# Probabilistic Predicate-Aware Modulo Scheduling

Mikhail Smelyanskiy *  Scott Mahlke  Edward S. Davidson

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

## ABSTRACT

Predicated execution enables the removal of branches by converting segments of branching code into sequences of conditional operations. An important side effect of this transformation is that the compiler must unconditionally assign resources to predicated operations. However, a resource is only put to productive use when the predicate associated with an operation evaluates to True. To reduce this superfluous commitment of resources, we propose probabilistic predicate-aware scheduling to assign multiple operations to the same resource at the same time, thereby over-subscribing its use. Assignment is performed in a probabilistic manner using a combination of predicate profile information and predicate analysis aimed at maximizing the benefits of over-subscription in view of the expected degree of conflict. Conflicts occur when two or more operations assigned to the same resource have their predicates evaluate to True. A predicate-aware VLIW processor pipeline detects such conflicts, recovers, and correctly executes the conflicting operations. By increasing the effective throughput of a fixed set of resources, probabilistic predicate-aware scheduling provided an average of 20% performance gain in our evaluations on a 4-issue processor, and 8% gain on a 6-issue 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 or 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 [26, 2, 15]. 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 if-converted branches, 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 over-saturation of the processor resources [14].

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. The only required resources are those up to the point of determining that its predicate is False; 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 reduces the number of must-use resources.

To overcome the problem of superfluous resource utilization by nullified operations, we propose **probabilistic predicate aware scheduling**. In this work we apply this technique to modulo scheduled loop regions which are generally resource constrained. The central idea of PPAMS (probabilistic predicate-aware modulo scheduling) is to allow over-subscription of may-use resources wherein multiple operations are allowed to reserve the same resource at the same time. As a consequence, it is possible for dynamic over-subscription of resources to take place so that two or more resource-sharing operations will have their predicates evaluate to True at runtime, resulting in a resource conflict. PPAMS is a generalization of deterministic predicate-aware modulo scheduling (DPAMS) [23]. DPAMS allows operations to share the same resource when their predicates are provably disjoint, i.e. at most one will evaluate to True at run-time. By allowing conflicts to occur, PPAMS finds many more combinable operations than DPAMS. Thus, PPAMS significantly increases the utilization of may-use resources and leads to improved processor performance. A secondary

*Mikhail Smelyanskiy is now with the System Technology Lab at Intel Corporation.

benefit of PPAMS is that with resource constraints lessened, more aggressive if-conversion can be applied to extract further benefit from branch elimination.

The consequence of probabilistic over-subscription is that it is possible for dynamic over-subscription of resources to take place. In this situation, two or more operations will have their predicates evaluate to True at run time resulting in resource conflict. To deal with this problem, PPAMS estimates resource conflicts and makes scheduling decisions to maximize the benefits of over-subscription. Predicates are probabilistically analyzed using a combination of predicate profile information and predicate analysis [11]. Predicate profile information provides statistics on the expected number of times a predicate will evaluate to True. Predicate analysis computes superset/subset and disjointness relations among predicates to identify when two or more predicates are guaranteed to definititely or never conflict. Probabilistic analysis is used to identify profitable opportunities for resource over-subscription. The scheduler takes advantage of these opportunities when they lead to a tighter schedule.

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. This may be unacceptable, especially for cost- or power-sensitive environments. Predicate-aware scheduling increases the utilization of existing processor resources and therefore increases application performance with a fixed set of resources. In this paper, we present the necessary hardware and software extensions to support PPAMS.

## 2. BACKGROUND AND MOTIVATION

Iterative Modulo Scheduling (IMS) [16] is a software pipelining technique that interleaves successive iterations of a loop. 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 longest cycle in the dependence graph.

To satisfy scheduling constraints, IMS uses two data structures known as the Schedule Reservation Table (SRT) and the Modulo Reservation Table (MRT). The actual realization of these two tables is implementation dependent. Conceptually they work as follows. The SRT displays the schedule for one iteration; it records the use of a particular resource at a particular time by each specific operation [6, 16]. Scheduling at that time is permitted only if the resource usage does not result in a resource conflict, and no latency constraints of prior operations on which the operation being scheduled depends are violated. The MRT is used by IMS to track the modulo constraint which 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 [26, 2, 15] is applied to broaden the class of loops that can be modulo scheduled. The performance of IMS is usually resource constrained, i.e. $ResMII$ is larger than $RecMII$. Predication, which merges together the operations along multiple control paths, further increases the resource constraints by requiring an operation to reserve its resource unconditionally, regardless of whether its path is actually taken.

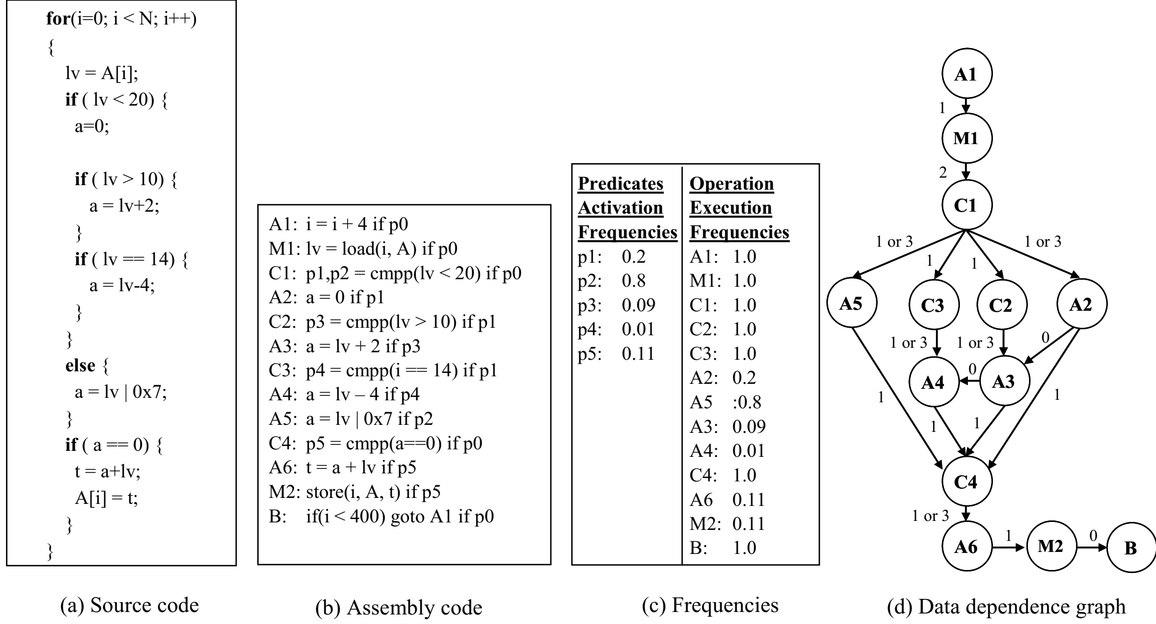
| Function Unit | Operations | Mnemonics | Latency |
|---|---|---|---|
| ALU (A) | Add<br>Or | **+**<br>\| | 1<br>1 |
| | Predicate Compare | **cmpp** | 1 or 3 |
| Memory (M) | Load<br>Store | **ld**<br>**st** | 2<br>1 |
| Branch (B) | Branch on Condition | **br** | 1 |

**Table 1: Description of a sample processor with a fetch/execute width of 3 operations**

To illustrate the application of conventional IMS along with the potential benefits of PPAMS, we consider a simple code example and processor model. The example processor, which can fetch and execute up to three operations per cycle, has three fully pipelined function units as detailed in Table 1. The table lists two latencies, 1 and 3, for the predicate-defining operation, cmpp, because, as discussed in Section 3, to support predicate-aware scheduling, the cmpp latency must be increased by at least one cycle; in this simple example, the cmpp latency is increased from 1 to 3 cycles.

Figure 1(b) shows the assembly code of an example if-converted loop whose source code is shown in Figure 1(a). The predicate-defining operation *cmpp*, which replaces the branch in the original control statement, sets the predicate(s). For example, $C1$ sets two disjoint predicates, $p1$ and its complement $p2$. Operations that were on either the *then* or *else* paths are now guarded under the corresponding predicates. For example, $A2$ is guarded under $p1$ and $A5$ is guarded under $p2$. Figure 1(c) lists each predicate used by the code along with its activation frequency, which is the fraction of (profiled) loop iterations in which the predicate evaluates to True. It also shows the execution frequency of each operation which is equal to the activation frequency of its guarding predicate. Note that unconditional cmpp operations [20] (which we assume in this example) always set the value of the destination predicate regardless of their guarding predicate, thus their execution frequency is always 1.0. Figure 1(d) shows the data dependence graph for the code in Figure 1(b). Each node shows the corresponding operation, and its outbound arcs are labeled with the operation's latency. Again, as shown in Table 1, each cmpp operation's outbound arcs are labeled with two latencies, 1 and 3. Note that the edges in the graph are all flow dependences with the exception of the edge from $M2$ to $B$ which is a control dependence.

