Clustered VLIW Architectures with Predicated Switching [*]

Margarida F. Jacome
jacome@ece.utexas.edu

Gustavo de Veciana
gustavo@ece.utexas.edu

Satish Pillai
satish@ece.utexas.edu

Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712

ABSTRACT

In order to meet the high throughput requirements of applications exhibiting high ILP, VLIW ASIPs may increasingly include large numbers of functional units(FUs). Unfortunately, 'switching' data through register files shared by large numbers of FUs quickly becomes a dominant cost/ performance factor suggesting that clustering smaller number of FUs around local register files may be beneficial even if data transfers are required among clusters. With such machines in mind, we propose a compiler transformation, predicated switching, which enables aggressive speculation while leveraging the penalties associated with inter-cluster communication to achieve gains in performance. Based on representative benchmarks, we demonstrate that this novel technique is particularly suitable for application specific clustered machines aimed at supporting high ILP as compared to state-of-the-art approaches.

1. INTRODUCTION

Real-time multimedia, graphics, visualization and communications applications often have high throughput requirements but fortunately may also exhibit high degrees of instruction level parallelism (ILP). In order to meet the performance requirements of these demanding applications it is of essence to use compilation techniques that expose/increase ILP and develop complementary specialized processor architectures that can efficiently support such ILP, e.g., Very Large Instruction Word (VLIW) Application Specific Instruction Processors (ASIPs) [12]. Efficiency might mean high throughput, reduced code size in the case of embedded applications, reduced energy consumption for portable devices, or simply relate to architectures that spend their silicon area wisely, so as to decrease power dissipation and cost. In this paper we focus on processors *specialized* to meet the throughput requirements of computationally intensive applications.

Since the time-critical loops of such applications often include *conditional constructs/branches*, predication combined with compiler-directed speculation is an effective approach to achieve increased ILP [3]. Predication allows one to concurrently schedule alternative paths of execution, with only the paths corresponding to the realized flow of control actually being allowed to modify the state of the processor. However, on its own, this technique may not lead to significant performance gains [3]. Speculation via predicate promotion enables one to execute operations on alternative control paths, prior to knowing which branches are to be taken, and later commit correct values. Performance gains are more significant when these two techniques are combined [3].

In order to maximize throughput, compiler transformations should be used to extract significant amounts of ILP. In turn, to take advantage of such ILP, datapaths with a large number of processing resources, e.g., VLIW processors, are required. A basic VLIW datapath might be based on a *single* register file shared by all of its functional units (FUs). In this case, the central register file provides internal storage/switching of data among FUs, while a typically slower interconnect provides access to/from the memory system. Unfortunately, this simple organization does not scale well when a large number of FUs are required. Indeed, when N FUs are connected to a register file, the area of the register file, delay and power dissipation can grow by up to N^3 [8]. In short, as the number of FUs increases, internal storage and communication quickly become a dominant, if not prohibitive, cost factor. This poor scaling can be overcome by restricting the connectivity between FUs and registers, so that each FU can only read/write from/to a limited subset of registers [8]. In particular *clustered* VLIW processors can reap these benefits by including multiple clusters of FUs connected to local storage (the cluster's register file). Although the move from a centralized to a distributed register file organization can achieve significant *delay*, *power* and *area* savings, there is a potential downside. Indeed, one may have to transfer/copy data among register files (i.e., datapath clusters), possibly resulting in increased *latency*, i.e., requiring additional scheduling steps. From the point of view of throughput, the tradeoffs are as follows. A datapath including several small (in number of FUs) clusters could operate at a higher clock rate, but might incur higher latency penalties due to additional switching operations among clusters. Of course, the potential downside

[*] This work is supported in part by an NSF ITR Grant ACI-0081791 and NSF Grant CCR-9901255 and by Grant ATP-003658-0649 of the Texas Higher Education Coordinating Board.

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.

associated with switching costs on clustered machines may not adversely impact throughput, since higher clock rates may permit faster execution.

The central and novel idea in this paper is *predicated switching* and its application in the context of clustered VLIW machines. As will be seen, predicated switching is analogous to standard predication in that it converts a program's control flow into data flow. However, in contrast to standard predication, in predicated switching the flow of control is realized through predicated switching operations whereby *predicated moves* select which among a set of results/values from alternative execution paths to place at a prespecified destination, correctly modifying the processor state. We formalize this conversion by proposing a compiler transformation to generate predicated switched code. The proposed transformation has the potential to efficiently exploit the hierarchical storage/switching resources in a clustered datapath. In particular, it enables aggressive speculative execution, while leveraging the penalties associated with transferring data across clusters to realize the code's flow of control.

