Compiler-Directed ILP Extraction for Clustered VLIW/EPIC machines: Predication, Speculation and Modulo Scheduling *

Satish Pillai
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78705
satish@ece.utexas.edu

Margarida F. Jacome
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78705
jacome@ece.utexas.edu

Abstract

Compiler-directed ILP extraction techniques are critical to effectively exploiting the significant processing capacity of contemporaneous VLIW/EPIC machines. In this paper we propose a novel algorithm for ILP extraction *targeting* clustered *EPIC machines that integrates three powerful techniques: predication, speculation and modulo scheduling. In addition, our framework schedules and binds operations, generating actual VLIW code. To the best of our knowledge, there is no other algorithm in the literature on predicated code optimizations that jointly considers speculation and modulo scheduling in the context of clustered EPIC machines. Our experimental results show that by jointly considering different extraction techniques in a resource aware context, the proposed algorithm can take maximum advantage of the resources available on the clustered machine, aggressively improving performance.*

1 Introduction

Multimedia, communications and security applications exhibit a significant amount of instruction level parallelism(ILP). In order to meet the performance requirements of these demanding applications, it is important to use compilation techniques that expose/extract such ILP and processor datapaths with a large number of functional units, e.g., VLIW/EPIC processors.

A basic VLIW datapath might be based on a *single* register file shared by all of its functional units (FUs). Unfortunately, this simple organization does not scale well with the number of FUs [12]. *Clustered* VLIW datapaths address this poor scaling by restricting the connectivity between FUs and registers, so that an FU on a cluster can only read/write from/to the cluster's register file, see e.g. [12]. Since data may need to be transferred among the machine's clusters, possibly resulting in increased latency, it is important to develop performance enhancing techniques that take such data transfers into consideration.

Most of the applications alluded to above have only a few time critical kernels, i.e. a small fraction of the entire code (sometimes as small as 3%) is executed most of the time, see e.g. [16] for an analysis of the Mediabench[5] suite of programs. Moreover, most of the processing time is typically spent executing the 2 innermost loops of such time critical loop nests – in [16], for example, this percentage was found to be about 95% for Mediabench programs. Yet another critical observation made in [16] is that there exists considerably high control complexity within these loops. This strongly suggests that in order to be effective, ILP extraction targeting such time critical inner loop bodies must handle control/branching constructs.

The key contribution of this paper is a novel resource-aware algorithm for compiler-directed *ILP extraction* targeting *clustered* EPIC machines that integrates three powerful ILP extraction techniques: predication, control speculation and software pipelining/modulo scheduling. In addition to extracting ILP from time-critical loops, our framework schedules and binds the resulting operations, generating actual VLIW code.

Background

The performance of a loop is defined by the average rate at which new loop iterations can be started, denoted initiation interval (II). Software pipelining is an ILP extraction technique that retimes [20] loop body operations (i.e., overlaps multiple loop iterations), so as to enable the generation of more compact schedules [18][3]. Modulo scheduling algorithms exploit such technique during scheduling, so as to expose additional ILP to the datapath resources, and thus decrease the initiation interval of a loop [23].

Predication allows one to concurrently schedule alternative paths of execution, with only the paths corresponding to the realized flow of control being allowed to actually modify the state of the processor. The key idea in predication is to eliminate branches through a process called *if-conversion* [8]. If-conversion, transforms conditional branches into (1) operations that define predicates, and (2) operations guarded by predicates, corresponding to alternative control paths.[1] A guarded operation is committed only if its predicate is true. In this sense, if-conversion is said to convert control dependences into data dependences (i.e., dependence on predicate values), generating what is

*This work is supported in part by an NSF ITR Grant ACI-0081791 and an NSF Grant CCR-9901255.

[1] Note that operations that define predicates may also be guarded.

called a *hyperblock* [11].

Control speculation "breaks" the *control dependence* between an operation and the conditional statement it is dependent on [6]. By eliminating such dependences, operations can be moved out of conditional branches, and can be executed before their related conditionals are actually evaluated. Compilers exploit control speculation to reduce control dependence height, which enables the generation of more compact, higher ILP static schedules.