The application of IMS to the example results in the II=10 schedule presented in the MRT shown in Figure 2(a). The notation used for each operation is *iter:opname*, where *iter* is the iteration to which the operation belongs relative to the current ($n_{th}$) iteration. Since each of ten ALU operations (*A1, A2, A3, A4, A5, A6,C1, C2, C3, C4*) must reserve the ALU resource at a different cycle to avoid conflict, ResMII=10 and this schedule is optimal (RecMII=1 for this loop). Predicated operations $A2$, $A3$, $A4$, $A5$ and $A6$ are executed conditionally but reserve the ALU unconditionally. Consequently, an ALU cycle is wasted every time the guarding predicate




(a) Source code (b) Assembly code (c) Frequencies (d) Data dependence graph

**Figure 1: Example code segment**

of its assigned operation is False. We can increase ALU utilization and hence reduce the schedule length if we combine two or more of these predicated operations to share the ALU in the same cycle.

DPAMS guarantees that no conflicts will occur by combining only disjoint operations to share the same resource [23]. A DPAMS schedule is shown in Figure 2(b). Disjoint operations *A2* and *A5* are scheduled in the same slot of the MRT. As a result, the ALU is always utilized at Time 8, and the achieved schedule length is 9 cycles, an 11% performance improvement over the 10 cycle baseline schedule in Figure 2(a). Note that the ALU is still underutilized at Times 1, 2 and 5 by operations $A3$, $A4$ and $A6$, but none of these operations can be combined by DPAMS.

PPAMS achieves a 6 cycle *static* schedule length ($II_{static}$) by also combining the operations $A3$ and $A4$ from the same iteration and $C4$ from the previous iteration to share the ALU at Time 2, and $C3$ with $A6$ from two iterations earlier to share the ALU at Time 5, as shown in Figure 2(c). Note that neither of these additional combinations is disjoint. Because $C4$ always requires the ALU resource, each time at least one of operations $A3$ and $A4$ executes there will be at least one conflict, causing some conflict recovery delay, $II_{CflDelay}$. Operations $C3$ and $A6$ will also conflict and delay the execution whenever $A6$ executes. If $A3$, $A4$ and $A6$ always execute, there will be at least 2 cycles of delay at Time 2 and 1 cycle of delay at Time 5 of the MRT, resulting in no improvement over DPAMS. But, we expect only 0.42 cycles of penalty on average as explained in Sections 4.1 and 4.3 and shown in the last row of the MRT in Figure 2(c).

PPAMS thus achieves a 6.42 cycle *expected* schedule length ($II_{expected} = II_{static} + II_{CflDelay}$): a 40% performance improvement over DPAMS, and 55% over the baseline.

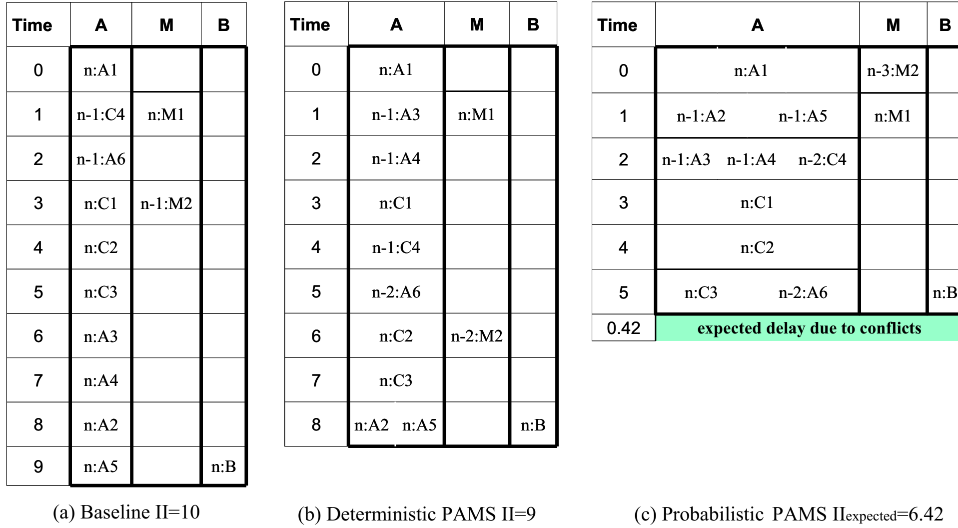
This simple example shows that allowing predicated operations to reserve the same resource in the same time-slot can reduce the resource requirements and static schedule length for a predicated code segment, but will cause conflicts at run time whenever two or more operations that are combined together have their predicates evaluate to True in the same cycle. The goal of PPAMS is to decrease the $II_{expected}$ schedule length by maximizing the degree of useful overlap, subject to controlling the expected degree of conflict.

# 3. PROBABILISTIC PREDICATE-AWARE VLIW PROCESSOR ARCHITECTURE

In this section, the probabilistic predicate-aware (PPA) architecture is described. 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 is may-use and can be reserved by several predicated operations at the same time. All the remaining resources are must-use and can only be reserved by a single operation at a time. The categorization rule is that every resource that is after the predicate read and operation nullification point in the pipeline is may-use. Reading predicates and nullifying operations earlier allows more resources to be may-use, which leads to shorter schedules. However accessing the predicate register file earlier in the processor pipeline increases the latency of the predicate defining operation which can be problematic if many of the predicate defining operations lie on the critical path of the application.

Our baseline architecture, shown in Figure 3(a), is similar to the TI ‘C6x architecture [10], except that its unified register read and execution stage is separated here. The baseline processor pipeline has 6 stages: fetch, dispatch, decode, register read, execute and write back.The predicates are read only 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 convert the baseline pipeline datapath into a PPA datapath, four issues must be addressed. First, nullification should be performed earlier in the pipeline to make more may-use resources available. Second, the cmpp latency should be kept as small as possible. Third, a conflict resolution mechanism is required to de-



| Time | A | M | B |
|---|---|---|---|
| 0 | n:A1 | | |
| 1 | n-1:C4 | n:M1 | |
| 2 | n-1:A6 | | |
| 3 | n:C1 | n-1:M2 | |
| 4 | n:C2 | | |
| 5 | n:C3 | | |
| 6 | n:A3 | | |
| 7 | n:A4 | | |
| 8 | n:A2 | | |
| 9 | n:A5 | | n:B |

(a) Baseline II=10

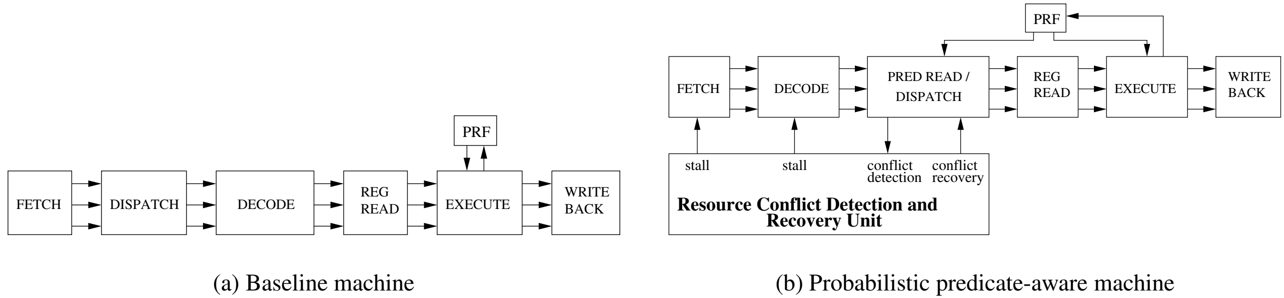
| Time | A | M | B |
|---|---|---|---|
| 0 | n:A1 | | |
| 1 | n-1:A3 | n:M1 | |
| 2 | n-1:A4 | | |
| 3 | n:C1 | | |
| 4 | n-1:C4 | | |
| 5 | n-2:A6 | | |
| 6 | n:C2 | n-2:M2 | |
| 7 | n:C3 | | |
| 8 | n:A2 n:A5 | | n:B |

(b) Deterministic PAMS II=9

| Time | A | M | B |
|---|---|---|---|
| 0 | n:A1 | n-3:M2 | |
| 1 | n-1:A2 n-1:A5 | n:M1 | |
| 2 | n-1:A3 n-1:A4 n-2:C4 | | |
| 3 | n:C1 | | |
| 4 | n:C2 | | |
| 5 | n:C3 n-2:A6 | | n:B |
| 0.42 | expected delay due to conflicts | | |

(c) Probabilistic PAMS $II_{expected}$=6.42

**Figure 2: Three schedules for the example code segment**




(a) Baseline machine

(b) Probabilistic predicate-aware machine

**Figure 3: Baseline and predicate-aware pipeline organizations**

tect and recover from conflicts. Fourth, to avoid compromising the cycle time, the pipeline complexity should not be increased substantially. To this end, we make three main changes in the baseline pipeline to make it predicate-aware, as shown in Figure 3(b).