In summary with increased ILP we expect to need (1) datapaths with large numbers of functional units, and (2) large register files to hold the associated increased number of live variables. Both of these suggest the need to partition register space across clustered functional units, i.e., include hierarchies of storage/switching resources, and the need for compiler techniques that are suitable for such datapaths – this is the focus of this paper. In Section 2, we briefly discuss standard predication and speculation techniques. In Sections 3 and 4 we present *predicated switching*, by introducing a novel compiler transformation, SSA-PS, that generates predicated switched code, and discussing its suitability in the context of clustered machines. In Section 5 we contrast our work with recent work on both predication and speculation for VLIW EPIC machines. In Section 6 we include experimental results validating the advantages of predicated switching on clustered machines. Conclusions are presented in Section 7.

2. BACKGROUND ON STANDARD PREDICATION

Predication is an ILP enhancing technique that exposes the hardware resources to multiple execution paths. Specifically, the basis for standard predication, *if-conversion* [5], transforms conditional branches into (1) operations that define predicates, and (2) guarded operations corresponding to alternative control paths. A guarded operation has a predicate associated with it, and is committed only if the predicate is true. In this sense, if-conversion is said to convert control dependences into data dependences (on predicate values), generating what is called a *hyperblock* [3]. Figure 1.a shows a sample code segment including a conditional statement, to be discussed below. The basic blocks associated with this code segment have been labeled A-D and the associated control flow graph is shown in Figure 1.b. Figure 1.d presents the scheduled standard predicated code for our example.[1] Throughout this paper, we shall assume that all operations take one cycle, except for reads from memory which take two cycles. Predicated operations are indicated by appending $< p >$ to represent the associated guards. Flexible predicate assignment types are used in this code, including the ut and uf, unconditional true and false types. For e.g., in Figure 1, $p1\_ut = (cond > 5)$ sets $p1$ to true if $(cond > 5)$ and false otherwise, while $p2\_uf$ does the opposite. (See [3] for details on more types of predicate define operations.) Note that once the predicate $p1$ (and its complement $p2$) are computed, they can serve as guards for the operations in basic blocks B and C. More complex flow of control can similarly be replaced by a sequence of predicated code.

Predicate promotion refers to speculation performed by changing a micro-instruction's predicate to a predicate whose expression subsumes that of the original predicate [3]. Specifically, if a predicated operation does not modify the processor state (i.e., does not change a value of a program variable), then it can be moved up to the point where its operand(s) are *uniquely* defined. In this case the predicate of the operation becomes that of the region where its operands are defined. When the new predicate differs from the original it is said to have been *promoted*, and the operation is speculatively executed. Predicate promotion is thus a form of compiler-directed speculation.

Note that, for the example in Figure 1, no predicate promotion was performed on the operations in Blocks B and C. Indeed the operations in each block directly modify the program variables x and y, or compute the variable $final$ based on x and y, and thus must be executed after their definition. Accordingly, the resulting code has no speculated operations – see Figure 1.d. In practice, however, opportunities for speculation (predicate promotion) occur quite frequently (e.g., in the Mediabench benchmark). In fact, this form of speculation has been shown to be effective, while avoiding the code explosion problems associated with other compiler-directed speculation techniques, see e.g., [10, 13, 15]. Space precludes us from discussing generic criteria used in selecting the code segments to be predicated. For details we refer the reader to [3]. In the sequel we will refer to if-conversion based predication and speculation techniques as *standard predication.*

3. PREDICATED SWITCHING TRANSFORMATION – SSA-PS

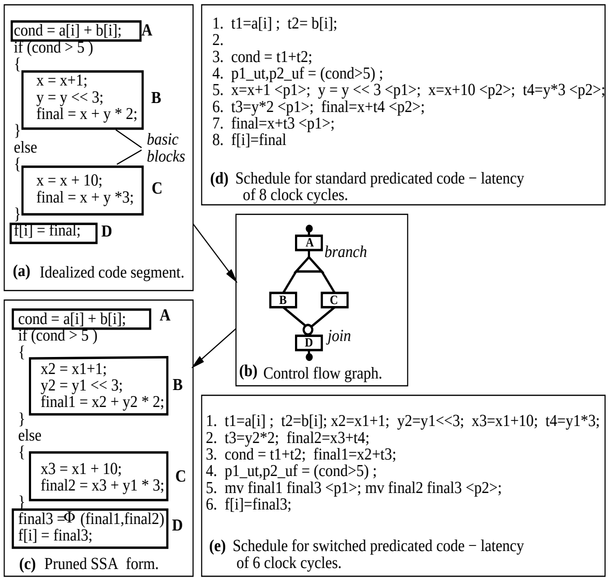
The proposed *Static Single Assignment - Predicated Switching* (SSA-PS) transformation is based on the well known SSA conversion, see e.g., [7]. We begin by briefly reviewing SSA using the code segment shown in Figure 1.a.