Overview

The proposed algorithm iterates through three main phases: *speculation*, *cluster binding* and *modulo scheduling* (see Figure 3). At each iteration, relying on effective load balancing heuristics, the algorithm incrementally extracts/exposes additional ("profitable") ILP, i.e., speculates loop body operations that are more likely to lead to a decrease in initiation interval during modulo scheduling. The resulting loop operations are then assigned to the machine's clusters and, finally, the loop is modulo scheduled generating actual VLIW code. An additional critical phase (executed right after binding) attempts to leverage required data transfers across clusters to realize the speculative code, thus minimizing their potential impact on performance(see details in Section 4).

To the best of our knowledge, there is no other algorithm in the literature on compiler-directed predicated code optimizations that jointly considers control speculation and modulo scheduling in the context of clustered EPIC machines. In the experiments section we will show that, by jointly considering speculation and modulo scheduling in a resource aware context, and by minimizing the impact in performance of data transfers across clusters, the algorithm proposed in this paper can dramatically reduce a loop's initiation interval with upto 68.2% improvement with respect to a baseline algorithm representative of the state of the art.

2 Optimizing Speculation

In this Section, we discuss the process of deciding which loop operations to speculate so as to achieve a more effective utilization of datapath resources and improve performance(initiation interval). The algorithm for optimizing speculation, described in detail in Section 3, uses the concept of load profiles to estimate the distribution of load over the scheduling ranges of operations, as well as across the clusters of the target VLIW/EPIC processor. Section 2.1 explains how these load profiles are calculated. Section 2.2 discusses how the process of speculation alters these profiles and introduces the key ideas exploited by our algorithm.

2.1 Load Profile Calculation

The load profile is a measure of resource requirements needed to execute a Control Data Flow Graph(CDFG) with a desirable schedule latency. To illustrate how the load profiles used in the optimization process are calculated, consider the CDFG in Figure 1. Note that, when predicated code is considered, the branching constructs actually correspond to the definition of a predicate(denoted *PD*), and the associated control dependence corresponds to a data dependence on the predicate values(denoted p and $p!$). The load profile is calculated for a given target profile latency(greater than the critical path of the CDFG).

First, As Soon As Possible(ASAP) and As Late As Possible(ALAP) scheduling is performed to determine the scheduling range of each operation. For the example, operation *a* in Figure 1 has a scheduling range of 2 steps (i.e. step 1 and step 2), while all other operations have a sheduling range of a single step. The mobility of an operation is defined as $\mu(op) = alap(op) - asap(op) + 1$ and equals 2 for operation *a*. Assuming that the probability of scheduling an operation at any time step in its time frame is given by a uniform distribution[22], the contribution to the load of an operation op at time step t in its time frame is given by $\frac{1}{\mu(op)}$. In Figure 1, the contributions to the load of all the addition operations (labelled *a*, *c*, *d*, *f*, *g* and *h*) is indicated by the shaded region to the right of the operations. To obtain the total load for type fu at a time step t, we sum the contribution to the load of all operations that are executable on resource type fu at time step t. In Figure 1, the resulting total *adder* load profile is shown.

The thick vertical line in the load profile plot is an indicator of the capacity of the machine's datapath to execute instructions at a particular time step. (In the example, we assume a datapath that contains 2 adders.) Accordingly, in step 1 of the load profile, the shaded region to the right of the thick vertical line represents an over-subscription of the adder datapath resource. This indicates that, in the actual VLIW code/schedule, one of the addition operations (i.e. either operation *a*, *d* or *g*) will need to be delayed to a later time step.

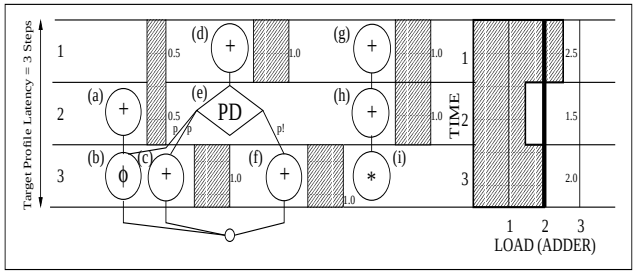

Figure 1. Adder Load Profile Calculation.