**Change 1:** The first change is to move the 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. During the dispatch stage, the PRF is accessed early in the cycle to read predicates for all the operations. Then, the dispatch logic nullifies those operations guarded under False and assigns the rest of the operations to their corresponding function units, provided there is no conflict.

**Change 2:** In the baseline processor, the latency of the cmpp operation is 1 cycle. The first proposed change above, however, increases it to 4 cycles. Our second change is to reverse the order of the decode and dispatch stages, thereby delaying the predicate read by one stage, which reduces the cmpp latency to 3 cycles. However, since decode now occurs before dispatch, the complexity of the decode logic is increased somewhat. In the worst case, $FW$ (fetch width) general purpose decoders, one per operation in the instruction word, are required. In the alternative, where decode follows dispatch, it might be possible to use more specialized and hence less expensive decoders.

**Change 3:** The third and most significant change is to add a **Resource Conflict Detection and Recovery Unit**. During the predicate read and dispatch stage, if more than one of the operations scheduled to execute on a particular function unit has its predicate set to True, the conflict is detected and the *conflict* signal is set. As a result the *stall* signal is sent to the fetch and decode stages, and a recovery process is executed.

A recovery process must address two important issues: first is how to dispatch the conflicting operations during recovery and second is when to start dispatching these operations.

To address the **first** issue, we note that there are two alternative schemes to dispatch operations to recover from such a conflict. In the first scheme, those operations in the dispatch stage that have True predicates are dispatched in parallel to the function units that they were originally assigned to, and one such operation is dispatched to each function unit in each cycle until there are no such operations left for that function unit. This process continues until the entire instruction word is dispatched. This scheme results in a simpler design, but a longer stall time than the second scheme. In the second alternative scheme, an operation may be dispatched to any available function unit that can serve it (rather than only to the function unit it was originally assigned to, as in the first scheme). This scheme will reduce the stall time, but its design is somewhat more complex. In our experiments we use the former, simpler re-

covery scheme. Other more complex schemes could also be considered, but are beyond the scope of this paper.

The **second** issue is whether the recovery itself may begin immediately, in the same cycle in which a conflict is detected, or requires additional time to initiate. To model the effect on performance, a machine parameter called the conflict detection and recovery unit latency ($CDRL$) is used in the experiments. We investigate $CDRL$ values of 0 cycles, wherein the first conflicting operation is dispatched in the conflict detection cycle itself, and 1 cycle, wherein it is dispatched one cycle later.

To address the issue of delay and complexity of the conflict detection and recovery unit, we note that there already are reasons why a dispatcher may be unable to dispatch an operation or group of operations in a particular cycle, e.g. the input buffer of a required execution unit may be full. The conflict described above is just one more simple reason, and the 'stall' or interlock mechanism it needs to invoke is no more complex than the mechanisms that already exist in today's microprocessors for this purpose. The stall signal is produced by simple combinational logic; as soon as the predicates are read early in the cycle, this logic generates the stall signal if more than one predicate that has reserved the same resource has a True value. This 'stall' signal could simply be added as an additional trigger to any of those existing mechanisms to effect a stall or fetch/decode restart, as appropriate.

The main potential performance degradation factor of our design is that since predicate read has been moved earlier in the pipeline, the schedule distance between a cmpp operation and its consumers must be increased, possibly increasing the length of the critical path in the code region, and thus degrading the performance. As shown in Section 5, modulo scheduled loops in the benchmarks that we examined are constrained by resources, not by latencies. Therefore, increasing the cmpp latency has only a small overall impact on the performance of PPAMS.

An interesting consequence of our design is that it is possible to selectively increase the cmpp latency, so as to restrict the increase in critical path length. Only some operations will see an increased cmpp latency; all other operations will see a cmpp latency of 1, which means that they will execute unconditionally on their resources (i.e. their execution frequency is increased to 1.0). To allow both kinds of operations to read their predicates in the same cycle, the 1-bit wide PRF in Figure 3(b) can be simultaneously accessed from the execute stage, as well as the predicate read and dispatch stage. Thus, twice as many PRF read ports are required, but they are only 1-bit wide. If this poses a design problem, a shadow PRF could be added for access by one of these stages.

Note that in this paper we do not take advantage of this feature of selectively increasing the cmpp latency, since modulo scheduled loops are generally resource rather than latency bound, and extending cmpp latency has only a small impact on the overall schedule length for most loops in our applications [23, 22]. However, for acyclic regions, selectively varying cmpp latency for certain operations does decrease schedule length resulting in performance improvement [22].

# 4. PROBABILISTIC PREDICATE-AWARE MODULO SCHEDULING

In this section, the details of the PPAMS algorithm are presented. PPAMS is an extension of conventional IMS; it decreases the expected *dynamic* kernel schedule length by relaxing resource constraints. It achieves this by allowing operations to reserve the same resource in the same cycle, while estimating and accounting for the resulting expected delay due to conflicts.

## 4.1 Computing Expected Conflict Delay

A key feature of PPAMS is the method of estimating the expected delay due to conflicts when two or more predicated operations share the same resource. The example code in Figure 1 and the machine model in Table 1 are used to demonstrate this technique.

We define an **execution vector** for a group of predicates as an assignment of a particular boolean value to each of these predicates. The expected delay ($ED_{cfl}$) for a group of predicated operations is the sum of the expected delays due to each of the execution vectors that result in a conflict. The expected delay of an execution vector is the number of extra delay cycles required to recover from the conflict multiplied by the probability of occurrence of this execution vector.

For example, assuming CDRL=1, the expected delay when operations $A3$, $A4$ and $A6$, guarded by the predicates $p3$, $p4$ and $p5$, respectively, are scheduled at the same time on a single ALU is

$$\begin{aligned}
&ED_{cfl}(A3\ ?\ p3,\ A4\ ?\ p4,\ A6\ ?\ p5)\\
&\quad = (1+2-1) \times P(p3=T, p4=T, p5=F)\\
&\quad + (1+2-1) \times P(p3=T, p4=F, p5=T)\\
&\quad + (1+2-1) \times P(p3=F, p4=T, p5=T)\\
&\quad + (1+3-1) \times P(p3=T, p4=T, p5=T)\\
&\quad = 2 \times P(p3=T, p4=T, p5=F)\\
&\quad + 2 \times P(p3=T, p4=F, p5=T)\\
&\quad + 2 \times P(p3=F, p4=T, p5=T)\\
&\quad + 3 \times P(p3=T, p4=T, p5=T)
\end{aligned}$$

The first three terms on the right side compute the expected delay for all possible execution vectors that cause exactly one conflict. For example, when the execution vector $(p3 = T, p4 = T, p5 = F)$ occurs, it will cause $A3$ and $A4$ to conflict over the ALU, resulting in 2 extra cycles of delay: 1 cycle (CDRL) to detect the conflict, plus 2 more cycles to dispatch the two conflicting ALU operations, minus 1 cycle to account for the fact that when there are no conflicts, it takes exactly one cycle to dispatch all operations. Similarly, as shown by the fourth term, to recover from conflicts in $(p3 = T, p4 = T, p5 = T)$ will take 3 extra cycles.

**Computing the Probability of an Execution Vector:** To compute the probability of a given execution vector, we introduce a Predicate Relationship Graph (**PRG**), which is similar in concept to the partition graph in [11] and the predicate hierarchy graph in [14]. A PRG represents the relationship between the predicates in a predicated block of code. Each node in the graph corresponds to a particular predicate and is labeled with the activation frequency of this predicate, as obtained from a profile run. There are two kinds of edges which connect the nodes of the PRG: implication edges (I-edges) and disjointness edges (D-edges). There is an I-edge from predicate $pi$ to $pj$ if $pj$ implies $pi$, i.e., whenever $pj$ is True, $pi$ is also True. This means that in the original non-predicated code an operation guarded by $pi$ lies on any control path that passes through an operation guarded by $pj$. A D-edge indicates that the two predicates it connects are disjoint, i.e., whenever one of these two predicates evaluates to True during execution, the other must evaluate to False.