3.1 SSA transformation

The defining characteristic of a program in SSA form is that each variable is the target of exactly *one* assignment statement. Transforming a program into SSA form keeps the same flow of control but includes: (1) functions to reconcile multiple assignments that reach a join point in the control flow; and (2) variable renaming to ensure that the single assignment property is satisfied.

Thus, the SSA transformation involves two steps: identifying the placement of the ϕ functions in the code and renaming variables appropriately. This is a relatively straightforward process. For example, consider the variable $final$ in our sample code. Assignments to $final$ are made in Blocks B and C, thus a ϕ function is inserted at the join point

[1] Note that a compiler breaks a complex expression such as $final = x + y * 2$ using temporary variables such as $t4$.



```
cond = a[i] + b[i];      A
if (cond > 5 )
{
    x = x+1;
    y = y << 3;          B
    final = x + y * 2;
}                          basic
else                       blocks
{
    x = x + 10;          C
    final = x + y *3;
}
f[i] = final;    D
```

(a) Idealized code segment.

(b) Control flow graph.

```
cond = a[i] + b[i];      A
if (cond > 5 )
{
    x2 = x1+1;
    y2 = y1 << 3;             B
    final1 = x2 + y2 * 2;
}
else
{
    x3 = x1 + 10;             C
    final2 = x3 + y1 * 3;
}
final3 =Φ (final1,final2)     D
f[i] = final3;
```

(c) Pruned SSA form.

```
1.  t1=a[i] ;  t2= b[i];
2.
3.  cond = t1+t2;
4.  p1_ut,p2_uf = (cond>5) ;
5.  x=x+1 <p1>;  y = y << 3 <p1>;  x=x+10 <p2>;  t4=y*3 <p2>;
6.  t3=y*2 <p1>;  final=x+t4 <p2>;
7.  final=x+t3 <p1>;
8.  f[i]=final
```

(d) Schedule for standard predicated code − latency of 8 clock cycles.

```
1.  t1=a[i] ;  t2=b[i]; x2=x1+1;  y2=y1<<3;  x3=x1+10;  t4=y1*3;
2.  t3=y2*2;  final2=x3+t4;
3.  cond = t1+t2;  final1=x2+t3;
4.  p1_ut,p2_uf = (cond>5) ;
5.  mv final1 final3 <p1>; mv final2 final3 <p2>;
6.  f[i]=final3;
```

(e) Schedule for switched predicated code − latency of 6 clock cycles.

Figure 1: Sample code segment(a), its control flow graph(b), its pruned SSA form(c), standard predicated code schedule(d) and switched predicated code schedule(e).

prior to Block D.[2] The renaming process involves giving distinct names to each assignment made to a variable, including those made by the ϕ functions. We denote renamed versions of a variable, say $final$, by $final1, final2, etc.$ For the example at hand, $final$ is renamed $final1$ in Block B, $final2$ in Block C and $final3 = \phi(final1, final2)$ for the ϕ function at the join point. If the variable $final$ were not subsequently used, i.e., written to memory, then ensuring a unique assignment from the point of view of subsequent basic blocks is unnecessary. Thus, as shown in Figure 1.c, the ϕ functions associated with the variables x and y can be pruned – such code is said to be in the *pruned* SSA form [4]. The role of the ϕ functions is to realize conditional assignments, depending on which control path is followed to the renamed variable. From a compilation point of view, the SSA form eliminates "false data dependences" by introducing additional variable names. Indeed, for our example, operations in basic Blocks B and C can now be speculated, and scheduled concurrently, as long as the ϕ function is realized thereafter, to guarantee that a correct assignment is eventually made to $final3$.

3.2 SSA-PS transformation

The idea underlying the SSA-PS transformation is to realize the conditional assignments corresponding to ϕ functions via *predicated switching* operations, in particular predicated *move* operations. Figure 1.e shows how this would be done for our example – the ϕ function resulting from converting our sample code into its pruned SSA form, i.e., $final3 = \phi(final1, final2)$ is realized through two conditional moves $mv\ final1\ final3 < p1 >$ and $mv\ final2\ final3 < p2 >$. Alternatively, a more 'efficient' realization can be obtained by optimizing the variable renaming process, but violating SSA's requirements. For example, the register associated with $final3$ could be the same as that of $final1$. In this case, upon exiting the conditional branch we would have $final1 = \phi(final1, final2)$ which can be realized by a *single* predicated move $mv\ final2\ final1 < p2 >$. This reduces both the number of moves and variables in the code. A simple post-processing step can be carried out to perform this optimization.

