The Dissertation Committee for Satish Pillai
certifies that this is the approved version of the following dissertation:

Compiler Directed Speculation for Embedded Clustered EPIC Machines

Committee:

Margarida F. Jacome, Supervisor

Gustavo de Veciana

Jacob Abraham

Calvin Lin

Ross Baldick

Compiler Directed Speculation for Embedded Clustered EPIC Machines

by

Satish Pillai, B.E., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2004

Acknowledgments

I wish to thank my advisor Dr. Margarida F. Jacome without whose guidance and support this thesis would not have been possible.

Compiler Directed Speculation for Embedded Clustered EPIC Machines

Publication No. _____

Satish Pillai, Ph.D.
The University of Texas at Austin, 2004

Supervisor: Margarida F. Jacome

Very Large Instruction Word (VLIW)/Explicitly Parallel Instruction Computing (EPIC) processors are a very attractive platform for many of today's multimedia and communications applications. In particular, clustered VLIW/EPIC machines can take aggressive advantage of the available instruction level parallelism (ILP), while maintaining high energy-delay efficiency. However, multicluster machines are more challenging to compile to than centralized machines. In this thesis, we propose a novel compiler-directed resource-aware ILP extraction technique, called predicated switching, that is targeted towards such multicluster VLIW/EPIC machines. The proposed technique integrates three powerful ILP extraction techniques – predication, speculation and software pipelining, in a combined framework. The three novel contributions in this dissertation are: (1) a compiler transformation, denoted Static Single Assignment - Predicated Switching (SSA-PS),

that leverages required data transfers between clusters for performance gains; (2) a static speculation algorithm to decide which specific kernel operations should actually be speculated in a region of code (hyperblock), possibly being simultaneously software pipelined, so as to maximize execution performance on the target processor; and (3) an ILP extraction flow incorporating several code generation phases critical to profitable ILP extraction by the compiler. Experimental results performed on a representative set of time critical kernels compiled for a number of target machines show that, when compared to two baseline "resource-unaware" speculation techniques (one that speculates aggressively and one that speculates conservatively), predicated switching improves performance with respect to at least one of the baselines in 65% of the cases by up to 50%. Moreover, we show that code size and register pressure are not adversely affected by our technique. Finally, we show that our ILP extraction framework combining speculation and software pipelining can effectively exploit the relative merits of both techniques.

Table of Contents

List of Tables

List of Figures

Chapter 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 or extract such ILP and to use processor datapaths with a large number of functional units, e.g., Very Large Instruction Word (VLIW)/Explicitly Parallel Instruction Computing (EPIC) processors.

A basic EPIC 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 or switching of data among FUs, while a typically slower interconnect provides access to or from the memory system. Unfortunately, this simple organization does not scale well with the number of FUs. When N FUs are connected to a register file, the area, delay and power dissipation of the register file can grow by up to N^3 [60]. 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 from

or write to a limited subset of registers [18, 60, 65]. In particular *clustered* EPIC processors can reap these benefits by organizing the datapath into multiple clusters of FUs connected to local storage. Figure 1.1 shows an example of such a datapath with two clusters each having two FUs. Note that for simplicity we do not show connections to or from the predicate register file. Many examples of clustered VLIW machines currently exist [4, 6, 14, 19, 22, 61].

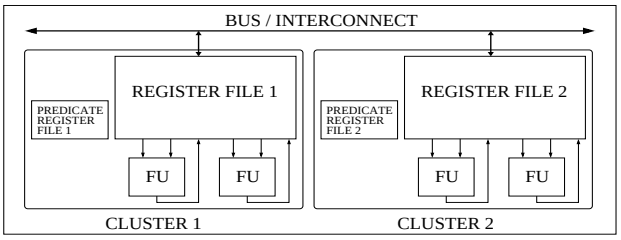


Figure 1.1: An example of a clustered datapath.

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 or copy data among datapath clusters (i.e., register files), 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 associated with switching costs on clustered machines may not adversely impact throughput, since higher clock rates may permit faster execution.

Many embedded applications 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. For example, a detailed analysis [21] of the Mediabench suite of programs [43] reveals that, on average, about 95% of the processing time is spent executing the 2 innermost loops. Yet another critical observation made [21] is that there exists considerably high control complexity within these loops. This strongly suggests that in order to be effective, ILP extraction techniques targeting such time critical inner loop bodies must handle control/branching constructs.

Predication combined with compiler-directed *speculation* has been shown to effectively increase ILP in the presence of conditionals [5]. 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. This is achieved by guarding operations with predicates and ensuring that an operation's predicate evaluates to true only if the control path where the operation resides is taken (see Chapter 2). However, on its own, this technique may not lead to significant performance gains [5]. Control *speculation* enables one to execute an operation on a conditional control path prior to knowing the value that the original operation's predicate would take [5]. Overall performance gains are more significant when these two techniques are combined [5]. Namely, by extracting extra ILP from a kernel (via control speculation), a more effective *utilization* of the target *machine resources* can be achieved leading to more compact and thus faster

code.

Unfortunately, as discussed in Chapter 5, aggressive compiler-directed control speculation may cause increased resource contention on critical code segments leading to performance degradation, an issue not previously addressed in the literature. It is therefore important to perform speculation selectively, using a resource aware cost function. A key hypothesis made in this dissertation is that, in order to be effective, compiler-directed control speculation techniques must be *resource-aware* and *highly selective*, i.e., individual operations should be speculated only when more compact code will indeed result. Accordingly, our proposed compiler algorithm, described in detail in Chapter 5, is not only resource aware, but can also leverage load distribution related data transfers across a machine's clusters to realize flow of control, being thus particularly effective for multicluster machines. These two characteristics make our approach unique in the literature (see Chapters 5 and 6).

Accordingly, in this dissertation, we propose a novel compiler-directed resource-aware ILP extraction technique, called predicated switching, targeted towards multicluster EPIC machines. The proposed technique integrates three powerful ILP extraction techniques – predication, speculation and software pipelining, in a combined framework to achieve significant performance gains. The effectiveness of predicated switching relies on several novel contributions as described below.

The first contribution is a novel compiler transformation, denoted Static

Single Assignment - Predicated Switching (SSA-PS), that has the potential to efficiently exploit the hierarchical storage/switching resources in a clustered datapath [35]. In particular, it exposes additional performance enhancement opportunities by leveraging the penalties associated with transferring data across clusters to realize the code's flow of control. As will be seen, SSA-PS is analogous to standard predication in that it converts a program's control flow into data flow. However, in contrast to standard predication, in code generated via SSA-PS, part of the flow of control may be realized through predicated data transfer operations, whereby *predicated moves* select which among a set of results or values from alternative execution paths to place at a prespecified destination, correctly modifying the processor state. As will be seen in Chapter 5, the SSA-PS transformation enables a form of targeted ILP extraction, by enabling *select* kernel operations to be aggressively speculated by the compiler in the final code. As alluded to above, such an ability is critical to achieving actual performance gains on specific target machines.