One important simplifying assumption that we make in this pa-

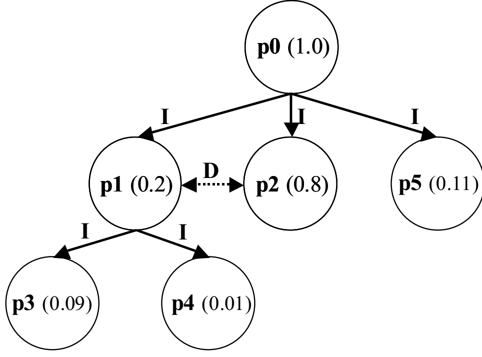



**Figure 4: Predicate Relationship Graph**

per is the *independence* assumption. This assumption states that any two predicates not connected by an edge are deemed to be independent, that is the probability that one of them evaluates to True (or False) is the same, regardless of the value of the other predicate. Note that any two predicates defined in different loop iterations are never connected by an edge and therefore are always assumed to be independent.

The PRG for the example code of Figure 1 is shown in Figure 4. The root of PRG is predicate $p0$ which always evaluates to True during the execution of this code, hence it has a frequency of 1.0. Obviously, $p1$ implies $p0$, as do $p2$ and $p5$. $p3$ implies $p1$ since the cmpp operation C2 may only set $p3$ if its guarding predicate, $p1$, is True (and its condition, $lv > 10$, is also True); likewise, $p4$ implies $p1$. Since implication is a transitive relation, we avoid cluttering the graph with redundant I-edges, e.g., we do not need to include an I-edge from $p0$ to $p3$ since there is an I-edge from $p0$ to $p1$ and one from $p1$ to $p3$. There is a D-edge between $p1$ and $p2$ since these two predicates (as defined in C1) are disjoint. Furthermore, since $p3$ and $p4$ imply $p1$, they must also be disjoint from $p2$. Therefore, the corresponding D-edges between $p3$ and $p2$ as well as $p4$ and $p2$ are not shown, as they can be inferred from the graph.

We use the PRG, the independence assumption, and techniques from elementary probability theory to compute the probability of occurrence of an execution vector. We demonstrate this computation with a simple representative example, the formal comprehensive derivation is presented in [22].

Suppose $A1$, $A3$, and $A6$ of the example are scheduled in the same row of an MRT. One probability we would need to compute is $P(p0 = T, p3 = F, p5 = T)$ for the case that $A1$ and $A6$ execute, but $A3$ does not. Furthermore, suppose that $A1$ and $A3$ are scheduled at the same time in the SRT, but $A6$ is scheduled at a different time in the SRT, i.e., $A1$ and $A3$ in the MRT row are from the same loop iteration, but $A6$ is from a different iteration. Since predicates from different iterations are assumed to be independent, $P(p0 = T, p3 = F, p5 = T) = P(p0 = T, p3 = F) \times P(p5 = T)$. To compute $P(p0 = T, p3 = F)$, we notice from the PRG that $p0$ evaluates to True and $p3$ evaluates to False, when one of two events occurs: (i) $p0$ evaluates to True, $p1$ evaluates to True and $p3$ evaluates to False, or (ii) $p0$ evaluates to True and $p1$ evaluates to False (in which case $p3$ is guaranteed to be False by the implication relationship). Hence, $P(p0 = T, p3 = F) = P(p0 = T, p1 = T, p3 = F) + P(p0 = T, p1 = F)$. We use the definition of conditional probability to compute each of the terms. For the second term, $P(p0 = T, p1 = F) = P(p1 = F|p0 = T) \times P(p0 = T)$. For the first term, $P(p0 = T, p1 = T, p3 = F) = P(p3 = F \mid p0 = T, p1 = T) \times P(p0 = T, p1 = T) = P(p3 = F \mid p1 = T) \times P(p1 = T \mid p0 = T) \times P(p0)$. $P(pi = T)$ is measured as the fraction of all profiled loop iterations for which $pi$ evaluates to True. $P(pi = T|pj = T)$ is the conditional edge transition probability that $pi$ evaluates to True, given $pj$ evaluates to True and is equal to $\frac{P(pi=T)}{P(pj=T)}$. Furthermore, $P(pi = F|pj = T) = 1 - \frac{P(pi=T))}{P(pj=T)}$. Hence, by plugging in the values, we obtain, $P(p0 = T, p3 = F, p5 = T) = P(p0 = T, p3 = F) \times P(p5 = T) = [\frac{(0.2-0.09)}{0.2} \times \frac{0.2}{1.0} \times 1.0 \times 0.11 + \frac{(1-0.02)}{1.0} \times 1.0] \times [0.11] = (0.0121 + 0.8) \times 0.11 = 0.10$.

Certain execution vectors are illegal since they will never occur, and hence have 0 probability. For example, $P(p1 = T, p2 = T) = 0$ since $p1$ is disjoint from $p2$ as indicated by the D-edge in the PRG. As another example, it is impossible for both $A2$ and $A6$ to execute and $M2$ not to execute if all three operations are from the same iteration; this is due to the fact that $A6$ and $M2$ are guarded under the same predicate $p5$, and clearly $P(p1 = T, p5 = T, p5 = F) = 0$.

We developed a general probabilistic model to compute the probability of occurrence of an arbitrary execution vector in terms of the PRG edge transition probabilities. This model takes advantage of the three possible relationships between a pair of predicates: disjointness, implication, and lastly independence (which we assume whenever neither disjointness nor implication has been found).

Finally, note that the computation complexity of the probability of occurence of a given execution vector can take time that is exponential in the size of the execution vector, in the worst case. However, the actual compilation time was small, since in practice our execution vector length rarely exceeded three.

## 4.2 Main PPAMS Extensions

The PPAMS algorithm uses the expected delay computation technique to try to find the smallest expected initiation interval ($II_{expected}$) for which a valid schedule can be found. $II_{expected} = II_{static} + II_{CflDelay}$, where $II_{static}$ is the static initiation interval when the delay due to conflicts is ignored, and $II_{CflDelay}$ is the expected delay due to conflicts. The following describes our main changes to the baseline iterative modulo scheduling (IMS) to support PPAMS.

**Computing ResMII:** IMS computes the resource-constrained lower bound (ResMII) by unconditionally adding the number of times that each operation uses a particular type of resource to that resource's usage count, regardless of the operation's guarding predicate. The cumulative usage count for the most heavily used resource determines ResMII for IMS. For our example in Figure 1, $ResMII = 10$, since there are 10 ALU operations and only a single ALU unit.

For PPAMS, a similar calculation is done, except that an operation is allowed to use a particular may-use resource only a fraction of the time, based on its execution frequency. When an operation uses a resource, this resource's usage count is only incremented by operation's execution frequency. Thus, continuing with the example, the probabilistic predicate-aware resource-constrained lower bound, PPAMS $ResMII = (1.0(A1) + 1.0(C1) + 0.2(A2) + 1.0(C2) + 0.09(A3) + 1.0(C3) + 0.01(A4) + 0.8(A5) + 1.0(C4) + 0.11(A6))/(1ALU) = 6.21$, which is the execution frequency summed over all ALU operations. Note that un-predicated operations have a frequency of 1.0. Finally, since the must-use resources are reserved regardless of guarding predicate values, their usage count is computed as in the baseline case. The lower bound ob-

```
double ppams_Main()
{
    II_low = max(ppamsResMII, ppamsRecMII);
    II_high = max(baseResMII, baseRecMII);
    while(II_high – II_low > smalldelta) {
      II_middle = (II_high + II_low) / 2;
      if (ppams_FindSchedule(II_middle) == true)
          II_high = II_middle;
      else
          II_low = II_middle;
    }
    return II_high;
}
```

(a)

```
bool ppams_FindSchedule(II_expected)
{
    for(II_static = floor(max(WidthResMII, ppamsRecMII));
              II_static <= floor(II_expected); II_static++) {
      II_CflDelay = II_expected  - II_CflDelay;
      if(ppams_IterativeScheduler(II_static, II_CflDelay)==true)
          return true;
    }
    return false;
}
```

(b)

**Figure 5: Main PPAMS scheduling routines**

tained is likely to be an optimistic estimate of the actual schedule length since it does not account for potential conflicts that may occur.

**Main Scheduler:** The main PPAMS driver (shown in Figure 5(a) uses a binary search method to find the smallest value of $II_{expected}$ for which a valid schedule can be found. The low (high) bound, $II_{low}$ ($II_{high}$) is computed as the maximum of the ppams (baseline) $ResMII$ and $RecMII$. Note that $ppamsRecMII$ is never less than $baseRecMII$ since PPAMS requires cmpp latency to be increased, which may increase the length of the critical path, but never decreases it. The **while** loop in Figure 5(a) calls the **ppams_FindSchedule** routine which tries to find a valid schedule for $II_{middle}$ - a halfway point between $II_{low}$ and $II_{high}$. If a schedule is found, $II_{high}$ is reduced to $II_{middle}$, otherwise $II_{low}$ is increased to $II_{middle}$. These steps are repeated until the difference between $II_{high}$ and $II_{low}$ drop below a small threshold value ($smalldelta$), at which time the $II_{expected}$ value is set equal to the current value of $II_{high}$.