For our example, predicated switched code takes 6 steps to complete (Figure 1.e) while standard predication takes 8 steps (Figure 1.d), i.e., a 25% improvement in performance. The proposed conversion to predicated switched code not only transforms control flow into data flow but, through variable renaming, enables additional speculative execution, which, in this case, leads to a performance advantage.

Predicated switching code execution on clustered machines requires an instruction set supporting three types of predicated moves: *internal*, *external*, and *logical* moves. A predicated internal move operation, specified by $mvI\ src\ dst < p >$ copies the value in the source register src into the destination register dst, if the predicate p is TRUE. Both register locations belong to the same local register file associated with a given cluster. External moves, $mvE\ src\ dst\ \ < p >$, have the same functionality except that the registers src and dst belong to register files on different clusters. In addition, in some cases it can be advantageous to support logical moves (mvL), whereby a value res computed by a given operation is conditionally placed in a second location dst (on the same register file) depending on a predicate p, e.g., $op\ o1\ o2\ res$, $mvL\ res\ dst\ \ < p >$. In such cases, we can realize the same functionality by collapsing both operations into a predicated operation $op\ o1\ o2\ dst\ \ < p >$. (Note however that such a logical move can only be inserted on the same step as the operation op if the predicate p is available prior to execution.) For a concrete example see the operations in bold in Figures 2.e, 2.f and 2.g.

We shall briefly present the main steps of an algorithm that generates SSA-PS code. We focus on how predicated move operations are inserted to realize the ϕ functions resulting from the pruned SSA transformation. The code segment in Figure 2.a is used to illustrate the process.

Step 1 Obtain the pruned SSA form for the code. Generate the required predicate define operations and generic predicated moves realizing ϕ functions, see Figure 2.b.

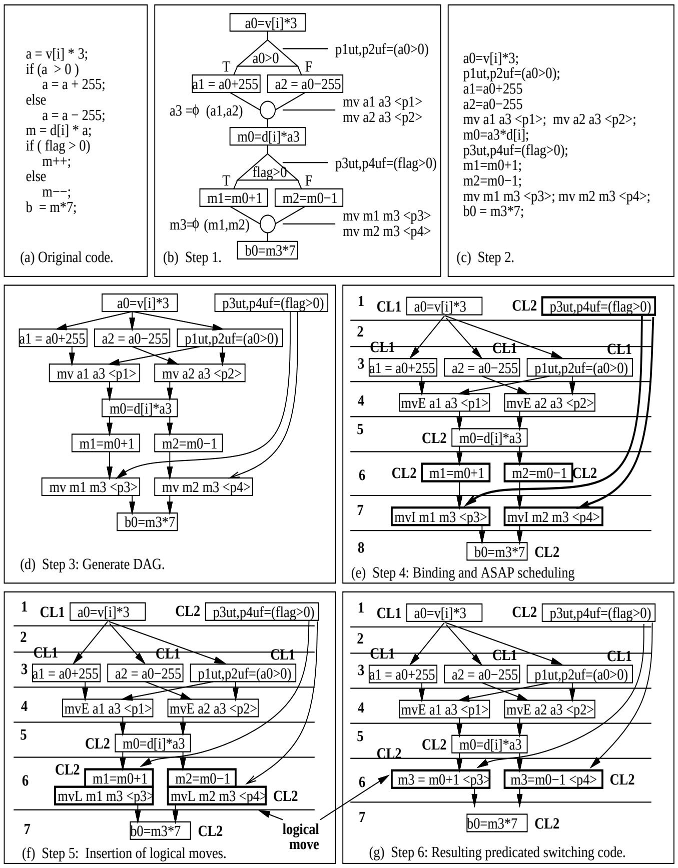
Step 2 Remove the program's flow of control and insert the predicate define and predicated mv operations, see Figure 2.c.

Step 3 Using data dependence analysis, generate a DAG for the code's operations, see Figure 2.d.

Step 4 Bind operations to clusters and annotate the DAG with the cluster IDs. Transform generic moves to either internal or external, as appropriate. Perform an ASAP (as soon as possible) scheduling of the DAG to determine which internal moves might be realized as logical moves, see Figure 2.e.