A second key contribution of this thesis is an algorithm to decide which operations to speculate on any given kernel, so as to maximize execution performance. Note that failure to speculate critical path operations to early points on the schedule, where resource contention may be less stringent, may lead to sub-optimal performance. On the other hand, excessive speculation may actually hurt performance instead of improving it, since it may lead to latency penalties due to operations on the critical path being delayed (see Chapter 5). Thus, the ability to consider the limited set of resources available

on the target processor, when performing speculation, is central to maximizing code performance. Our proposed resource-aware static speculation algorithm addresses this complex problem in the context of clustered EPIC machines. To the best of our knowledge, our proposed *resource-constrained* speculation technique is unique in the literature on static speculation for clustered EPIC machines with support for predication. Experimental results show that the speculation technique implemented in predicated switching can deliver performance improvements of up to 40.5% by an average of 7.2% on clustered machines, when compared to resource-unaware speculation techniques. Moreover, we show that code size and register pressure are not adversely affected by our technique.

A third contribution of this dissertation is an iterative ILP extraction flow incorporating several code generation phases critical to a successful (profitable) ILP extraction by the compiler, including if-conversion, binding, control speculation, scheduling and software pipelining (see Chapter 5). We empirically show that by interleaving speculation and software pipelining, we can trade off the relative merits of both techniques and thus achieve high quality code. Our experimental results show that the proposed combined resource aware ILP extraction flow can deliver performance increases of up to 50% by an average of 6.6%, over resource-unaware baseline techniques.

The rest of the thesis is organized as follows. Chapter 2 provides some background on EPIC machines and introduces relevant features supported by such architectures, namely predication and speculation. Chapter 3 discusses

previous relevant work. Chapter 4 presents the proposed SSA-PS transformation. Chapter 5 presents our resource-aware ILP extraction algorithm and details the overall optimization flow framework along with experimental results. Chapter 6 shows how software pipelining can be seamlessly included in our optimization framework. Chapter 7 outlines conclusions and future work.

Chapter 2

Background

EPIC architectures enable the compiler to expose the ILP present in the application directly to the micro-architecture. Moreover, they provide a set of novel features that enable the compiler to effectively extract an application's ILP, so as to generate code that more efficiently utilizes the large number of issue slots typically available on such machines. This chapter introduces some of the new features available on EPIC machines that contribute towards achieving high performance. Specifically, we will discuss predication and speculation, two powerful ILP extraction techniques supported on EPIC machines.

2.1 Predication

Branches present a major barrier to ILP. Namely, in the presence of a branch, the pipeline of a machine with no branch prediction needs to stall to wait for the target address to be computed. On machines with branch predictors, the penalties due to branch mispredictions can still be quite significant [27].

Predicated execution is an architectural model [30, 37, 59] in which each

operation is guarded by a predicate whose value determines if the operation is to be committed or nullified. Required extensions to the instruction set architecture include an additional boolean operand (called predicate) guarding all (or some) instructions, and a set of compare instructions used to define predicates.

Predication provides a mechanism to eliminate branches through a process called *if-conversion* [3]. 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 (on predicate values), generating what is called a *hyperblock* [49].

Similar to previous static (or compiler-directed) ILP extraction techniques, if-conversion enlarges scheduling scope across various basic blocks, but does so without causing expensive increases in code size [3]. Since if-conversion allows the compiler to overlap the execution of independent control paths, it can be used to eliminate unpredictable (unbiased) branches and associated performance losses due to stalls and/or branch misprediction recovery time [48, 63].

We illustrate if-conversion using the sample code segment (with a conditional statement) shown in Figure 2.1(a). The basic blocks associated with

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

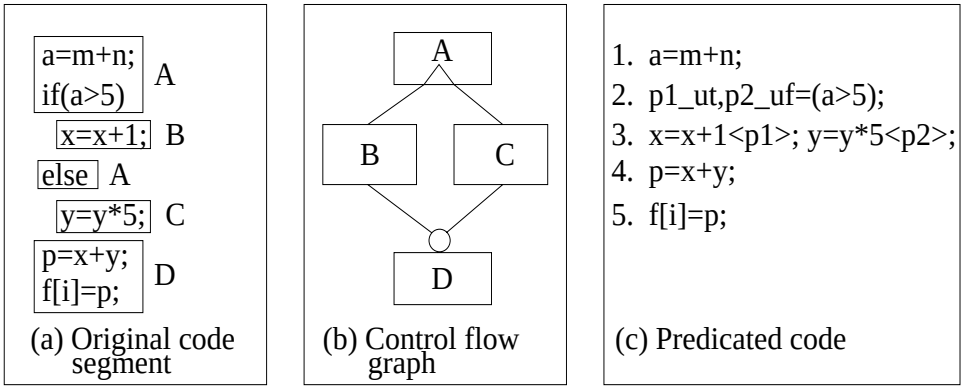


Figure 2.1: Sample code segment (a), its control flow graph (b) and its predicated version (c).

this code segment have been labeled A-D and the associated control flow graph is shown Figure 2.1(b). Throughout this discussion, we shall assume that all operations take one cycle, except for memory accesses which take two cycles. Predicate values can be used *one cycle* after their definitions [49]. Note that in order to support the use of predicate values one cycle after their definitions, current machines nullify operations on the write-back/commit phase (as opposed to the fetch or execute phase). Thus, operations guarded by FALSE predicates are actually *executed* but their results are not allowed to modify the processor state.

Figure 2.1(c) shows the resulting predicated code. Each EPIC instruction in the code comprises a set of micro-instructions (which we denote operations) to be started simultaneously – on the left of Figure 2.1(c), we assign a scheduling step to each such EPIC instruction. For example, the EPIC instruction at step 3 of the schedule in Figure 2.1(c) has one addition and one

multiplication operation. Predicated operations are indicated by appending a predicate operand $< p >$, holding the associated guard. For conciseness, the $< TRUE >$ predicate on unconditionally guarded operations is omitted. Flexible predicate assignment types are used in this code, including the ut and uf, unconditional true and false types, e.g., in our example $p1\_ut = (a > 5)$ sets $p1$ to true if $(a > 5)$ and false otherwise, while $p2\_uf$ does the opposite. (See [5] for details on more types of predicate define operations.) Note that once 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 represented by predicated code.

Using the if-conversion transformation illustrated above, an entire acyclic control flow region can be converted to a single branch-free block of predicated code. Naturally, the extent to which actual performance gains can be achieved through predication strongly depends on a proper selection of the code regions to be if-converted, i.e. hyperblocks to be formed. 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 thesis – a good set of criteria can be found in the work by Mahlke [49].