2.2 Load Balancing through Speculation

We argue that, in the context of VLIW/EPIC machines, the goal of compiler transformations aimed at speculating operations should be to "redistribute/balance" the load profile of the original/input kernel so as to enable a more effective utilization of the resources available in the datapath. More specifically, the goal should be to judiciously modify the scheduling ranges of the kernel's operations, via speculation, such that overall resource contention is

decreased/minimized, and consequently, code performance improved. We illustrate this key point using the example CDFG in Figure 2(a), and assuming a datapath with 2 adders, 2 multipliers and 1 comparator. Figure 2(b) shows the *adder* load profile for the original kernel and Figures 2(c) and (d) show the load profiles for the resulting CDFG's with one and three(all) operations speculated, respectively.[2][22] As can be seen, the load profile in Figure 2(c) has a smaller area above the line representing the datapath's resource capacity, i.e., implements a better redistribution of load and, as a result, allowed for a better schedule under resource constraints(only 3 steps) to be derived for the CDFG.[3] From the above discussion, we see that a

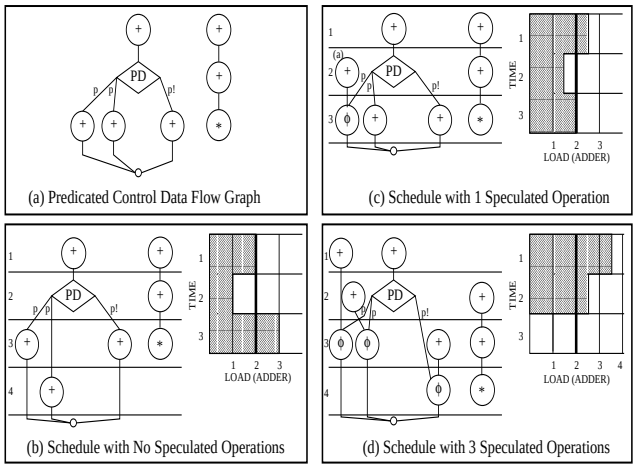

Figure 2. Performance vs Speculation: Resource Constrained Example.

technique to judiciously speculate operations is required, in order to ensure that speculation provides consistent performance gains on a given target machine. Accordingly, we propose an optimization phase, called OPT-PH (described in Section 3), which given an input hyperblock and a target VLIW/EPIC processor, possibly clustered, decides which operations should be speculated, so as to maximize the execution performance of the resulting VLIW code.

3 OPT-PH – An Algorithm for Optimizing Speculation

Our algorithm makes incremental decisions on speculating individual operations, using a heuristic ordering of nodes. The suitability of an operation for speculation is evaluated based on two metrics, Total Excess Load(TEL) and Speculative Mobility(*Spec_μ*), to be discussed below. Such metrics rely on a previously defined binding function(assigning operations to clusters), and on a target profile latency(see Section 2.1).

[2]Those load profiles were generated assuming a minimum target profile latency equal to the critical path of the kernel.

[3]The operations marked ϕ represent predicated reconciliation operations(see Section 4).

3.1 Total Excess Load(TEL)

In order to compute the first component of our ranking function, i.e. Total Excess Load(TEL), we start by calculating the load distribution profile for each cluster c and resource type fu, at each time step t of the target profile latency alluded to above. This is denoted by $Clust\_Load_{fu,c}(t)$. To obtain the cluster load for type fu, we sum the contribution to the load of all operations bound to cluster c(denoted by $nodes\_to\_clust(c)$) that are executable on resource type fu(denoted by $nodes\_on\_typ(fu)$) at time step t. Formally:

$$Clust\_Load_{fu,c}(t) = \sum_{\forall op \in S \text{ s.t. } t \in tf(op)} \frac{1}{\mu(op)}$$

where $S = nodes\_to\_clust(c) \bigcap nodes\_on\_typ(fu)$

$Clust\_Load_{fu,c}(t)$ is represented by the shaded regions in the load profiles in Figure 2.

Recall that our approach attempts to "flatten" out the load distribution of operations by moving those operations that contribute to greatest excess load to earlier time steps. To characterize Excess Load(EL), we take the difference between the actual cluster load, given above, and an ideal load, i.e.,

$$Diff_{fu,c}(t) = Clust\_Load_{fu,c}(t) - Ideal\_Load_{fu,c}$$

The ideal load, denoted by $Ideal\_Load_{fu,c}$, is a measure of the balance in load necessary to efficiently distribute operations both over their time frames as well as across different clusters. It is given by,