Step 5 Transform all mvI operations whose predicates are defined (or available) prior to the completion of the operation defining the value (to be moved) into logical

[2] Efficient algorithms for determining a *minimal* number of locations for ϕ functions are discussed in [7].



```
a = v[i] * 3;
if (a  > 0 )
    a = a + 255;
else
    a = a − 255;
m = d[i] * a;
if ( flag > 0)
    m++;
else
    m−−;
b  = m*7;
```

(a) Original code.

(b) Step 1.

```
a0=v[i]*3;
p1ut,p2uf=(a0>0);
a1=a0+255
a2=a0−255
mv a1 a3 <p1>;  mv a2 a3 <p2>;
m0=a3*d[i];
p3ut,p4uf=(flag>0);
m1=m0+1;
m2=m0−1;
mv m1 m3 <p3>; mv m2 m3 <p4>;
b0 = m3*7;
```

(c) Step 2.

(d) Step 3: Generate DAG.

(e) Step 4: Binding and ASAP scheduling

(f) Step 5: Insertion of logical moves.

(g) Step 6: Resulting predicated switching code.

Figure 2: Algorithm to obtain SSA-PS code.

moves, mvL, and generate corresponding predicated operations, see Figures 2.f and 2.g.

The resulting predicated switching code is shown in Figure 2.g.

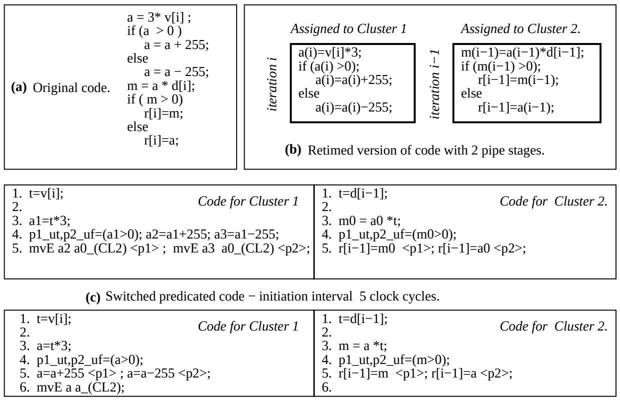
The proposed SSA-PS transformation generates code with predicated switching operations, while all other operations are speculatively executed. Due to their high energy consumption, speculating read/writes from/to memory may be undesirable. When required, the compiler can perform a simple pre-processing step to identify such operations, isolate them via partial replication of conditional constructs, and then mark the selected conditionals to be preserved during the SSA-PS transformation. Possible conditional jumps (exiting a hyperblock) are treated similarly. Micro-architectural support for predicated switching code is identical to that required by standard predicated code(for details on the IMPACT EPIC architecture, see [3]).

4. PREDICATED SWITCHING AND CLUSTERED MACHINES

It can be shown that the initiation interval (throughput) of loops implemented as predicated switching code can never be worse than that of standard predicated and speculated code on unbounded datapaths.

FACT 1. *Consider a (clustered) datapath with unbounded resources (FUs and register files) such that predicate define and internal move operations both take the same amount of time to execute, say 1 step. Suppose the standard and switched predicated code for a hyperblock are generated such that functionally equivalent operations (with possibly renamed variables) are bound to the same clusters. Then the latency for switched predicated code will never exceed that of standard predicated code combined with predicate promotion.*

See [12] for a formal proof of FACT 1. The question remains as to how frequently predicated switching is better, and by how much.



(a) Original code.

```
a = 3* v[i] ;
if (a  > 0 )
    a = a + 255;
else
    a = a − 255;
m = a * d[i];
if ( m > 0)
    r[i]=m;
else
    r[i]=a;
```

Assigned to Cluster 1
Assigned to Cluster 2.
iteration i
iteration i−1

```
a(i)=v[i]*3;
if (a(i) >0);
    a(i)=a(i)+255;
else
    a(i)=a(i)−255;
```

```
m(i−1)=a(i−1)*d[i−1];
if (m(i−1) >0);
    r[i−1]=m(i−1);
else
    r[i−1]=a(i−1);
```

(b) Retimed version of code with 2 pipe stages.

Code for Cluster 1

```
1.  t=v[i];
2.
3.  a1=t*3;
4.  p1_ut,p2_uf=(a1>0); a2=a1+255; a3=a1−255;
5.  mvE a2 a0_(CL2) <p1> ;  mvE a3  a0_(CL2) <p2>;
```

Code for Cluster 2.

```
1.  t=d[i−1];
2.
3.  m0 = a0 *t;
4.  p1_ut,p2_uf=(m0>0);
5.  r[i−1]=m0  <p1>; r[i−1]=a0 <p2>;
```

(c) Switched predicated code − initiation interval 5 clock cycles.