2.2 Speculation

There are two fundamental types of speculation: control and data speculation. *Control* speculation breaks the *control dependence* between an operation and the conditional statement it is dependent on. By eliminating this de-

pendence, the operation can be moved out of the conditional branch it resides in, and be scheduled for execution before its related conditional is actually evaluated. By exploiting control speculation, the compiler can thus reduce control dependence height, which enables the generation of more compact, higher ILP code. Similarly, *data* speculation enables breaking of *data flow dependences* between operations, e.g., moving a memory read operation ahead of a memory write operation, when the independence between these operations cannot be proven [5]. In this thesis, we focus strictly on compiler-directed *control* speculation.

Control speculation in the context of a hyperblock is realized via predicate promotion, whereby an operation's (micro-instruction's) predicate is changed to a predicate whose expression subsumes that of the original predicate [5]. Consider, for example, the simple case of a predicated operation that does not modify the processor state (i.e., does not change the value of a program variable). Then, the compiler can move it up to the point where its operand(s) are *uniquely* defined [5, 47]. 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 will be speculatively executed at run time.

When the operation to be predicate promoted does not modify the state of the processor, we say that it can be *directly* promoted. Note that for the example in Figure 2.1 no such direct predicate promotion could be performed on the operations in Blocks B and C (see Figure 2.1(a)). Indeed,

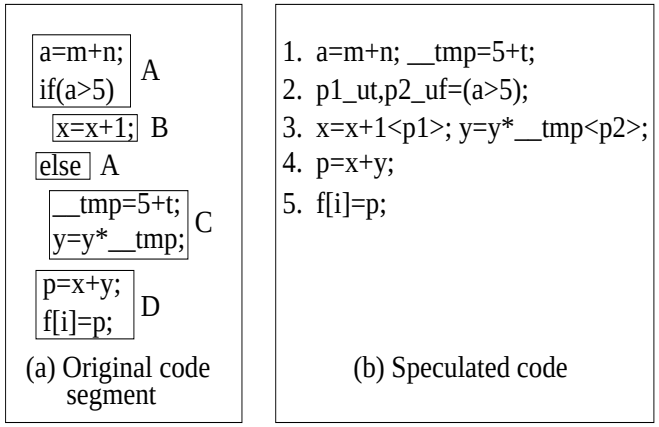


Figure 2.2: Sample code segment (a) and its speculative version (b).

the operations in each block directly modify program variables x and y. They must thus be executed after the definition of their corresponding predicates, or the processor state (i.e. the values of variables x and/or y) may be incorrectly modified. However, consider the example code segment shown in Figure 2.2(a). Here, *__tmp* represents a compiler generated temporary value and thus its corresponding definition operation (originally in basic block C) can be scheduled for speculative execution using direct predicate promotion, as shown in Figure 2.2(b). The operation, in this case, is promoted to basic block A which is unconditionally executed and for simplicity its $< TRUE >$ predicate is omitted.

Throughout this thesis, we refer to this form of speculation (realized by direct predicate promotion) as *conservative speculation*. Note that, in contrast to more *aggressive speculation* techniques, which require adding extra operations to the code, or extensive power hungry hardware support, conservative

speculation has no inherent cost and thus will never harm performance. We found that in practice opportunities for speculation via direct predicate promotion occur quite frequently (e.g., in the Mediabench benchmark suite), since the compiler typically generates a significant amount of temporary variables.

Both predication and speculation will be used in our optimization framework, described in detail in Chapter 5. In the next chapter, we discuss previous related work concerning the use of these techniques to optimize code performance.

Chapter 3

Previous Work

Compiler related research as well as high level synthesis have extensively addressed the concept of speculation for ILP extraction. In this chapter, we briefly review work from both these areas.

3.1 Compilers: Region Based Scheduling and Control Speculation.

We start by reviewing research in the area of compiler-directed region based scheduling. The key idea underlying region based scheduling is to provide the compiler with an optimization scope that is sufficiently large so as to create increased opportunities for optimization. These approaches thus rely on an appropriate selection of large contiguous *regions* of code and generally incur code explosion due to region expanding transformations like loop unrolling. A region may encompass multiple basic blocks with control flow handled as discussed in more detail below. In fact, one of the key advantages of predication (also a region-based technique) was to overcome the code explosion problem incurred by most previous region-based techniques.

Trace scheduling [20] is a region based technique that uses profile in-

formation to divide the code into a set of frequently executed paths, called traces, and schedules those paths by ignoring entries to or exits from them. Later, compensation code is added to each entry and/or exit from a trace to maintain program correctness. However, the addition of such compensation code can result in large increases in code size.

Superblock scheduling [34], an evolution of trace scheduling, forms large regions of code, called superblocks, in which entrances to the regions are allowed only from the top. All side entrances are eliminated by a process called tail duplication [34], which unfortunately also increases code size. Note that the elimination of side entrances obviates the need for complex compensation code that would need to be added when instructions are moved either above or below such side entrances. In addition, applying optimizations to the selected traces (e.g. constant propagation) becomes simpler when side entrances do not exist, since compensation code induced by those optimizations to ensure program correctness due to side entrances is now not necessary. Finally, the formation of superblocks typically involves a series of code transformations, e.g., branch target expansion, loop peeling, loop unrolling etc., which cause further code expansion. Such increases in code size are unacceptable in the context of memory limited embedded systems and may also significantly affect performance in general purpose processors, thus impacting the figure of merit that one is trying to optimize in the first place.

Note that the success of the two region based techniques discussed above obviously depends on how frequently the formed traces or superblocks

actually execute. If the program's behavior varies significantly with its input data set, it may be difficult to generate the large optimized traces needed to truly enhance performance. Thus, performance can only be reliably improved when there is a significant *bias on execution paths*. Other compiler-directed techniques based on similar principles [11, 12] suffer from the same problems.

A state of the art static compiler-directed ILP extraction technique that is particularly relevant to this dissertation is if-conversion [3] based predication enhanced with control/data speculation [5]. It is empirically shown that predication combined with control and data speculation achieves much better performance than each individual technique when used alone [5]. The proposed techniques rely on the IMPACT EPIC execution model [5], which has advanced architectural features for supporting predication and speculation. Specifically, hardware mechanisms exist to detect potential exceptions on control-speculative operations and initiate repair. Deferred exception handling is used, whereby an exception is serviced only if it would have occurred in the original code. This is accomplished by additions to the operation encoding as well as to the register file. Namely, a single bit, called S-bit, is added to each operation that can be speculated. This bit is set on operations that are either control-speculated or are data dependent on a data speculated load [5]. Additionally, an extra bit called the DS-bit is added to each load operation and is set when the load is data speculated. These operation encodings differentiate speculative operations from non-speculative ones. This differentiation is important because only speculative operations require delayed exception

handling. Moreover, two extra bits called the E-tag and the R-tag are added to each register. The E-tag of a register is set to indicate that an exception occured in the creation of the value in that register. The R-tag is used for executing only dataflow successors of excepting speculative operations, during recovery from exceptions [5]. Also, the compiler facilitates delayed exception handling by inserting an explicit check operation in the homeblock of every operation that has been speculated.[1] Only when the check operation in the original homeblock of the excepting speculated instruction is executed, will the exception be serviced. This scheme ensures that unnecessary exceptions are not serviced. Additionally, data speculation is enabled by the Memory Conflict Buffer, a device that checks for conflicts between speculatively executed loads and subsequent stores [23]. The work of August [5] also assumes hardware support to buffer speculative state. In contrast, our approach to control speculation (see Chapter 4) relies on compiler-driven variable renaming, thus leveraging the existing architectural registers for the purpose. This "all software" strategy reduces the need for certain power hungry hardware assists by transferring to the compiler the task of buffering speculative state and deciding whether speculative results should be either committed or discarded.