$$Ideal\_Load_{fu,c} = max\{Avg\_Load_{fu,c}, Clust\_Capacity_{fu,c}\}$$

where $Clust\_Capacity_{fu,c}$ is the number of resources of type fu in cluster c and $Avg\_Load_{fu,c}$ is the average cluster load over the target profile latency pr, i.e.,

$$Avg\_Load_{fu,c} = \frac{1}{pr} \sum_{t=1}^{pr} Clust\_Load_{fu,c}(t)$$

As shown above, if the resulting average load is smaller than the actual cluster capacity, the ideal load value is upgraded to the value of the cluster capacity. This is performed because load unbalancing per se is not necessarily a problem unless it leads to over-subscription of cluster resources. The average load and cluster capacity in the example of Figure 2 are equal and, hence, the ideal load is given by the thick vertical line in the load profiles.

The excess load associated with operation op, $EL(op)$, can now be computed, as the difference between the actual cluster load and the ideal load over the time frame of the operation, with negative excess loads being set to zero, i.e.,

$$EL(op) = \sum_{t=asap(op)}^{alap(op)} max\{0, Diff_{typ(op),clust(op)}(t)\}$$

where operation op is bound to cluster $clust(op)$ and executes on resource type $typ(op)$. In the load profiles of Figure 2, excess load is represented by the shaded area to the right of the thick vertical line. Clearly, operations with high excess loads are good candidates for speculation, since such speculation would reduce resource contention at their current time frames, and may thus lead to performance improvements. Thus, EL is a good indicator of the relative suitability of an operation for speculation.

However, speculation may be also beneficial when there is no resource over-subscription, since it may reduce the CDFG's critical path. By itself, EL would overlook such opportunities. To account for such instances, we define a second suitability measure, which "looks ahead" for empty scheduling time slots that could be occupied by speculated operations. Accordingly, we define Reverse Excess Load to characterize availability of free resources at earlier time steps to execute speculated operations:

$$REL(op) = -\sum_{t=1}^{asap(op)-1} min\{0, Diff_{typ(op),clust(op)}(t)\}$$

This is shown by the unshaded regions to the left of the thick vertical line in the load profiles of Figure 2.

We sum both these quantities and divide by the operation mobility to obtain Total Excess Load per scheduling time step.

$$TEL(op) = \frac{EL(op) + REL(op)}{\mu(op)}$$

3.2 Speculative Mobility($Spec\_\mu$)

The second component of our ranking function, denoted $Spec\_\mu(op)$, is an indicator of the number of additional time steps made available to the operation through speculation. It is given by the difference between the maximum ALAP value over all predecessors of the operation, denoted by pred(op), before and after speculation. Formally:

$$Spec\_\mu(op) = max_{\forall n \in pred(op)} alap(n) - max_{\forall n \in pred(op) \text{ s.t. } n \neq \text{cond. op}} alap(n)$$

3.3 Ranking Function

The composite ordering of operations by suitability for speculation, called $Suit(op)$, is given by:

$$Suit(op) = TEL(op) + Spec\_\mu(op)$$

$Suit(op)$ is computed for each operation that is a candidate for speculation, and speculation is attempted on the operation with the highest value, as discussed in the sequel.

4 Optimization Framework

Figure 3 shows the complete iterative optimization flow of OPT-PH. During the initialization phase, if-conversion is applied to the original CDFG – control dependences are converted to data dependences by defining the appropriate predicate define operations and data dependence edges[4] and an initial binding is performed.

After the initialization phase is performed, the algorithm enters an iterative phase. First it decides on the next best candidate for speculation. The ranking function used during this phase has already been described in detail in Section 3. The simplest form of speculating the selected operation is by deleting the edge from its predecessor predicate define operation(denoted *predicate promotion* [6]). In certain cases, the ability to speculate requires renaming and creating a new successor predicated move operation for reconciliation(denoted SSA-PS [9]).

After speculation is done, binding is performed using a modified version of the binding algorithm proposed in [15], that accounts for our framework's ability to leverage data transfers across clusters. The next optimization phase applies the critical transformation of collapsing binding related move operations with reconciliation related predicated moves, see [9]. Finally a modulo scheduler schedules the resulting Data Flow Graph(DFG). A two level priority function that ranks operations first by lower alap and next by lower mobility is used by the modulo scheduler.