Code for Cluster 1

```
1.  t=v[i];
2.
3.  a=t*3;
4.  p1_ut,p2_uf=(a>0);
5.  a=a+255 <p1> ; a=a−255 <p2>;
6.  mvE a a_(CL2);
```

Code for Cluster 2.

```
1.  t=d[i−1];
2.
3.  m = a *t;
4.  p1_ut,p2_uf=(m>0);
5.  r[i−1]=m  <p1>; r[i−1]=a <p2>;
6.
```

(d) Standard predicated code − initiation interval 6 clock cycles.

Figure 3: Example switched and standard predicated code on a clustered datapath.

As seen in the example in Figure 1, predicated switching can lead to performance improvements over standard predication for centralized machines. It is however on clustered VLIW machines that SSA-PS proves to provide more aggressive and *consistent* performance gains. Consider the illustrative code shown in Figure 3.a and the software pipelined (retimed) version, with increased ILP, shown in Figure 3.b. (The retiming function was selected to cut the initiation interval by about a half.) Assuming each pipe stage is assigned to a different cluster of the machine, Figures 3.c and 3.d show the resulting optimal schedules (and corresponding initiation intervals) achieved by switching and standard predicated code. The standard predicated code incurs a 20 % performance penalty when compared to predicated switching code. Below we discuss why this is the case, and why one might expect such gains to be consistent.

For this example, the predicated switched code speculates the two conditional updates to the renamed variable a on Step 4, and then moves the correct value to Cluster 2 on Step 5, see Figure 3.c. The predicated external move operations thus realize both the *required ϕ function* and the *required data transfer* from Cluster 1 to 2.[3] By contrast, for the standard predicated code shown in Figure 3.d, predicate promotion cannot be applied to the conditional updates of a alluded to above. Accordingly, the predicated updates of a can only be scheduled after the predicates $p1, p2$ are defined. Moreover, after these are carried out, the external

[3] Note that in this example the write operations on Step 5 of Cluster 2 are not speculated in the predicated switching code. This is done by using the pre-processing step alluded to in Section 3.2.

move from Cluster 1 to 2 must be *appended* to the code, which delays the schedule.

In summary the main advantage of switched over standard predicated code on *clustered* machines is the leveraging of *required* data transfer (move) operations across clusters to realize the *flow of control* (i.e., ϕ functions). Thus we argue that SSA-PS is more 'cluster friendly' than if-conversion. The test cases considered in Section 6 demonstrate that such opportunities occur frequently in real application's code, enabling predicated switching to achieve (usually quite significant) performance gains on clustered machines.

5. RELATED WORK

This paper proposes the SSA-PS transformation, an aggressive form of compiler-directed speculation targeting clustered machines. In Section 4, we argued that the SSA-PS transformation is more cluster friendly (i.e., typically achieves superior performance) than standard predication, and pinpointed the rationale for this. Experimental evidence in Section 6 supports these claims for a number of time-critical loops extracted from multimedia benchmarks [2]. Below we discuss other related work.

There has been some work on the use of conditional moves to achieve performance enhancements, e.g. see [9, 14, 1]. The contribution most relevant to this paper is [9], where the authors discuss support for predication on machines that only support *conditional moves.* The code for machines that only support conditional move operations is generated by first performing *if-conversion* of the original code, and then by substituting each predicated operation by a *sequence* of micro-instructions with equivalent functionality. This transformation is highly inefficient, in that it may introduce large chains of extra data dependences. Accordingly, significant performance degradation is reported with respect to standard predication [9].

More traditional compiler-directed speculation techniques, such as [10, 13, 15, 11] can only reliably improve performance when there is a significant bias on execution paths, and typically lead to code explosion.

In [6] the authors propose a predicated single static assignment conversion to enable aggressive speculation on VLIW machines. Unfortunately, the proposed transformation leads to code explosion – a problem predication aimed to circumvent in the first place. In simple terms, they use the SSA transformation for renaming variables, but rather than reconciling alternative variable values (i.e., implementing the ϕ function) at join points, the transformation merely replicates all the code that follows the join point. Specifically, replication occurs as many times as the number of distinct control paths in the program reaching a particular join point. Then, the transformation computes and associates predicates with each replica, so as to determine which execution path would be committed. It should be clear that such increases in code size, along with added complexity of predicate define operations, are highly undesirable. In contrast, the key aspects of the proposed SSA-PS transformation is a careful realization of ϕ functions and the leveraging of external moves on clustered machines.