**ppams_FindSchedule** (Figure 5(b)) tries to find a valid schedule for $II_{expected}$ (the sum of $II_{static}$ and $II_{CflDelay}$). In fact, it is possible for several valid schedules of length $II_{expected}$ to exist for several different values of $II_{static}$ and $II_{CflDelay}$ as long as both add up to $II_{expected}$. Therefore, we vary $II_{static}$ in the loop starting from the ceiling of the maximum of the width resource- and the recurrence-constrained lower bound up to the **floor** of $II_{expected}$. The width resource-constrained lower bound, *WidthResMII*, is defined as $\frac{\text{number of operations in the loop body}}{\text{machine fetch width } (FW)}$ and is a hard limit on $II_{static}$ for a machine of the given fetch width ($FW$). Thus, $II_{static}$ can never be less than $WidthResMII$, and since $II_{static}$ must be an integer, it cannot be less than ceiling(WidthResMII). $II_{static}$ is also constrained by the recurrence-constrained lower bound, $ppamsRecMII$, computed from the data dependence graph after the cmpp latency extension phase. The extension of cmpp latencies may increase the length of the critical path compared with the baseline code. Note that with PPAMS, as opposed to baseline modulo scheduling, $II_{static}$ can be less than $ResMII$. In addition, since we are only interested in schedules with $II \leq II_{expected}$, the $II_{static}$ term of a schedule of interest cannot exceed floor($II_{expected}$).

For each value of $II_{static}$, the corresponding maximum allowed value of $II_{CflDelay}$ is computed (as the difference between $II_{expected}$ and $II_{static}$) and the **ppams_IterativeScheduler** is called with both values passed as parameters. The **ppams_IterativeScheduler** returns True if a valid schedule is found, in which case **ppams_FindSchedule** also returns the value True. Otherwise, the next combination of $II_{static}$ and $II_{CflDelay}$ is tried, until either the **ppams_IterativeScheduler** returns True or the **for** loop terminates. If the loop terminates, no valid schedule has been found for a given value of $II_{expected}$, and **ppams_FindSchedule** returns False to the **ppams_Main driver**.

**ppams_IterativeScheduler**($II_{static}$, $II_{CflDelay}$) is a slightly modified version of the baseline iterative modulo scheduler algorithm [16] which iteratively schedules and unschedules the operations of the loop until either a valid schedule is found or the maximum number of allowed scheduling steps (a run-time budget) is exceeded. In PPAMS, the scheduler tries to place operation *op* at Time *r* of the MRT that is chosen so that the increase in total estimated conflict delay (i.e., the summation of the estimated conflict delay over all rows of the MRT) is minimized. If an operation cannot be placed without exceeding the delay constraint $II_{CflDelay}$, a backtrack step is executed in which some operations are chosen to be unscheduled (to be tried again later) so as to allow the current *op* to be scheduled without violating the delay constraint.

## 4.3 Example Application of PPAMS

To illustrate the application of the algorithm, PPAMS is applied to the example in Figure 1 with the machine model in Table 1, a cmpp latency of 3 cycles, and a fetch width equal to the number of function units (3).

We show the details of a single call to the **ppams_IterativeScheduler** function. The goal is to find a valid 6 cycle ($II_{static}$=6) schedule which also satisfies the delay constraint $II_{CflDelay} \leq 0.42$ cycles. 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 current delay values in the $II_{CflDelay}$ column are updated: the overall schedule length, which is $II_{static}$ plus the sum of the $II_CflDelay$ entries over all rows is shown in the last row of the MRT. We use delay due to conflict computation method from Section 4.1 and the PRG in Figure 4 to compute the value of $II_{CflDelay}$ at each

**Schedule Reservation Table**

| Time | A | M | CflDelay |
|---|---|---|---|
| 0 | A1 | | |
| 1 | | M1 | |
| 2 | | | |
| 3 | C1 | | |
| 4 | C2 | | |
| 5 | C3 | | |
| 6 | | | |
| 7 | A2?p1 A5?p2 | | |
| 8 | A3?p3 **A4?p4** | | 0.0018 |
| 9 | | | 0.02 |
| 10 | | | 0.02 |
| 11 | | | 0.02 |
| 12 | | | 0.02 |
| 13 | | | 0.02 |
| 14 | | | |
| 15 | | | |
| 16 | | | |
| 17 | | | |
| 18 | | | |
| 19 | | | |
| 20 | | | |
| 21 | | | |
| 22 | | | |
| 23 | | | |

**Modulo Reservation Table**

| Time | A | M | CflDelay |
|---|---|---|---|
| 0 | n:A1 | | 0 |
| 1 | n-1:A2 n-1:A5 | n:M1 | 0 |
| 2 | n-1:A3 n-1:A4 | | 0.0018 |
| 3 | n:C1 | | 0 |
| 4 | n:C2 | | 0 |
| 5 | n:C3 | | 0 |
| 0.0018 | **expected delay due to conflicts** | | |

(a) Scheduling A4

**Schedule Reservation Table**

| Time | A | M | CflDelay |
|---|---|---|---|
| 0 | A1 | | |
| 1 | | M1 | |
| 2 | | | |
| 3 | C1 | | |
| 4 | C2 | | |
| 5 | C3 | | |
| 6 | | | |
| 7 | A2?p1 A5?p2 | | |
| 8 | A3?p3 A4?p4 | | |
| 9 | | | 2.0 |
| 10 | | | 2.0 |
| 11 | | | 2.0 |
| 12 | | | 2.0 |
| 13 | | | 2.0 |
| 14 | **C4** | | 0.1991 |
| 15 | | | |
| 16 | | | |
| 17 | | | |
| 18 | | | |
| 19 | | | |
| 20 | | | |
| 21 | | | |
| 22 | | | |
| 23 | | | |

**Modulo Reservation Table**

| Time | A | M | CflDelay |
|---|---|---|---|
| 0 | n:A1 | | 0 |
| 1 | n-1:A2 n-1:A5 | n:M1 | 0 |
| 2 | n-1:A3 n-1:A4 **n-2:C4** | | 0.1991 |
| 3 | n:C1 | | 0 |
| 4 | n:C2 | | 0 |
| 5 | n:C3 | | 0 |
| 0.1991 | **expected delay due to conflicts** | | |

(b) Scheduling C4

**Schedule Reservation Table**

| Time | A | M | CflDelay |
|---|---|---|---|
| 0 | A1 | | |
| 1 | | M1 | |
| 2 | | | |
| 3 | C1 | | |
| 4 | C2 | | |
| 5 | C3 | | |
| 6 | | | |
| 7 | A2?p1 A5?p2 | | |
| 8 | A3?p3 A4?p4 | | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| 12 | | | |
| 13 | | | |
| 14 | C4 | | |
| 15 | | | |
| 16 | | | |
| 17 | **A6** | | 0.22 |
| 18 | | M2 | 0.22 |
| 19 | | | 0.22 |
| 20 | | | invalid |
| 21 | | | 0.22 |
| 22 | | | 0.22 |
| 23 | | | 0.22 |

**Modulo Reservation Table**

| Time | A | M | CflDelay |
|---|---|---|---|
| 0 | n:A1 | n-3:M2 | 0 |
| 1 | n-1:A2 n-1:A5 | n:M1 | 0 |
| 2 | n-1:A3 n-1:A4 **n-2:C4** | | 0.1991 |
| 3 | n:C1 | | 0 |
| 4 | n:C2 | | 0 |
| 5 | n:C3 n-2:A6 | | 0.22 |
| 0.4191 | **expected delay due to conflicts** | | |

(c) Scheduling A6

**Figure 6: PPAMS Scheduling of the example in Figure 1 for $II_{static}$=6 and $II_{CflDelay}$=0.42**

step. We omit the $B$-resource column to save space: in traditional modulo scheduling of the counter-based loops, a single branch operation $B1$ is always scheduled first within the first $II$ rows of the SRT and never causes conflicts.

Figure 6(a) shows the partial SRT and MRT with operations $A1$, $M1$, $C1$, $C2$, $C3$, $A2$ and $A3$ and $A5$ already scheduled, and operation $A4$ is being scheduled. Note that the partial schedule (before $A4$ is scheduled) has no conflicts since each ALU operation occupies a separate entry in the MRT, except for $A2$ and $A5$ which are disjoint. The earliest schedule time for $A4$ (in the SRT) is Time 8 since $A4$ is dependent on the three-cycle $C3$ operation scheduled at Time 5. If $A4$ is scheduled at Time 8, it will share the ALU with $A3$ from the same iteration and will result in 0.0018 cycles of expected delay due to conflict, namely 2 recovery cycles $\times P(p3 = T, p4 = T) = 2 \times 0.01 \times 0.09 = 0.0018$, as shown in the rightmost column. Scheduling $A4$ at any of Times 9 through 12 of the SRT would result in 0.02 cycles of delay (2 recovery cycles $\times$ $P(p4 = T) = 2 \times 0.01 = 0.02$), since the operations scheduled at each of these times in the MRT always execute (execution frequency = 1.0). Scheduling $A4$ at Time 13 will also result in 0.02 cycles of delay since $A2$ and $A5$ are a complete set of disjoint operations (they are guarded under complementary predicates). The scheduler places $A4$ at Time 8 of the SRT (Time 2 of the MRT) which causes the smallest increase in the overall conflict delay for the given partial schedule.