A predicated Static Single Assignment (SSA)[2] conversion has been proposed [10], aimed at enabling aggressive speculation on VLIW machines. Unfortunately, the proposed transformation leads to code explosion – a problem

[1] When possible, an existing operation in the homeblock may be used as an implicit check operation.

[2] The SSA transformation is discussed in detail in Section 4.1.

that predication aimed to circumvent in the first place. In simple terms, the SSA representation is used 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. Thus, 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. Other SSA based optimizations aimed at increasing memory parallelism and decreasing redundant memory references can be found in the work by Budiu [8].

Conditional move operations, i.e. move operations that are predicated, have been previously used to achieve performance enhancements [39, 45, 46]. For example, support for predication on machines that only support *conditional move* operations (as opposed to machines that support predication on all instructions in the ISA) has been proposed [46]. 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 operations and related data dependences. Accordingly, significant performance degradation is reported with respect to standard predication [46].

There is also a large body of previous work concerning processor support for dynamic control/data speculation, most of which requires extensive, power hungry hardware assists to dynamically predict when to speculate operations and restore processor state due to possible incorrect speculation. In contrast, this thesis addresses compiler-directed, i.e. *static* speculation techniques.

As will be seen in Chapters 5 and 6, we have developed a pure software approach for hyperblocks that incurs no code explosion. Moreover, we show that for pure software approaches it is important to perform speculation selectively, using a resource aware cost function, since excessive speculation may actually hurt performance, an issue not previously addressed in the literature. Accordingly, our proposed compiler algorithm is not only resource aware, but can also leverage load distribution related data transfers across a machine's clusters to realize flow of control, being thus particularly effective for multicluster machines. These two characteristics make our approach unique in the literature (see Chapters 5 and 6).

3.2 High Level Synthesis – Programmable Hardware Accelerators.

Some of the fundamental assumptions underlying speculation techniques proposed for high level synthesis do not apply to the code generation problem addressed in this thesis, for the following reasons. First, the cost functions used for hardware synthesis, aiming at minimizing functional unit,

control, multiplexing and interconnect costs, are significantly different from those used by a software compiler, since the target datapath assumed by the compiler is "fixed" and thus schedule length is the key cost function to optimize. Second, most research in the high level synthesis area exploit conditional resource sharing. Unfortunately, this cannot be exploited in EPIC code generation, because predicate values are unknown at the time of instruction issue. In other words, two operations (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. Thus, the basic optimization problem addressed in this thesis is fundamentally different from that solved by high level synthesis approaches.

Still, for completeness, it is worth mentioning that the previously alluded to conditional resource sharing techniques have been exploited [38, 54, 55, 62, 64, 66]. Node dividing and conditional tree duplication techniques using conditional vectors have been proposed [64], in an attempt to optimize all possible paths in the schedule. Aggressive code motions (speculation) have been proposed [25, 26] for the high level synthesis scheduling problem; these techniques are geared towards minimizing hardware costs, and are thus not applicable in the code generation context, where the target machine has a "fixed" micro-architecture.

Speculative execution has been incorporated in the framework of a list scheduler [40]. This technique makes decisions on speculation by considering branch probabilities and resource constraints. Although this method may appear to be similar to what is proposed in this thesis, it is in fact vastly different, for the following reasons. Speculation, as performed by our proposed SSA-PS technique, is unique in the sense that copies of speculated operations are made, and the resulting values are reconciled by predicated move operations - this critical aspect of SSA-PS is extensively discussed in Chapter 4. In contrast, there is no concept of predication and, hence, no merging of control paths to create a single block of straight line code. Moreover, being a high level synthesis paper, there is no notion of a "fixed" clustered datapath to compile to, i.e., optimize for. Also, the heuristic used speculates operations on the longest path (criticality). Although this strategy is effective in some circumstances, it does not consider load balancing, which is critical for clustered architectures.

Path based scheduling [9] finds all possible execution paths and schedules each one of them optimally. It then overlaps these individual schedules using a heuristic ordering. This technique, apart from being computationally intensive, may also result in severe code duplication.

Percolation scheduling [51] starts with an optimal resource unconstrained schedule and applies semantics-preserving transformations that convert an original program graph to a higher ILP form. Specifically, by repeatedly applying a set of atomic/basic transformations, several operations are allowed to percolate to the top of the program graph. However, as with the previous

case, this technique may lead to code explosion. Also, the incremental nature of these transformations reduces the efficiency of the technique. This was curbed to some extent by Trailblazing [52], an efficient hierarchical approach to percolation scheduling, but the fundamental problem of code explosion remained.

Profiling information has been used [28, 29] to predict the path through the loop kernel with the highest execution probability. This path is scheduled using software pipelining [24]. If the prediction is incorrect, a restore phase is executed. This approach clearly relies on the ability to accurately predict branches within a loop, and is thus not particularly effective in the presence of unbiased branches.

Control flow transformations aimed at effective design space exploration have been described in the literature [53]. These high-level condition expression transformations target hardwired architectures, and are thus not relevant to our work.

An exact solution to the problem of scheduling and binding for clustered VLIW application specific processors (ASIPs), assuming an unlimited register capacity and no support for predication or speculation, has been proposed by us [56]. Our approach uses Binary Decision Diagrams (BDDs) to capture the entire solution space in a compact data structure. Efficient algorithms to search the solution space for optimal solutions have also been developed [56]. This technique, however, like all other exact techniques, is computationally intensive and may be unfeasible for large sized problems.

Certain special purpose architectures, like transport triggered architectures [15], do not fit in either of the two research categories discussed in this thesis, as these processors 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 EPIC processors assumed in this thesis [16].

Chapter 4

SSA-PS Transformation

This section discusses in detail the proposed Static Single Assignment - Predicated Switching (SSA-PS) compiler transformation [35]. Since the SSA-PS transformation relies on the well known Static Single Assignment (SSA) representation [17],[7], we will first briefly review it.

