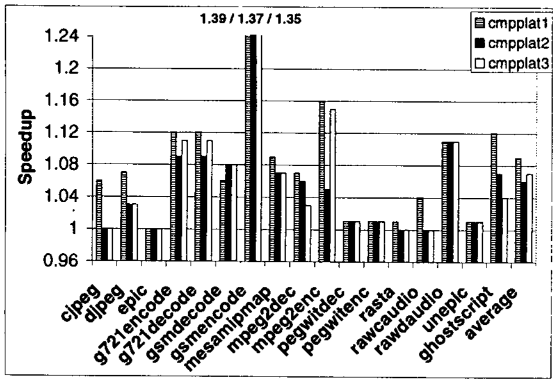
Table 3 (columns 2-9) shows the individual speedups achieved by predicate-aware scheduling on acyclic regions for each processor model. Column 1 lists all the benchmarks. Column 2 shows the percent of execution time on a 4-wide baseline that the application spends in pa-ready acyclic regions. Columns 3-5 show the speedup achieved by each predicate-aware 4-wide machine over the 4-wide baseline machine for all pa-ready acyclic regions. Columns 6-9 show similar data for the 6-wide machines. Columns 10-17 show the corresponding data for the cyclic regions. The last row shows the arithmetic mean over all benchmarks.

As stated earlier, the schedule length of acyclic regions is constrained by the latencies of the operations, with the cmpp operation generally being on the critical path. Therefore, the performance of PALS decreases with increased cmpp latency for both processor models, resulting in very small speedup (3% and 1%) for cmpp latency of 2 and 3 cycles for 4-wide machines and no speedup for 6-wide machines with the same latencies. Notice that speedup never falls below 1. This is due to the fact that the predicate-aware scheduling is only applied to regions that can benefit from this technique; otherwise, the baseline schedule is used for these regions.

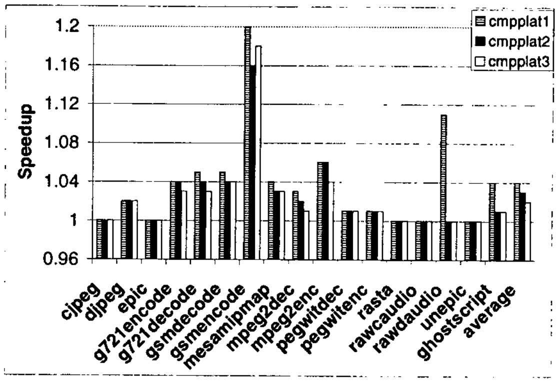
On the other hand, as we go to a wider machine, while keeping the latency of the predicate-aware machine fixed, the latency-constrained lower bound remains the same. However, the length of the baseline schedule (and the resource-constrained upper bound) decreases, approaching (and even finally falling below the latency-constrained lower bound, as resources (FW and/or other resources) are added. This also reduces the headroom for the predicate-aware scheduler. This explains the degradation in PALS speedup as we go from a 4-wide to a 6-wide baseline machine for fixed cmpp latency.

In the case of cyclic regions, the PAMS lower-bound is determined by the resource-constrained schedule length of the predicate-aware processor ($ResMII_{pa}$, see Table 2). Latency-constrained schedule length ($RecMII$) is not a limiting factor for either 4-wide or 6-wide machine models. As Table 2 shows, $RecMII$ for, even for a cmpp latency of 3, is much smaller than $ResMII_{pa}$. Therefore, as columns 10-17 of Table 3 show, PAMS achieves substantial speedups for all cmpp latencies (18%, 16%, 18%). Note that PAMS performance in cyclic regions is not very sensitive to cmpp latency.