Operation $C4$ is scheduled similarly, as shown in Figure 6(b), as is operation $A6$, resulting in the final schedule shown in Figure 6(c). The overall delay is 0.4191 cycles (0.1991 cycles from Time 2 of the MRT plus 0.22 cycles from Time 5) which results in a valid schedule that satisfies both resource and delay constraints.

## 5. PERFORMANCE EVALUATION

We use an existing VLIW compiler toolset, **Trimaran** [25], to evaluate the effectiveness of PPAMS. This compiler system is capable of performing if-conversion with hyperblock formation [14], modulo scheduling [16], and predicate analysis [11], 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 to analyze predicated code and construct predicate relationship graphs to compute the expected delay due to conflicts as described in Section 4.1.

We use the notation (F,I,FP,M,B,C, D) to represent the processor in this study. **F** is fetch width, **I** - number integer units, **FP** - number of floating-point units, **M** - number of memory units, **B** - number of branch units, **C** - latency of the predicate defining operation (cmpp), and **D** is conflict detection latency, or the number

| | % ppa-ready | Base | | | | | DPAMS | | | | | PPAMS | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Benchmark | regions | BRec | BRes | BII | Besc | Brr | DRec | DRes | DII | Desc | Drr | PRec | PRes | PIIs | PIIc | PIId | %Error | Pesc | Prr |
| cjpeg | 6.81 | 1.31 | 6.49 | 6.49 | 2.00 | 10.73 | 1.92 | 5.89 | 6.35 | 2.44 | 13.57 | 1.92 | 4.69 | 5.50 | 5.69 | 5.66 | 0.53 | 4.53 | 17.70 |
| djpeg | 48.03 | 1.00 | 58.64 | 58.64 | 1.00 | 28.38 | 1.00 | 53.17 | 53.66 | 1.01 | 29.89 | 1.00 | 42.69 | 46.22 | 47.56 | 47.67 | -0.23 | 2.50 | 48.26 |
| epic | 3.37 | 2.16 | 21.02 | 21.41 | 1.77 | 12.78 | 2.93 | 19.10 | 19.48 | 2.00 | 13.86 | 2.93 | 16.38 | 12.39 | 16.44 | 16.33 | 0.67 | 7.86 | 16.85 |
| unepic | 53.45 | 1.00 | 13.27 | 13.27 | 1.85 | 19.31 | 1.00 | 12.84 | 12.84 | 2.70 | 21.70 | 1.00 | 10.52 | 10.45 | 10.82 | 11.08 | -2.40 | 3.14 | 28.37 |
| g721encode | 39.70 | 1.00 | 30.00 | 30.00 | 1.00 | 12.00 | 1.00 | 21.00 | 23.50 | 3.00 | 21.50 | 1.00 | 20.24 | 18.50 | 21.20 | 19.36 | 8.68 | 4.50 | 27.00 |
| g721decode | 39.63 | 1.00 | 30.00 | 30.00 | 1.00 | 12.00 | 1.00 | 21.00 | 23.00 | 4.00 | 24.00 | 1.00 | 20.26 | 19.50 | 21.52 | 19.87 | 7.67 | 4.50 | 25.50 |
| ghostscript | 20.08 | 7.87 | 43.13 | 44.10 | 1.02 | 20.85 | 7.87 | 33.35 | 34.33 | 2.02 | 23.79 | 7.87 | 31.35 | 30.38 | 31.39 | 31.37 | 0.06 | 3.97 | 45.26 |
| gsmdecode | 89.04 | 7.89 | 27.66 | 27.85 | 1.10 | 13.85 | 11.39 | 24.88 | 25.66 | 1.97 | 18.14 | 11.39 | 21.40 | 19.22 | 22.72 | 22.61 | 0.48 | 7.62 | 49.18 |
| gsmencode | 95.34 | 7.81 | 73.88 | 74.34 | 1.00 | 16.50 | 8.86 | 46.45 | 54.49 | 1.93 | 17.69 | 8.86 | 45.30 | 49.37 | 52.63 | 52.58 | 0.10 | 3.41 | 33.55 |
| mesamipmap | 38.12 | 1.00 | 22.00 | 22.00 | 1.67 | 20.67 | 1.00 | 16.33 | 16.33 | 2.67 | 27.00 | 1.00 | 15.92 | 16.00 | 16.33 | 16.32 | 0.06 | 3.00 | 24.33 |
| mpeg2dec | 33.43 | 1.00 | 28.17 | 28.17 | 1.50 | 17.02 | 1.00 | 25.69 | 25.69 | 1.85 | 15.90 | 1.00 | 18.58 | 19.63 | 21.60 | 21.68 | -0.37 | 3.26 | 30.82 |
| mpeg2enc | 76.85 | 2.96 | 20.21 | 20.21 | 1.01 | 8.24 | 4.91 | 17.26 | 17.26 | 1.99 | 10.21 | 4.91 | 12.34 | 12.30 | 13.86 | 13.86 | 0.00 | 4.08 | 18.61 |
| pegwitdec | 55.53 | 1.96 | 20.51 | 20.51 | 0.08 | 13.94 | 1.96 | 18.56 | 18.56 | 1.08 | 14.00 | 1.96 | 12.39 | 12.75 | 13.90 | 13.96 | -0.43 | 5.93 | 38.16 |
| pegwitenc | 69.14 | 1.61 | 19.33 | 19.33 | 0.96 | 13.36 | 1.61 | 18.26 | 18.26 | 1.60 | 16.18 | 1.61 | 12.66 | 13.30 | 13.60 | 13.97 | -2.72 | 4.97 | 29.37 |
| rasta | 8.17 | 3.02 | 6.80 | 6.81 | 1.25 | 7.42 | 4.14 | 6.53 | 6.68 | 1.84 | 7.86 | 4.14 | 4.85 | 5.56 | 5.89 | 5.94 | -0.85 | 4.66 | 16.18 |
| rawcaudio | 99.82 | 20.00 | 24.00 | 26.00 | 1.00 | 12.00 | 20.00 | 22.00 | 25.00 | 3.00 | 20.00 | 20.00 | 17.13 | 25.00 | 25.00 | 25.00 | 0.00 | 1.00 | 17.00 |
| rawdaudio | 99.83 | 6.00 | 20.00 | 20.00 | 1.00 | 13.00 | 10.00 | 18.00 | 18.00 | 3.00 | 18.00 | 10.00 | 14.21 | 15.00 | 16.12 | 16.09 | 0.19 | 11.00 | 42.00 |
| **Avg. (4-wide)** | **51.55** | **4.03** | **27.36** | **27.59** | **1.19** | **14.83** | **5.39** | **22.37** | **23.48** | **2.24** | **18.43** | **5.39** | **18.88** | **19.47** | **20.96** | **20.79** | **1.59** | **4.70** | **29.89** |
| **Avg. (6-wide)** | **54.18** | **4.03** | **13.90** | **14.63** | **2.15** | **18.82** | **5.39** | **12.36** | **14.09** | **3.29** | **23.71** | **5.39** | **11.11** | **13.34** | **13.48** | **21.68** | **0.87** | **4.33** | **26.24** |

**Table 2: Various scheduling measurements for 4-wide base, dpas and ppas machines, cmpp latency is 3 cycles and CDRL is 0 cycles**

of cycles after the conflict is detected in which the recovery mechanism dispatches the first conflicting operation. We use two base processors in our study: (4,2,1,1,1,1, -) and (6,4,2,1,1,1, -) called $P_{base}(4)$ and $P_{base}(6)$, respectively. In addition, we assumed 64 scalar and 64 rotating registers in our experiments and operation latencies that match the Itanium processor.

Each baseline processor $P_{base}(i)$ is compared with two corresponding probabilistic predicate-aware processors $P_{ppas}(i, 0)$ and $P_{ppas}(i, 1)$ with the same number of resources as the baseline processor, but conflict detection latency of zero and one cycles, respectively. We also compare the performance of the baseline processor $P_{base}(i)$ with the corresponding deterministic predicate-aware processor $P_{dpas}(i)$. As mentioned in Section 2, DPAMS avoids conflicts by conservatively combining only provably disjoint operations. In addition, we assume that if an operation's predicate is to be read early (in the predicate read and dispatch stage), the operation must be separated from its corresponding cmpp by at least 3 cycles for both deterministic and probabilistic predicate-aware machines. All cmpp latencies are increased to 3 cycles: as our results show, most of the loops are resource bound (rather than recurrence bound), and therefore the results have little sensitivity to the cmpp latency.