4.1 SSA representation

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. variable renaming, to ensure that the single assignment property is satisfied and;

2. ϕ functions, to reconcile multiple assignments that reach a join point in the control flow.

Thus, the transformation to represent a code segment in SSA form involves two steps: identifying the placement of the ϕ functions in the code and renaming variables appropriately. This is a relatively straightforward process,

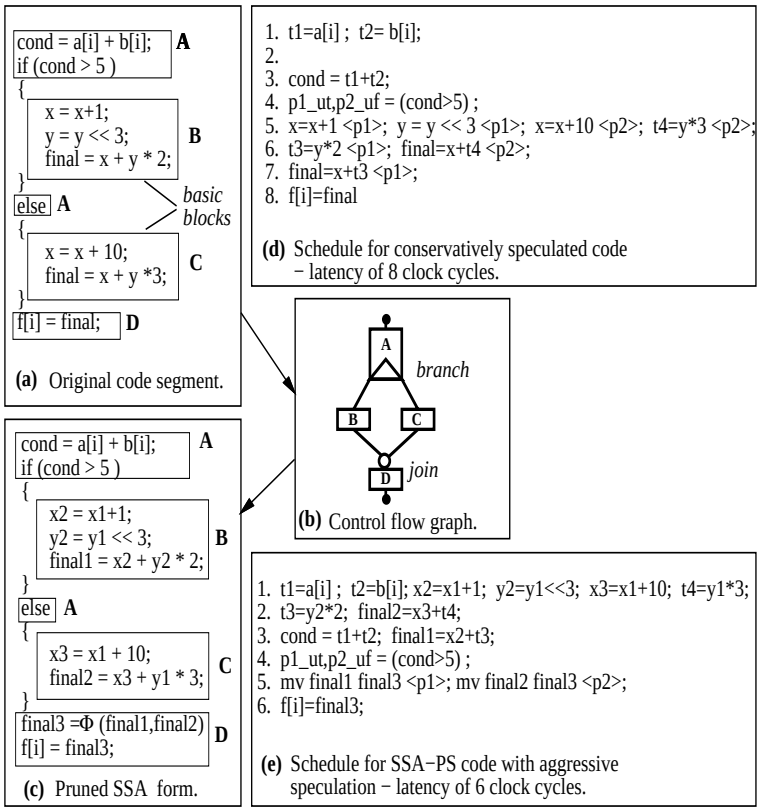


Figure 4.1: Sample code segment (a), its control flow graph (b), its pruned SSA form (c), conservatively speculated code schedule (d) and SSA-PS code schedule (e).

and we will use the code segment (with a conditional statement) shown in Figure 4.1(a) to illustrate the transformation. The basic blocks associated with this code segment have been labeled A-D and the associated control flow graph is shown in Figure 4.1(b). Consider the variable $final$ in the sample code. Assignments to $final$ are made in Blocks B and C; thus a ϕ function is inserted at the join point at the entry to Block D, as shown in

Figure 4.1(c).[1] 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 value resolved by the ϕ function at the join point. Note that, if the variable $final$ were not subsequently used (in the example, written to memory), then ensuring a unique assignment from the point of view of subsequent basic blocks would be unnecessary. Thus, as shown in Figure 4.1(c), the ϕ functions associated with the variables x and y can be eliminated – such code is said to be in the *pruned* SSA form [13].

The role of the ϕ functions is to realize conditional assignments to the renamed variable, depending on which control path is followed. 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 $final3$.

4.2 SSA-PS Fundamentals – Flexibility in Speculation

The idea underlying the SSA-PS conversion is to realize the conditional assignments corresponding to ϕ functions via *predicated switching* op-

[1] Efficient algorithms for determining a minimal number of locations for ϕ functions are discussed in [17].

erations, in particular, predicated *move* operations. Figure 4.1(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 $mv\ final2\ final1 < p2 >$. A simple post-processing step can be carried out to perform this optimization, so as to reduce the number of move operations in the code.

Consider now that we if-convert the same code segment and then conservatively speculate it. For our small illustrative kernel, the schedule of the conservatively speculated code would take 8 steps (Figure 4.1(d)) while SSA-PS code would take only 6 steps (Figure 4.1(e)), i.e., give a 25% improvement in performance. This is so because SSA-PS not only transforms control flow into data flow, but, through SSA's variable renaming, enhances the ability of the compiler to speculate operations. In Chapter 5, we further elaborate on the critical importance of providing the compiler with such extended flexibility to speculate operations.

In the illustrative example discussed so far, we have assumed a target *centralized* machine, and illustrated the improved ability to speculate opera-

tions provided by SSA-PS. There is however a key additional benefit to using SSA-PS, in the context of clustered machines. The next section discusses such benefits.

4.3 SSA-PS Transformation

In this section, we first discuss the instruction set architecture (ISA) extensions that will be required to support the SSA-PS transformation on a clustered machine. Next, we discuss the potential of the SSA-PS transformation to improve performance on such clustered machines.

4.3.1 ISA Extensions for Clustered EPIC Machines

Clustered machines require an instruction set that supports data transfers between the clusters in the datapath. Recall that, for such machines, clusters have their own (*local*) general-purpose and predicate register files. Thus, two possible types of *predicated move operations* can be defined: *internal* and *external* 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, where both register locations belong to the same local/cluster register file.

Predicated external moves can be further classified into two types: *basic* external moves and *predicate transfer* external moves. Basic external moves, denoted $mvE\ src\ dst\ < p >$, have the same functionality of internal moves,

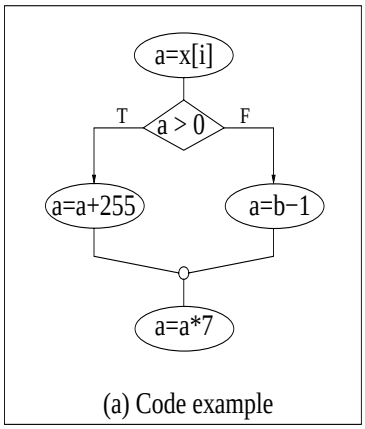
except that the registers *src* and *dst* belong to register files in different clusters. Predicate transfer external moves, denoted $mvET\ src\ dst\ < p >$, in addition to the functionality of the basic external move instruction, also transfer the predicate value from the source to the destination cluster.[2] This type of instruction is required in cases where the predicate needs to be made available to the destination cluster, so that squashing of the operation (if needed) can be performed in the write back stage, with no delay penalties [49]. It should be clear that the ISA extensions defined above must necessarily be supported when *clustered* machines with support for predication are considered.