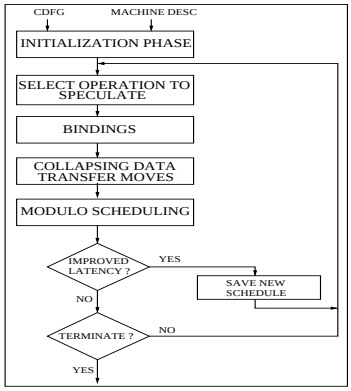

Figure 3. Overview of optimization flow.

If execution latency is improved with respect to the previous best result, then the corresponding schedule is saved. Each new iteration produces a different binding function that considers the modified scheduling ranges resulting from the operation speculated in the previous iteration. The process continues iteratively, greedily speculating operations, until the termination condition is satisfied. Currently this condition is simply a threshold on the number of successfully speculated operations, yet more sophisticated termination conditions can very easily be included.

5 Previous Work

We discuss below work that has been performed in the area of modulo scheduling and speculation. The method presented in [19] uses control speculation to decrease the initiation interval of modulo scheduled of loops in control-intensive non-numeric programs. However, since this

[4] The process of selecting the set of time-critical inner loop bodies to be optimized for any given application is beyond the scope of this paper – a good set of criteria can be found in [11].

method is not geared towards clustered architectures, it does not consider load balancing and data transfers across clusters. A modulo scheduling scheme is proposed in [24] for a specialized clustered VLIW micro-architecture with distributed cache memory. This method minimizes inter-cluster *memory* communications and is thus geared towards the particular specialized memory architecture proposed. It also does not explicitly consider conditionals within loops. The software pipelining algorithm proposed in [26] generates a near-optimal modulo schedule for all iteration paths along with efficient code to transition between paths. The time complexity of this method is, however, exponential in the number of conditionals in the loop. It may also lead to code explosion.

A state of the art static, compiler-directed ILP extraction technique that is particularly relevant to the work presented in this paper is standard (if-conversion based) predication with speculation in the form of predicate promotion[6](see Section 4), and will be directly contrasted to our work.

A number of techniques have been proposed in the area of high level synthesis for performing speculation. However, some of the fundamental assumptions underlying such techniques do not apply to the code generation problem addressed in this paper, for the following reasons. First, the cost functions used for hardware synthesis(e.g. [10, 13]), aiming at minimizing control, multiplexing and interconnect costs, are significantly different from those used by a software compiler, where schedule length is usually the most important cost function to optimize. Second, most papers (e.g. [14, 21, 4, 28, 27]) in the high level synthesis area exploit conditional resource sharing. Unfortunately, this cannot be exploited/accommodated in the actual VLIW/EPIC code generation process, because predicate values are unknown at the time of instruction issue. In other words, two micro-instructions cannot be statically bound to the same functional unit at the same time step, even if their predicates are known to be mutually exclusive, since the actual predicate values become available only after the *execute stage* of the predicate defining instruction, and the result (i.e. the predicate value) is usually forwarded to the write back stage for squashing, if necessary. Speculative execution is incorporated in the framework of a list scheduler by [7]. Although the longest path speculation heuristic proposed is effective, it does not consider load balancing, which is important for clustered architectures. A high-level synthesis technique that increases the density of optimal scheduling solutions in the search space and reduces schedule length is proposed in [25]. Speculation performed here, however, does not involve renaming of variable names and so code motion is comparatively restricted. Also there is no merging of control paths performed, as done by predication.

Certain special purpose architectures, like transport triggered architectures, as proposed in [1], are primarily programmed by scheduling data transports, rather than the CDFG's operations themselves. Code generation for such architectures is fundamentally different, and harder than code generation for the standard VLIW/EPIC processors assumed in this paper, see [2].

6 Experimental Results

Compiler algorithms reported in the literature either jointly address speculation and modulo scheduling but do not consider clustered machines, or consider modulo scheduling on clustered machines but cannot handle conditionals i.e., can only address straight code, with no notion of control speculation (see Section 5). Thus, the novelty of our approach makes it difficult to experimentally validate our results in comparison to previous work.

We have however devised an experiment that allowed us to assess two of the key contributions of this paper, namely, the proposed load-balancing-driven incremental control speculation and the phasing for the complete optimization problem. Specifically, we compared the code generated by our algorithm to code generated by a baseline algorithm that binds state of the art predicated code (with predicate promotion only)[6] to a clustered machine and then modulo schedules the code. In order to ensure fairness, the baseline algorithm uses the same binding and modulo scheduling algorithms implemented in our framework.