We notice that in some cases, the speedup for higher cmpp latency is greater than the speedup for smaller cmpp latency. For example, for g721encode, PAMS achieves the speedup of 1.16 on a 4-wide predicate-aware processor with a cmpp latency of 2 and a speedup of 1.25 on the processor with a cmpp latency of 3 cycles. This happens due to the fact that the performance of a modulo scheduled loop is also limited by the loop trip count and the size of the epilogue. As cmpp latency increases, PAMS fails to find a schedule with the same II (due to a register limitations). It then schedules the loop with a higher II, but generally, with a shorter



(a) Speedup of $P_{pa}(4, 1/2/3)$ over $P_{ba}(4)$



(b) Speedup of $P_{pa}(6, 1/2/3)$ over $P_{ba}(6)$

Figure 12: Overall Speedup

epilogue, so that the overall performance of this loop with a small trip count can be better than with a smaller cmpp latency, resulting in higher speedup over the baseline processor.

Figure 12 shows the overall speedup due to predicate-aware scheduling. Figure 12(a) shows the speedup of each of the three predicate-aware processors $P_{pa}(4, 1)$, $P_{pa}(4, 2)$, $P_{pa}(4, 3)$ over the corresponding baseline processor $P_{ba}(4)$ for each application. Figure 12(b) shows similar data for the baseline processor $P_{ba}(6)$. The average speedups achieved over all applications is 9%, 6% and 7% for 4-wide machine and 4%, 3% and 2% for 6-wide machine with cmpp latencies of one, two and three cycles, respectively. For cmpp latencies of 2 and 3, most of the performance improvement comes from PAMS. The speedup achieved on the entire application is smaller than the speedup achieved on pa-ready cyclic regions alone, since as Table 2 shows, these regions constitute on average only 36% of the total application baseline execution time.

5. RELATED WORK

Predication is a widely used technique. A number of existing VLIW machines, both embedded and general-purpose, have hardware support for predication. They do vary as to what stage in the pipeline predicates are read and when operations are nullified. Regardless of these differences, no architecture that we are aware of allows disjoint operations to reserve the same resource in the same clock cycle, as in our predicate-aware technique. However, these architectures can be extended to support predicate-aware schedules.

Texas Instrument's TMS320C6000 [6] accesses its predicate register file (PRF) early in the execution stage before accessing the register file. Ideally, this could provide an opportunity to select among several disjoint operations, and only let those operations guarded under a True predicate proceed with execution. Nevertheless, this is not done. A predicate-aware TI 'C6x would require disjoint operations to be scheduled at least two cycles later than its corresponding cmpp operation (vs. three, as assumed in our processor model, see Section 4.1). This is possible since the TI 'C6x combines register read and execute in the same pipeline stage.

Intel's Itanium [7] [16] processor accesses its PRF in the execution stage of the pipeline. Itanium also accesses the PRF in the register read pipeline stage in parallel with general registers. This is done in order to nullify an operation waiting on the result from a long latency operation which is currently being executed: if the read predicate is False, the consuming operation is squashed and execution can continue if there are no other hazards. However, the impact of this optimization on overall performance is small. Since the PRF is accessed in parallel with the general registers in the register read stage, distinct register ports must be reserved by all simultaneously scheduled disjoint operations.

On the compiler side, a number of techniques were proposed in the past to improve modulo schedules of loops with internal control flow. Hierarchical reduction [10] collapses all conditional constructs into a single operation, modulo schedules the resulting straight-line code, and then regenerates all conditional constructs. All Path Pipelining [18] pipelines each path separately using a software pipelining technique for straight-line loops and then merges the pipeline kernels of the paths. Modulo Scheduling with Multiple Initiation Intervals [22] schedules if-converted code so that control paths with higher execution frequencies have shorter IIs than paths with lower frequencies. Predicated operations are used to execute the correct operations and loop-back branch based upon which path is actually executed dynamically. The advantage of these techniques is that they do not assume any special hardware support in the form of predication, but as a result they suffer from significant code growth in the loop kernel.

Warter's Enhanced Modulo Scheduling [21] initially modulo schedules predicated code in which disjoint operations are allowed to share resources. This is similar to our technique as well as to hierarchical reduction in that its sharing enables the resource requirements of operations from disjoint paths to be the union rather than the sum of the requirements of each individual operation. However, this scheme assumes no hardware support for predication. Therefore, in the next step the control flow is regenerated from the intermediate schedule to obtain the final pipelined schedule. The intermediate schedule is then discarded; the idea of executing the shared-resource schedule with predicate-aware hardware is not explored in their work.