4.3.2 Performance on Clustered Machines

We start by illustrating the ideas using the small code segment shown in Figure 4.2(a). When scheduled on a centralized machine (see Figure 4.2(b)), both conservatively speculated code and code generated by SSA-PS give the same latency (5 steps). Thus, the extended ability of SSA-PS to speculate did not lead to any performance improvement in this case.

Assume now that the target machine is clustered (with 2 clusters) and that, in order to take advantage of the available ILP, the operations on the two branches are bound to different clusters, as indicated in Figure 4.3(a). The conservatively speculated code now takes 6 steps to complete while the code generated by SSA-PS still takes only 5 steps (see Figure 4.3(b)). In particular, note that, for the conservatively speculated code schedule, the variable

[2]Note that, for conciseness, the predicate transfer destination is not shown explicitly.



(a) Code example

	CONSERVATIVE SPECULATION	SSA−PS
1	a=x[i];	a=x[i]; a2=b−1;
2		
3	p1_ut,p2_uf=(a>0);	p1_ut,p2_uf=(a>0); a1=a+255;
4	a=a+255<p1>; a=b−1<p2>;	mvl a1 a<p1>; mvl a2 a<p2>;
5	a=a*7;	a=a*7;

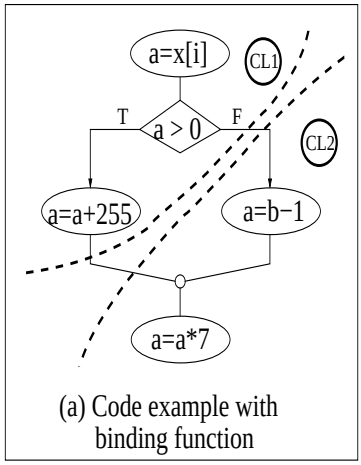
(b) Conservative speculation and SSA−PS schedules

Figure 4.2: Code example (a) with schedules (b) for a centralized machine.

a computed on the TRUE branch of the conditional (at step 4) needs to be transferred to cluster 2 *on the subsequent time step*. SSA-PS, on the other hand, *collapses* the resolution of variable a (ϕ function implementation) with the required inter cluster data transfer operation (see step 4 of the schedule), and thus no additional time step was required for execution on the clustered machine, resulting in more compact/faster code.

Consider now the same code segment with a second binding function, shown in Figure 4.4(a). Similar to the previous case, on step 5 of the SSA-PS schedule (see Figure 4.4(b)), the predicated move operation performs the dual function of variable resolution and data transfer between clusters. Due to such collapsing of functions, SSA-PS code requires only 6 steps while conservatively speculated code takes 7 steps.

Consider now a more realistic example, namely, the loop body shown in



(a) Code example with binding function

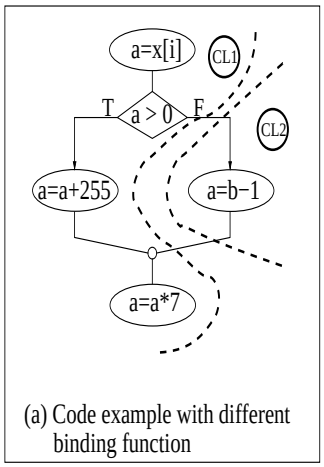
	CONSERVATIVE SPECULATION		SSA−PS	
	CLUSTER 1	CLUSTER 2	CLUSTER 1	CLUSTER 2
1	a=x[i];		a=x[i];	a=b−1;
2				
3	p1_ut,p2_uf=(a>0);		a1=a+255; p1_ut,p2_uf=(a>0);	
4	a=a+255<p1>; mvE p2 p2_CL2;		mvET a1 a_CL2<p1>;	
5	mvET a a_CL2<p1>;	a=b−1<p2>;		a=a*7;
6		a=a*7;		

(b) Conservative speculation and SSA−PS Schedules

Figure 4.3: Code example with binding function (a) and schedules (b) for a 2 cluster machine.

Figure 4.5(a). The ILP of the loop body is increased through software pipelining [44] [41], a technique that allows overlapping multiple loop iterations in the same execution cycle (see Chapter 6). Operations belonging to a single iteration correspond to a pipe stage of the software pipelined loop (see Section 6.1 for more details on software pipelining). The corresponding software pipelined (retimed) version of the loop is shown in Figure 4.5(b). Note that the retiming function (see Chapter 6) was selected to cut the loop initiation interval[3] by about a half. Assuming that each pipe stage is assigned to a different cluster of the machine (as indicated in Figure 4.5(b)), Figures 4.5(c) and 4.5(d) show the resulting optimal schedules (and corresponding loop initiation intervals) achieved by SSA-PS and conservatively speculated code. The conservatively

[3]The initiation interval of a loop is the average rate at which a new loop iteration can be started (see Chapter 6).



(a) Code example with different binding function

	CONSERVATIVE SPECULATION		SSA−PS	
	CLUSTER 1	CLUSTER 2	CLUSTER 1	CLUSTER 2
1	a=x[i];		a=x[i];	a2=b−1;
2				
3	p1_ut,p2_uf=(a>0);		a1=a+255; p1_ut,p2_uf=(a>0);	
4	a=a+255<p1>; mvE p2 p2_CL2;		mvE p2 p2_CL2; mvI a1 a<p1>;	
5		a=b−1<p2>;		mvET a2 a_CL1<p2>;
6		mvET a a_CL1<p2>;	a=a*7;	
7	a=a*7;			

(b) Conservative speculation and SSA−PS Schedules

Figure 4.4: Code example with different binding function (a) and schedules (b) for a 2 cluster machine.

speculated code incurs a 20% performance penalty when compared to the code generated using SSA-PS. Note that for this example, code generated via SSA-PS speculates the two conditional updates of the renamed variable a to Step 4, and then moves the correct value to Cluster 2 on Step 5 (see Figure 4.5(c)). The predicate transfer external move operations in step 5 thus realize both the ϕ function and the required data transfer from Cluster 1 to 2. In contrast, for the conservatively speculated code shown in Figure 4.5(d), the external move of variable a from Cluster 1 to 2 must be *appended* to the code, which delays the schedule.