Kernel	Benchmark	Main Inner Loop from Function
Lsqsolve	Rasta/Lsqsolve	eliminate()
Csc	Mpeg	csc()
Shortterm	Gsm	Fast_Short_term_synthesis_filtering()
Store	Mpeg2dec	conv422to444()
Ford	Rasta	FORD1()
Jquant2	Jpeg	find_nearby_colors()
Huffman	Epic	encode_stream()
Add	Gsm	gsm_div()
Pixel1	Mesa	gl_write_zoomed_index_span()
Pixel2	Mesa	gl_write_zoomed_stencil_span()
Intra	Mpeg2enc	iquant1_intra()
Nonintra	Mpeg2enc	iquant1_non_intra()
Vdthresh8	TI suite	
Collision	TI suite	
Viterbi	TI suite	

Table 1. Kernel Characteristics

We present experimental results for a representative set of critical loop kernels extracted from the MediaBench[5] suite of programs and from TI's benchmark suite[17](see Table 1). The frequency of execution of the selected kernels for typical input data sets was determined to be significant. For example, the kernel Lsqsolve from the least squares solver in the rasta distribution, one of the kernels where OPT-PH performs well, takes more than 48% of total time taken by the solver. Similarly, Jquant, the function find_nearby_colors() from the Jpeg program, yet another kernel where OPT-PH performs well, takes more than 12% of total program execution time.

The test kernels were manually compiled to a 3-address like intermediate representation that captured all dependences between instructions. This intermediate representation together with a parametrized description of a clustered VLIW datapath was input to our automated optimization tool(see Figure 3). Four different clustered VLIW datapath configurations were used for the experiments. All the configurations were chosen to have relatively small clusters

since these datapath configurations are more challenging to compile to, as more data transfers may need to be scheduled. The FU's in each cluster include Adders, Multipliers, Comparators, Load/Store Units and Move Ports. The datapaths are specified by the number of clusters followed by the number of FU's of each type, respecting the order given above. So a configuration denoted $3C:|1|1|1|1|1|$ specifies a datapath with three clusters, each cluster with 1 FU of each type. All operations are assumed to take 1 cycle except Read/Write, which take 2 cycles. A move operation over the bus takes 1 cycle. There are 2 intercluster buses available for data transfers.

Kernel	3C: |1|1|1|1|1|			3C: |2|2|1|1|		
	Std. Pred.-MOD	OPT-PH-MOD	% Imp	Std. Pred.-MOD	OPT-PH-MOD	% Imp
Lsqsolve	4	3	25	3	3	0
Csc	12	12	0	12	12	0
Shortterm	5	4	20	4	4	0
Store	22	9	59.1	22	7	68.2
Ford	4	4	0	4	4	0
Jquant	31	28	9.7	31	26	16.1
Huffman	4	4	0	4	4	0
Add	2	2	0	2	2	0
Pixel1	5	4	20	5	4	20
Pixel2	4	3	25	4	3	25
Intra	8	6	25	8	6	25
Nonintra	9	7	22.2	6	6	0
Vdthresh8	4	4	0	4	4	0
Collision	7	7	0	7	7	0
Viterbi	24	10	58.3	8	8	0
Kernel	4C: |1|1|1|1|1|			4C: |2|2|1|1|		
	Std. Pred.-MOD	OPT-PH-MOD	% Imp	Std. Pred.-MOD	OPT-PH-MOD	% Imp
Lsqsolve	4	4	0	4	4	0
Csc	11	11	0	11	11	0
Shortterm	5	5	0	5	4	20
Store	8	7	12.5	7	7	0
Ford	12	9	25	12	9	25
Jquant	31	28	9.7	29	28	3.5
Huffman	4	4	0	4	4	0
Add	3	3	0	2	2	0
Pixel1	4	4	0	4	4	0
Pixel2	3	3	0	3	3	0
Intra	6	6	0	6	6	0
Nonintra	9	7	22.2	7	7	0
Vdthresh8	4	4	0	4	4	0
Collision	6	6	0	6	6	0
Viterbi	9	8	11.1	9	8	11.1

Table 2. Initiation Interval

Detailed results of our experiments (15 kernels compiled for 4 different datapath configurations) are given in Table 2. For every kernel and datapath configuration, we present the performance of modulo scheduled standard predicated code with predicate promotion(denoted Std. Pred-MOD) compared to that of code produced by our approach(denoted OPT-PH-MOD). As can be seen, our technique consistently produces shorter initiation intervals and gives large increases in performance, upto a maximum of 68.2%. This empirically demonstrates the effectiveness of the speculation criteria and load balancing scheme developed in the earlier Sections. The breaking of control dependences and efficient redistribution of operation load both over their time frames as well as across clusters, permits dramatic decreases in initiation interval.