We evaluated the set of 17 MediaBench [13] applications, applying deterministic and probabilistic predicate-aware scheduling optimizations to their modulo scheduled loops. Clearly, PPAMS can only benefit if-converted regions of code that contain at least one *if-then* clause, including loops with CASE statements, and will be ineffective for other code regions. We call these regions **ppa-ready**). There are total of 130 ppa-ready loops. All the remaining loops are scheduled using a conventional baseline modulo scheduler.

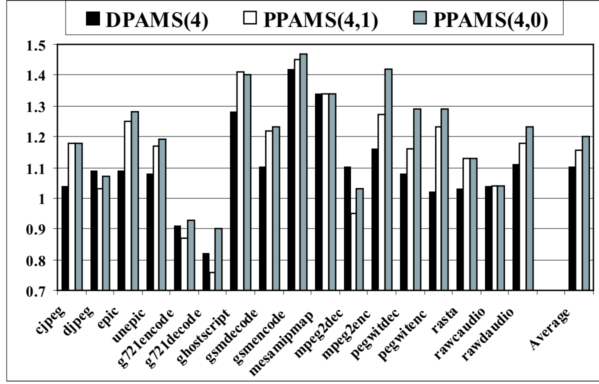
## 5.1 Evaluation Results

Table 2 shows various scheduling measurements for $P_{base}(4)$, $P_{dpas}(4)$ and $P_{ppas}(4,0)$ machines. For a given application, each measurement is an average over all ppa-ready loops in this application weighted by their execution frequency. Column 1 lists all the benchmarks. The second to last row of the table, Avg. (4-wide), shows the average results for a 4-wide machine, and the last row, Avg. (6-wide), shows the average results for a 6-wide machine. Column 2 shows the percent of dynamic operations that lie in ppa-ready cyclic regions (loops with at least one predicated operation) for a given application. On average, 52% of operations are from such regions. Columns BRec, DRec and PRec show the $RecMII$ for the three schedulers. As expected, for some applications (epic, gsmdecode, gsmencode, mpeg2enc, rasta, rawcaudio and rawdaudio) $RecMII$ increases for both DPAMS and PPAMS. This happens because the predicate-aware pipelines require that the cmpp latency be increased to support early discarding of operations predicated on a False predicate. For these applications, some of the cmpps lie on the critical recurrence cycles, thus the increased cmpp latency increases RecMII. We also see that for most of the applications $RecMII$ is not a limiting factor to performance for all three schedulers because it is much less than $ResMII$.

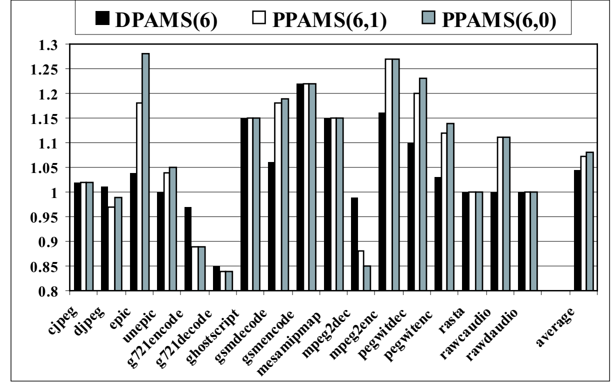
Columns labeled BRes, DRes and PRes show the value of $ResMII$ for the three schedulers, respectively. As explained in Section 4.2, the baseline scheduler increments the resource usage by 1 regardless of the operation's guarding predicate and its execution frequency. DPAMS increments the resource usage count by 1 per group of disjoint operations. Many ppa-ready loops have a large number of if-then-else statements; DPAMS reduces their $ResMII$ on average by 22%. PPAMS further decreases the $ResMII$ by an average of 15%, with respect to DPAMS, by incrementing the resource usage count by the operation's execution frequency. Note that for some of the benchmarks, such as gsmencode and mesamipmap, both DPAMS and PPAMS result in similar $ResMII$. This is due to the fact that these applications contain a number of dominating loops which consist primarily of a large number of well balanced if-then-else statements.

Columns labeled BII, DII, and PIIs show the $II$ for the three schedulers. For PPAMS, $PIIs$ is a static $II$ that does not account for the delay due to conflicts. PPAMS reduces the $II$ by 17% with respect to DPAMS, since it allows more flexible operation combining than DPAMS. As the example in Section 4.3 demonstrated, PPAMS can combine any operations, both disjoint and non-disjoint, from the same or different loop iterations, whereas DPAMS can only combine disjoint operations from a single loop iteration.

This flexibility in combining may, however, result in an additional delay due to conflict ($II_{CflDelay}$). The column labeled PIIc shows the compiler estimate of the expected initiation interval ($II_{expected}$), which does include $II_{CflDelay}$. Based on the formula $II_{expected} = II_{static} + II_{CflDelay}$, there are two ways to achieve a given $II_{expected}$: either by allowing more sharing




(a) Speedup of $P_{ppas}$(4,0/1) and $P_{dpas}$(4) over $P_{base}$(4)




(b) Speedup of $P_{ppas}$(6,0/1) and $P_{dpas}$(6) over $P_{base}$(6)

**Figure 7: Speedup for cyclic regions alone**

with tighter operation scheduling, which decreases $II_{static}$ but increases $II_{CflDelay}$, or by allowing less sharing which increases $II_{static}$ but decreases $II_{CflDelay}$. For example with epic, PPAMS chooses the first approach and achieves an $II_{static}$ of 12.39 and $II_{CflDelay}$ of 4.05 (16.44 - 12.39). Conversely for unepic, PPAMS chooses the second approach and achieves an $II_{static}$ of 10.45 and $II_{CflDelay}$ of 0.37 (10.82-10.45).

Columns labeled PIId and %Error show the achieved run-time initiation interval ($II_{dynamic}$), and the relative error between $II_{dynamic}$ and $II_{expected}$ computed as $(II_{dynamic} - II_{expected})/II_{dynamic}$. Note that for the last ('average') row, we show the average of the absolute values of the relative errors to avoid mutual cancellation of the negative and positive errors. We see that for most of the applications the error is quite small (less than 3%). However, in g721encode, the error is 8.7%, and 7.7% in g721decode. This happens because some of the predicates of operations that map to the same resource violate the independence assumption that the compiler made during scheduling. These predicates turn out to be correlated and result in more run-time conflict than was originally estimated by the compiler.

Columns labeled Besc, Desc and Pesc show the size of the epilogue (in terms of its stage count) of the modulo scheduled loops for the three schedulers. Epilogue stage count ($esc$) is defined as the ceiling of the length of the loop schedule in the SRT divided by the $II_{static}$. We can see that on average DPAMS increases the baseline schedule epilogue by a factor of 1.5, and PPAMS doubles the baseline schedule epilogue. The reason for this increase in the epilogue size is twofold: first, as cmpp latency increases, the SRT schedule length increases, and second, both DPAMS and PPAMS intentionally stretch the schedule by moving operations further away from their producers to allow more aggressive combining.

Finally, columns labeled Brr, Drr and Prr show the average number of rotating registers [7] required by each of the three schemes. Rotating registers are used to allocate variables with multiple lifetimes, which occur when a variable is simultaneously live in several loop iterations [19]. The farther away the latest consumer is scheduled from its producer, the more rotating registers the producer will require. Hence, the increase in the number of required rotating registers that we see with DPAMS and PPAMS happens for the same two reasons that cause the increase in the size of the epilogue for these two schemes. However, despite the increased demand for rotating registers with DPAMS and PPAMS, the 64 rotating registers assumed in our experiments are enough for almost all loops. In the very few cases in which the optimum loop schedule requires more than 64 rotating registers, the expected initiation interval of the loop is increased in order to reduce the rotating register requirement.

The last row summarizes similar data for $P_{base}$(6), $P_{dpas}$(6) and $P_{ppas}$(6,0) machines for which conclusions similar to those for the 4-wide machine can be drawn. However, the 6-wide predicate-aware machine with 4 integer units has more resources and therefore achieves less benefit from operation sharing than the equivalent 4-wide machine with 2 integer units. This explains the lower performance gain with DPAMS and PPAMS for the 6-wide machines.