In summary, the proposed SSA-PS transformation exposes new opportunities for performance enhancement, through the *merging/collapsing* of two distinct types of operations: (1) predicated move operations (ϕ functions) re-

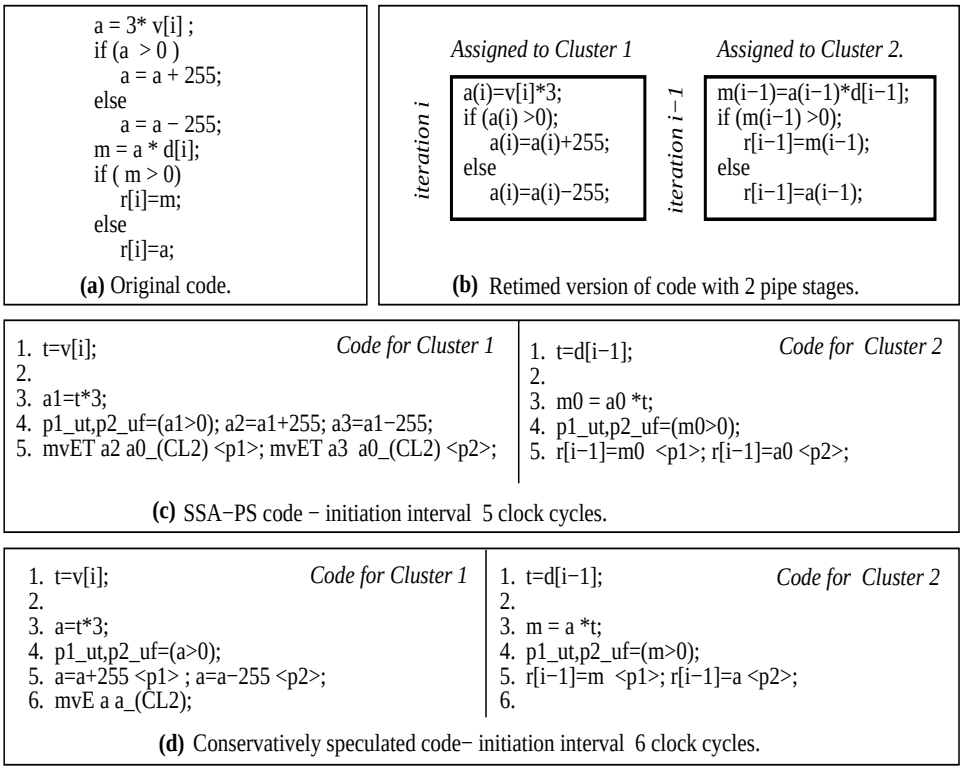


Figure 4.5: Example SSA-PS and conservatively speculated code on a clustered datapath.

alizing the *code's control flow*, i.e., needed to ensure correct *code behavior*; with (2) *load distribution operations*, needed to ensure an effective *machine utilization*. We thus argue that SSA-PS is particularly effective in the context of *clustered* machines, since it performs code transformations specifically directed towards reducing potential latency penalties incurred by such machines. The experimental results presented in Chapters 5 and 6 demonstrate that such opportunities occur frequently in real application's code, enabling code generation via application of SSA-PS to select operations to achieve (usually quite

significant) performance gains on clustered machines.

Chapter 5

Resource Aware Speculation – Predicated Switching

So far, we have implicitly assumed that the target machine has sufficient resources to execute all operations, including the speculated ones, at the earliest possible scheduling steps i.e., the As Soon As Possible (ASAP) schedule (defined by data dependences only). However, as suggested in Chapter 1, the objective of compiler-driven speculation should be to improve performance by enabling a more effective utilization of the target machine's datapath resources. Intuitively, in order to improve performance, the compiler should try to speculate/displace operations located at oversubscribed steps in the ASAP schedule, to earlier less-congested points, thus realizing a more flattened static load distribution, which in turn enables the generation of more compact/higher ILP code. Accordingly, in this chapter, we discuss the problem of deciding which operations to speculate so as to achieve a more effective utilization of datapath resources and thus improve performance.

Typical load distribution profiles of real kernels (from multimedia and communications applications) can be quite complex, i.e., may contain several arbitrary size regions of resource overloading interleaved with regions of

resource under utilization. To compound the problem, the mobility of operations within such regions is typically constrained by a complex web of true data dependences.

Accordingly, in order to generate code that maximizes performance, it is important to provide the compiler with maximum *flexibility* to selectively speculate individual kernel operations, i.e., to have the ability to rearrange operation's scheduling ranges as aggressively as possible, during the optimization process. Note also that *clustered* EPIC machines significantly increase problem complexity. Indeed, extracting extra ILP (via operation speculation) to only have the associated performance gain be overshadowed by required data transfers across clusters is obviously undesirable. Our SSA-PS transformation (described in Chapter 4) provides the compiler with the required flexibility to perform such selective speculation, as well as reduce the cost of required data transfers across clusters. The critical importance of these two points will become clear during the discussion of our proposed resource aware speculation algorithm (see Section 5.3).

The proposed algorithm for optimizing speculation 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 EPIC datapath. In the next section, we explain how these load profiles are calculated and how the process of speculation alters such profiles.

5.1 Load Profile Calculation

The load profile [42] is a measure of the resources required to execute a Control Data Flow Graph (CDFG) within a desirable schedule latency. In this section, we discuss how the load profile for a given CDFG is determined.

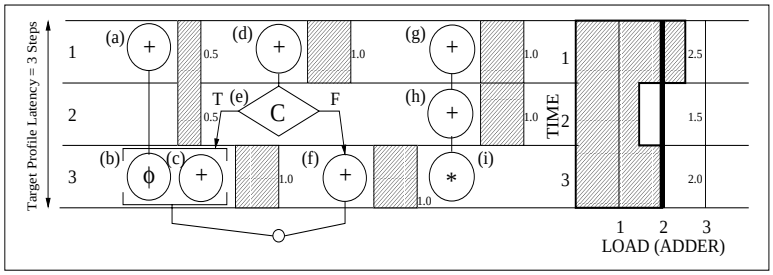


Figure 5.1: Adder Load Profile Calculation.

Consider the CDFG in Figure 5.1. The load profile is calculated for a given target profile latency (greater than or equal to the critical path of the CDFG). First, As Soon As Possible (ASAP) and As Late As Possible (ALAP) schedulings are performed, to determine the scheduling range of each operation. For the example in Figure 5.1, operation a has a scheduling range of 2 steps (i.e. step 1 and step 2), while all other operations have a scheduling 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 [55], 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 5.1, the contributions to the load of all the addition operations (labeled 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 5.1, 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 EPIC code/schedule, one of the addition operations (i.e. either operation a, d or g) needs to be delayed to a later time step (see e.g. schedule in Figure 5.3(b), with operation a delayed to time step 2).

The next section shows how the generated load profiles can be used during compiler-directed speculation, to perform load balancing.

5.2 Load Balancing through Speculation

We start by arguing that speculating operations without considering resource constraints in the datapath may lead to performance losses. Consider again the small kernel given in Figure 5.2(a). Note that, when predicated code is considered, the branching constructs shown in Figure 5.2(a) actually correspond to the definition of a predicate, and the associated control dependence corresponds to a data dependence on the predicate values. This is shown in

Figure 5.2(b), where PD denotes the predicate define operation which defines the mutually exclusive predicates p and $p!$.

The schedules resulting from speculating zero or more operations are shown in Figures 5.3(a), 5.3(b) and 5.3(c). Recall that the ϕ operations in Figure 5.3 reconcile speculated definitions of variables to their correct values (see Chapter 4).