6. EXPERIMENTAL RESULTS

In this section we provide experimental evidence that the SSA-PS transformation can improve on the state-of-the-art for *clustered* VLIW machines executing high-throughput applications. We will consider idealized machines with one, two and three clusters, denoted C, 2CL, and 3CL respectively, where C corresponds to a *centralized* machine. In order to assess the maximum achievable performance, we assume that an unlimited number of micro-instructions can be issued at each clock cycle by each cluster. For scheduling purposes, we assume that read operations take 2 cycles, while arithmetic/logic and write, and move micro-operations take 1 cycle. We denote machines supporting predicated switching by PS and those supporting standard predication by P. Thus, for example, a two cluster machine supporting predicated switching is denoted by PS/2CL.

Table 1 summarizes our results for a number of time-critical loops extracted from the Mediabench benchmarks [2]. As shown in the table, various retiming functions were applied to the sample code segments - in each case, the goal was to reduce the initiation interval to a target percentage of the original (non-pipelined/retimed) code. The predicated switching and standard predicated code resulting from the retimed versions of the original source code were then independently optimized, so as to maximize performance (i.e., minimize initiation interval), and minimize data transfers among clusters, while balancing the load among clusters. For each experiment, we report the achieved initiation interval, II, and MI, the maximum number of operations issued by any cluster on any cycle of the corresponding schedule. (Note that MI excludes predicate define operations, since those are performed by distinct FUs.) We also compute the maximum number of live variables ML on any cluster for the schedule. The table shows the absolute performance achieved, and the relative performance/cost degradation incurred by standard predication/speculation versus predicate switching executing the same code segment.

The experiments that were conducted are numbered and grouped into sets labelled A, B and C. Experiment 1 in Group A illustrates a case where standard predication results in a 6.6 % performance degradation as compared to predicated switching, executing on the same *centralized* machine.

The experiments in Group B compare the performance of standard predication with predicated switching, when executing on the same *clustered* machine. We see a consistent performance degradation, up to 33 %, with an average of 17.7 %. It is important to note that the code segments for these cases correspond to retimed code exhibiting high ILP, where it is expected clustered machines will be of interest.

The experiments in Group C show that, for most cases, predicated switching achieves a performance boost by leveraging external moves. Specifically, the standard predicated code in those experiments was executed on a centralized machine (thus eliminating the need for required external moves on the critical path), or on a machine with fewer clusters, again to avoid such moves. By avoiding external move operations, in all cases standard predication was able achieve the same initiation interval as predicated switching code executing on a clustered machine. However, as shown in the table, the use of more 'centralized' machines (by standard predication) comes at a cost. Namely, there is an increase (up to 80 % with an average of 59 %) in the maximum number of instructions issued by any given cluster on any given scheduling step – this directly impacts the number of FUs / register file ports required on the largest cluster. Simi-

Mediabench Benchmarks	Retiming Function	Machine	Results	Case
adpcm: Main inner loop body of **adpcm-decoder()** (file: adpcm.c).	none	PS/C P/C	$II_{PS} = 15$, $II_P = 16$ $(II_P - II_{PS})/II_{PS}$ =**6.6%**	1 (A)
	2 pipe stages	PS/2CL P/2CL	$II_{PS} = 8$, $II_P = 9$ $(II_P - II_{PS})/II_{PS}$ =**12.5 %**	2 (B)
		PS/2CL P/C	$II_{PS} = 8$, $II_P = 8$ $(II_P - II_{PS})/II_{PS}$ =**0%** $(MI_P - MI_{PS})/MI_{PS}$ =**50%** $(ML_P - ML_{PS})/ML_{PS}$ =**22%**	3 (C)
rasta: Main inner loop body of **eliminate()** (file: lqsolve.c).	2 pipe stages	PS/2CL P/2CL	$II_{PS} = 4$, $II_P = 5$ $(II_P - II_{PS})/II_{PS}$ =**25%**	4 (B)
		PS/2CL P/C	$II_{PS} = 4$, $II_P = 4$ $(II_P - II_{PS})/II_{PS}$ =**0%** $(MI_P - MI_{PS})/MI_{PS}$ =**80%** $(ML_P - ML_{PS})/ML_{PS}$ =**16.6%**	5 (C)
	3 pipe stages	PS/2CL P/2CL	$II_{PS} = 3$, $II_P = 4$ $(II_P - II_{PS})/II_{PS}$ =**33.3%**	6 (B)
		PS/2CL P/C	$II_{PS} = 3$, $II_P = 3$ $(II_P - II_{PS})/II_{PS}$ =**0%** $(MI_P - MI_{PS})/MI_{PS}$ =**57%** $(ML_P - ML_{PS})/ML_{PS}$ =**33.3%**	7 (C)
mpeg2enc: Main inner loop body of **iquant1-intra()** (file: quantize.c).	2 pipe stages	PS/2CL P/2CL	$II_{PS} = 7$, $II_P = 7$ $(II_P - II_{PS})/II_{PS}$ =**0%**	8 (B)
	3 pipe stages	PS/3CL P/3CL	$II_{PS} = 5$, $II_P = 6$ $(II_P - II_{PS})/II_{PS}$ =**20%**	9 (B)
		PS/3CL P/2CL	$II_{PS} = 5$, $II_P = 5$ $(II_P - II_{PS})/II_{PS}$ =**0%** $(MI_P - MI_{PS})/MI_{PS}$ =**50%** $(ML_P - ML_{PS})/ML_{PS}$ =**66%**	10 (C)
mpeg2enc: Main inner loop body of **iquant1-non-intra()** (file: quantize.c).	3 pipe stages	PS/3CL P/3CL	$II_{PS} = 6$, $II_P = 7$ $(II_P - II_{PS})/II_{PS}$ =**16.6%**	11 (B)
		PS/2CL P/2CL	$II_{PS} = 6$, $II_P = 7$ $(II_P - II_{PS})/II_{PS}$ =**16.6%**	12 (B)