6. CONCLUSIONS

We have proposed and evaluated a new predicate-aware scheduling technique which can achieve better schedules on both acyclic and cyclic predicated code regions by reducing wasted resources in VLIW processors with predicated execution. This technique enables the compiler to schedule operations on the same processor resource in the same cycle as long as two conditions hold: (i) the compiler can prove that the operations are guarded by disjoint predicates, and (ii) the processor has nullified all operations guarded under False before using the may-use resource.

The predicate aware modulo and scalar schedulers have been implemented and evaluated on the suite of Mediabench applications. The overall results show an average performance gain of 7% and 2% for 4-issue and 6-issue VLIW processors, respectively. These gains are primarily due to loops where resources, and not dependences, often limit performance. For loops, predicate-aware scheduling achieves an average gain of 18% and 7% for the same processors.

7. ACKNOWLEDGMENTS

We thank Michael Chu, Joel Emer, Trygve Fossum, and the anonymous referees for their advice and helpful comments. This research was supported in part by the DARPA/MARCO C2S2 Research Center. Mikhail Smelyanskiy was supported in part by an Intel Fellowship.

8. REFERENCES

[1] T. Adam, K. Chandy, and J. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACMr*, 17(12):685–690, Dec. 1974.

[2] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 143–180, Jan. 1983.

[3] E. Davidson, L. Shar, A. Thomas, and J. Patel. Effective control for pipelined computers. In *Proceedings of COMPCON*, pages 181–184, Feb. 1975.

[4] A. E. Eichenberger and E. S. Davidson. Register allocation for predicated code. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 180–191, November 1995.

[5] D. M. Gillies, R. D. Ju, R. C. Johnson, and M. S. Schlansker. Global predicate analysis and its application to register allocation. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 114–125, December 1996.

[6] T. Instruments. TMS320C62x/67x CPU and Instruction Set Reference Guide. http://www-s.ti.com/sc/psheets/spru189f/spru189f.pdf, 1998.

[7] Intel. Itanium Processor Microarchitecture Reference for Software Optimization. ftp://download.intel.com/design/Itanium/Downloads/, Nov. 2001.

[8] R. Johnson and M. Schlansker. Analysis techniques for predicated code. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, Dec. 1996.

[9] V. Kathail, M. Schlansker, and B. R. Rau. *HPL PlayDoh Architecture Specifications: Version 1.0.* HP Laboratories Technical Report HPL-93-80, 1994.

[10] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–327, 1988.

[11] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, Dec. 1997.

[12] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, December 1992.

[13] J. Park and M. Schlansker. *On predicated execution.* HP Laboratories Technical Report HPL-91-58, 1991.

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

[15] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. Code Generation Schema for Modulo Scheduled Loops. In *Proceedings of the 25th Annual International Symposium on Michorarchitecture*, December 1992.

[16] H. Sharangpani and K. Arora. Itanium Processor Microarchitecure. *IEEE Micro*, Vol. 20, No. 5:24–43, September/October 2000.

[17] J. W. Sias, D. I. August, and W. W. Hwu. Accurate and efficient predicate analysis with binary decision diagrams. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 112–123, December 2000.

[18] M. Stoodley and C.G.Lee. Software pipelining loops with conditional branches. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 262–273, Dec. 1996.

[19] The Trimaran System. www.trimaran.org, 1999.

[20] R. Towle. *Control and Data Dependence for Program Transformations.* PhD Dissertation, The University of Illinois, 1976.

[21] N. Warter, G. Haab, K. Subramanian, and J. Bockhaus. Enhanced modulo scheduling for loops with conditional branches. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 170–179, 1992.

[22] N. Warter and N. Partamian. Modulo scheduling with multiple initiation intervals. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 111–118, 1995.