Figure 7 shows the actual speedups achieved by DPAMS and PPAMS on the cyclic regions alone. The leftmost bars of Figure 7(a) show the speedup achieved by the 4-wide DPAMS machine over the baseline machine. The middle and rightmost bars show the speedup achieved by $P_{ppas}$(4,1) and $P_{ppas}$(4,0), respectively, over the 4-wide baseline machine. Figure 7(b) shows similar data for the 6-wide machines. We can see that for some of the applications, such as g271decode and g721encode, the performance drops for both DPAMS and PPAMS on both 4- and 6-wide machines. However, from Table 2 we see that the achieved $II$ with either predicate-aware technique is better than $II$ for the baseline machine. This behavior is due to the large size of the epilogue produced by both schemes and the relatively short trip count of the loops in these applications. The run-time of a modulo scheduled loop with trip count $n$ is equal to $(n + esc) \times II$. If $n$ is small, a large $esc$ can have significant impact on overall loop performance. A short trip count loop with larger $II$ and shorter epilogue may outperform the same loop when scheduled with a smaller $II$ and longer epilogue, as is the case here.

On average a 4(6)-wide PPAMS processor with conflict detection latency of 0 cycles performs 10%(3.6%) better than a 4(6)-wide DPAMS machine, and 20%(8%) better than the corresponding baseline machine. We also notice that a 4(6)-wide PPAMS processor with a conflict detection latency of 0 cycles performs

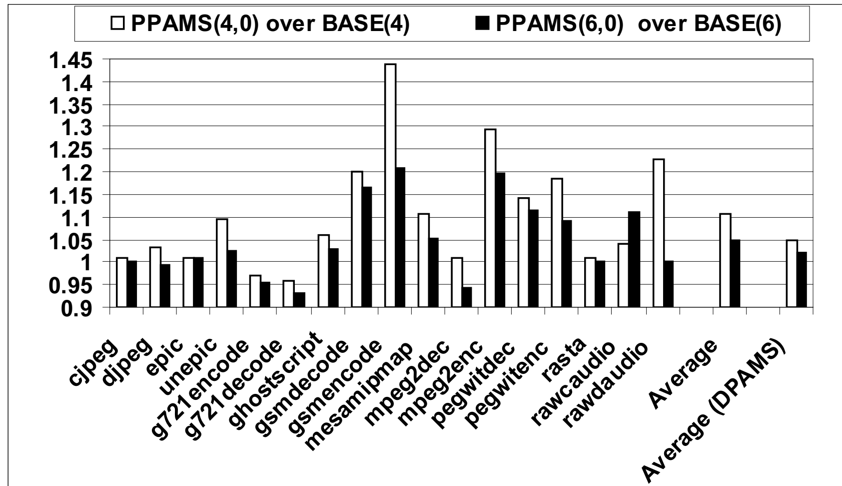



**Figure 8: Overall benchmark speedup with PPAMS**

5%(1)% better than the corresponding PPAMS processor with a conflict detection latency of 1 cycle. Higher conflict detection latency has more impact on the 4-wide PPAMS processor than on the 6-wide PPAMS processor because, as said above, the 6-wide machine has more resources than the 4-wide machine and thus has less resource sharing and fewer conflicts.

Finally, Figure 8 shows the overall speedup due to PPAMS for the entire application. The left bar shows the speedup of the 4-wide PPAMS processor, $P_{ppas}(4,0)$, over the corresponding baseline processor, $P_{ba}(4)$, for each application. The right bar shows similar data for the 6-wide PPAMS processor, $P_{ppas}(6,0)$. As shown by the two "Average" bars, the average total application speedup is 10% for the 4-wide machine and 5% for the 6-wide machine. The speedup achieved on the entire application is smaller than the speedup achieved on pa-ready cyclic regions alone, since as column 2 of Table 2 shows, these regions constitute on average only 52% of the total application baseline execution time. The two rightmost bars, "Average (DPAMS)", show the corresponding speedups over the baseline for the DPAMS processors, namely DPAMS processor achieves 5% speedup for the 4-wide machine, and 2.2% for 6-wide. We see that on average, for the entire application, the 4(6)-wide PPAMS processor outperforms the corresponding 4(6)-wide DPAMS processor by 5%(2.28%).

# 6. RELATED WORK

Software pipelining is a well studied technique for scheduling loops [3, 18, 17, 5, 9, 12, 4]. A number of techniques were proposed in the past to improve software pipeline schedules of loops with internal control flow [1, 8, 21]. Below we describe the techniques that are most relevant to our work.

Hierarchical reduction [12] collapses all conditional constructs into a single operation, modulo schedules the resulting straight-line code, and then regenerates all conditional constructs. All Path Pipelining [24] pipelines each path separately using a software pipelining technique for straight-line loops and then merges the pipeline kernels of the paths.

Enhanced Modulo Scheduling [27] initially modulo schedules predicated code in which disjoint operations are allowed to share resources. This is similar to both the PPAMS and DPAMS techniques [23]. However, PPAMS does not require the sharing operations to be disjoint, whereas the two other schemes do. In addition, Enhanced Modulo Scheduling 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 [27].

The advantage of the techniques in [12, 24, 27] (with an exception of the DPAMS technique) 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.

Modulo Scheduling with Multiple Initiation Intervals [28] schedules if-converted code so that control paths with higher execution frequencies (assigned based upon dynamic profiling of the loop) have shorter IIs than paths with lower frequencies. Predicated operations are used to execute the correct operations and the loop-back branch based upon which path is actually executed dynamically. This technique can heavily penalize the performance of some of the paths (and therefore overall performance) if the execution paths have similar frequencies.

# 7. CONCLUSIONS

We have proposed and evaluated a new probabilistic predicate-aware scheduling technique that can achieve better schedules on predicated cyclic code regions by reducing wasted resources in VLIW processors with predicated execution. To this end, we have made the following three contributions. First, we proposed a general concept of using the compiler to derive conflict-conscious schedules. This enables arbitrary (and not just provably disjoint) predicated operations to share the same resource in the same cycle. Second, we developed a probabilistic *delay due to conflicts* model that constructs and analyzes the predicate relationship graph for arbitrary predicated operations in order to derive the expected conflict delay. Third, we proposed a modulo scheduling algorithm that uses this delay model in conjunction with binary search method to compute the expected II in the face of resource conflicts, and find a schedule with minimal expected II.

The probabilistic predicate aware modulo schedulers have been

implemented and evaluated on the Mediabench application suite. The overall results show an average performance gain of 10% and 5% for 4-issue and 6-issue VLIW processors, respectively. For loops, probabilistic predicate-aware scheduling achieves an average gain of 20% and 8% for the same processors.

## 8. ACKNOWLEDGMENTS

We thank 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 and equipment donated by Intel Corporation.

## 9. REFERENCES


[1] A. Aiken and A. Nicolau. Perfect pipelining: a new loop parallelization technique. In *Proceedings of the 1988 European Symposium on Programming*, pages 221– 235, 1988.

[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] A. Charlesworth. An approach to scientific array processing: the architectural design of the ap-120b/fps-164 family. *IEEE Computer*, 14(9):18–27, Sept. 1981.

[4] J. Codina, J. Llosa, and A. Gonzalez. A comparative study of modulo scheduling techniques. In *Proceedings of the 16th international conference on Supercomputing*, pages 97– 106, 2002.

[5] R. G. Cytron. *Compiler-time Scheduling and Optimization for Asynchronous Machines*. Dept. of Computer Science Report UIUCDCS-R-84-1177, University of Illinois at Urbana-Champaign, Urbana, IL, 1984.

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

[7] J. Dehnert and R. Towle. Compiling for the Cydra 5. *Journal of Supercomputing*, 7(1):181–227, May 1993.

[8] K. Ebcioglu. A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. In *Proceedings 20th Workshop on Microprogramming*, pages 69– 79, Dec. 1988.

[9] P. Y.-T. Hsu. *Highly Concurrent Scalar Processing*. PhD Dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 1986.

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

[11] R. Johnson and M. Schlansker. Analysis techniques for predicated code. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 100–113, Dec. 1996.

[12] 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.

[13] 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*, pages 330–335, Dec. 1997.

[14] 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.

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

[16] 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.

[17] B. R. Rau, C. D. Glaeser, and R. L. Picard. Efficient code generation for horizontal architectures: Computer techniques and architectural support. In *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pages 131–139, 1982.

[18] B. R. Rau, P. J. Kuekes, and C. D. Glaeser. *A Statically Scheduled VLSI Interconnect for Parallel Processors*. Computer Science Press, 1981.

[19] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 283–299, June 1992.

[20] 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*, pages 158–169, December 1992.

[21] S. Shim and S. Moon. Split-path enhanced pipeline scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 14(5):447–462, May 2003.

[22] M. Smelyanskiy. *Hardware/Software Mechanism for Increasing Resource Utilization on VLIW/EPIC Processors*. Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, University of Michigan, 2004.

[23] M. Smelyanskiy, S. Mahlke, E. Davidson, and H. Lee. Predicate-Aware Scheduling: A Technique for Reducing Resource Constraints . In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 169– 178, March 2003.

[24] 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.

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

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

[27] 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.

[28] 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.