Table 1: Experimental results.

larly, in all cases there is a significant increase (up to 66 %, with an average of 44 %) in the maximum number of live variables on any *given* register file, which directly impacts register file size requirements. As discussed in the introduction, machines with smaller clusters are likely to sustain a higher clock rate, thus although the initiation interval is the same for the experiments in Group C, it is likely that the performance of predicated switching code on a clustered machine would be superior to that of standard predication on a more centralized machine.

Finally note that, although the initiation interval is the same for the predicated switching machines with 3 and 2 clusters in experiments 11 and 12, these cases are still interesting, since the machine with 3 clusters might be able to run at a faster clock rate due to its reduced issue width requirements. Such trade-offs should be carefully explored when specializing a VLIW clustered machine (with support for predicated switching) to specific target applications.

7. CONCLUSIONS

In this paper, we proposed a novel compiler transformation (SSA-PS) to generate predicated switching code. We argued that, on clustered VLIW machines, predicated switching execution compares favorably with state-of-the-art predicated execution, since it enables aggressive speculation while leveraging the penalties associated with inter-cluster communication. Experimental results for multimedia benchmarks are presented, demonstrating that in practice such leveraging leads to significant performance gains. Given these encouraging preliminary results, we are are currently working on software pipelining and binding algorithms suitable for clustered VLIW machines with support for predicated switching execution.

8. REFERENCES

[1] Artur Klauser et. al. Dynamic Hammock Predication for Non-predicated Instruction Set Architectures. In *PACT*, 1998.

[2] C.Lee et. al. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *MICRO*, 1997.

[3] D.August et. al. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *ISCA*, 1998.

[4] J.-D.Choi et. al. Automatic construction of sparse data flow evaluation graphs. In *POPL*, 1991.

[5] J.R.Allen et. al. Conversion of control dependence to data dependence. In *POPL*, 1983.

[6] L.Carter et. al. Predicated static single assignment. In *PACT*, 1999.

[7] R.Cytron et. al. Efficiently computing static single assignment form and the control dependence graph. In *ACM Trans. on Prog. Lang. and Systems*, 1991.

[8] Scott Rixner et. al. Register Organization for Media Processing. In *HPCA*, 1999.

[9] S.Mahlke et. al. A comparison of full and partial predicated execution support for ILP processors. In *ISCA*, 1995.

[10] W.Chang et. al. Using profile information to assist classic code optimizations. In *Software Practice and Experience*, 1991.

[11] W.Hwu et. al. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. In *The Journal of Supercomputing*, Jan 1993.

[12] http://horizon.ece.utexas.edu/∼jacome/nova.

[13] J.A.Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. on Computers*, 1981.

[14] Rainer Leupers. Exploiting Conditional Instructions in Code Generation for Embedded VLIW Processors. In *DATE*, 1999.

[15] P.Chang and W.Hwu. Trace Selection for compiling Large C Application Programs to Microcode. In *Intl. Workshop on Microprogramming and Microarchitecture*, 1988.