Finally, although the results of our algorithm cannot be compared to results produced by high level synthesis approaches(as discussed in Section 5), we implemented a modified version of the list scheduling algorithm proposed in [7], that performed speculation "on the fly", and compared it to our technique. Again, our algorithm provided consistently faster schedules with performance gains upto 57.58%. Detailed results are omitted due to space constraints.

7 Conclusions

The paper proposed a novel resource-aware algorithm for compiler-directed *ILP extraction* targeting *clustered* EPIC machines. In addition to extracting ILP from time-critical loops, our framework schedules and binds the resulting operations, generating actual VLIW code. Key contributions of our algorithm include: (1) an effective phase ordering for this complex optimization problem; (2) efficient load balancing heuristics to guide the optimization process; and (3) a technique that allows for maximum flexibility in speculating individual operations on a segment of predicated code.

References

[1] H. Corporaal. TTAs: Missing the ILP complexity wall. *Journal of Systems Architecture*, 1999.

[2] H. Corporaal and J. Hoogerbrugge. Code generation for Transport Triggered Architectures, 1995.

[3] K. Ebcioglu. A compilation technique for software pipelining of loops with conditional jumps. In *ISCA*, 1987.

[4] C. J. Tseng et. al. Bridge: A Versatile Behavioral Synthesis System. In *DAC*, 1988.

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

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

[7] Ganesh Lakshminarayana et. al. Incorporating speculative execution into scheduling of control-flow intensive behavioral descriptions. In *DAC*, 1998.

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

[9] M. Jacome et. al. Clustered VLIW architectures with predicated switching. In *DAC*, 2001.

[10] S. Gupta et. al. Speculation techniques for high level synthesis of control intensive designs. In *DAC*, 2001.

[11] S. Mahlke et. al. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *MICRO*, 1992.

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

[13] Sumit Gupta et. al. Conditional Speculation and its Effects on Performance and Area for High-Level Synthesis. In *ISSS*, 2001.

[14] T. Kim et. al. A Scheduling Algorithm for Conditional Resource Sharing. In *ICCAD*, 1991.

[15] V. Lapinskii et. al. High-quality operation binding for clustered VLIW datapaths. In *DAC*, 2001.

[16] J. Fritts. Architecture and Compiler Design Issues in Programmable Media Processors, Ph.D. Thesis, 2000.

[17] http://www.ti.com.

[18] Monica Lam. *A systolic array optimizing compiler*. PhD thesis, Carnegie Mellon University, 1987.

[19] Daniel M. Lavery and Wen mei W. Hwu. Modulo scheduling of loops in control-intensive non-numeric programs. In *ISCA*, 1996.

[20] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. In *Algorithmica*, 1991.

[21] N. Park and A. C. Parker. SEHWA: A Software Package for Synthesis of Pipelines for Synthesis of Pipelines from Behavioral Specifications. In *IEEE Trans. on CAD*, 1988.

[22] P. G. Paulin and J. P. Knight. Force-Directed Scheduling in Automatic Data Path Synthesis. In *DAC*, 1987.

[23] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *ISCA*, 1994.

[24] J. Sanchez and A. Gonzalez. Modulo scheduling for a fully-distributed clustered VLIW architecture. In *ISCA*, 2000.

[25] L. Dos Santos and J. Jess. A reordering technique for efficient code motion. In *DAC*, 1999.

[26] M. G. Stoodley and C. G. Lee. Software pipelining loops with conditional branches. In *ISCA*, 1996.

[27] K. Wakabayashi and H. Tanaka. Global Scheduling Independent of Control Dependencies Based on Condition Vectors. In *DAC*, 1992.

[28] P. F. Yeung and D. J. Rees. Resources Restricted Global Scheduling. In *VLSI 1991*, 1991.