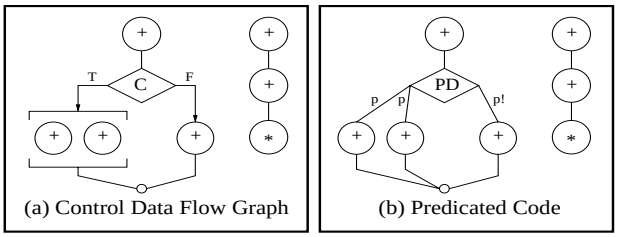


Figure 5.2: Example Kernel.

Assume that a centralized EPIC datapath with 2 adders, 2 multipliers and a single comparator, each of which taking a single-cycle to execute, is being targeted. Assume also that predicated move operations can be executed by any of the FUs. Speculating a single operation in the original Control Data Flow Graph (CDFG), shown in Figure 5.2(a), improves the original loop initiation interval by 25% (see Figure 5.3(b) with operation a speculated) over the schedule with no speculation (see Figure 5.3(a)). However, speculating three operations would provide the same initiation interval as the schedule with no speculation, i.e. provide no performance improvement (see Figure 5.3(c)).

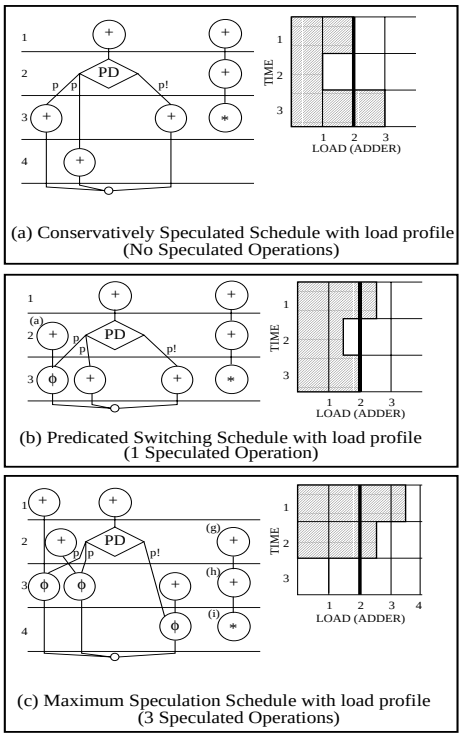


Figure 5.3: Performance vs Speculation.

As illustrated in Figure 5.3, we see that the process of speculation increases the possible scheduling ranges of the operations being speculated. This, in turn, results in increased contention for resources at earlier time steps, as reflected by the altered load profiles. When an operation can be speculated by direct predicate promotion (i.e., when it defines a temporary variable), the original set of scheduling solutions is included in the set of new scheduling solutions now made possible by speculation. Thus, as indicated in Chapter 2, such conservative speculation will never harm performance. However, when operation speculation is performed by the SSA-PS transformation, a new rec-

onciliation ϕ function (operation) is generated, which will result in the new set of possible scheduling solutions not encompassing the original set. Moreover, the newly generated reconciliation operation also adds to the contention for resources. This may result in critical path operations being delayed and/or the critical path itself being modified, and thus may actually lead to an increase in overall latency. Our experimental results section (Section 5.5) shows several such examples for real benchmarks executing on a variety of target machines.

In light of the above, we see that the process of speculating operations requires a careful consideration of resource constraints in order to provide significant and consistent performance gains. Accordingly, in this section we argue that in the context of EPIC machines, the goal of compiler transformations aimed at speculating operations should be to redistribute or balance the static load profile of the original 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 or minimized, and consequently, code performance improved. We illustrate this key point using again the example in Figure 5.2. Figure 5.3(a) shows the *adder* load profile for the original kernel, and Figures 5.3(b) and (c) show the load profiles for the resulting CDFGs with one and three operations speculated, respectively.[1] As can be seen, the load profile in Figure 5.3(b) has a smaller area

[1]Those load profiles were generated assuming a scheduling latency equal to the critical path of the kernel, i.e., the minimum scheduling latency.

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) to be derived for the CDFG.

In conclusion, a technique to judiciously speculate operations is required to ensure that speculation provides consistent and significant performance gains on the target centralized or clustered EPIC datapath. Accordingly, we propose an optimization phase, called Predicated Switching (described in Section 5.3), which aims at aggressively improving code performance by realizing such resource aware, targeted (or partial) ILP extraction via speculation. Specifically, given an input hyperblock and a target EPIC processor, possibly clustered, Predicated Switching optimizes control speculation under resource constraints, so as to maximize the execution performance of the resulting EPIC code.

5.3 Predicated Switching – An Algorithm for Optimizing Speculation

The algorithm will be discussed considering a target clustered machine, yet the technique can also be applied to centralized machines, since they are essentially a special case of clustered machines. The proposed algorithm makes incremental decisions on speculating individual operations, using a heuristic ordering of nodes. Specifically, the suitability of an operation for speculation is evaluated based on two metrics, Total Excess Load (TEL) and Speculative Mobility ($Spec\_\mu$), to be discussed below. Such metrics rely on a previously

defined binding function, i.e., assignment of operations to clusters (see Section 5.4), and on a target profile latency.

5.3.1 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:

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

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

$$\text{and } tf(op) = [asap(op), alap(op)]$$

$Clust\_Load_{fu,c}(t)$ is represented by the shaded regions in the load profiles in Figure 5.3.

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 available 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 5.3 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 5.3, 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 5.3.

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

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

5.3.2 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:

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

5.3.3 Ranking Function

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

$$Suit(op) = \begin{cases} TEL(op) + C.Spec\_\mu(op) & \text{if } Spec\_\mu(op) > 0 \\ 0 & \text{otherwise} \end{cases}$$

The constant multiplier C has units $\#FUs/\#steps$ and roughly represents the number of required FUs over the additional time steps (made available to the operation through speculation), so that speculation is profitable. For our experiments, $C = 1$ gave consistently good results, i.e., we assumed an extra FU over the added range, which is sufficient to execute the speculated operation.

$Suit(op)$ 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 next section. Note that we only attempt to speculate an operation if $Spec\_\mu(op) > 0$, since reduction of schedule length is likely only if speculation creates additional opportunities for early scheduling of the operation.

As an example to illustrate the calculation of $Suit(op)$, consider again the code segment of Figure 5.1, with operation a speculated. Let us say that we are interested in determining the suitability of speculating operation c. For operation c, μ is 1 (mobility is 1), EL is 0 (no shaded region to the right of the thick vertical line in time step 3), REL is 0.5 (unshaded region to left of thick vertical line in time steps 1 and 2) and $Spec\_\mu$ is 2 (difference in maximum ALAP of all its predecessors before speculation (2) and after speculation (0)). Hence, for operation c, $Suit(op) = \frac{0+0.5}{1} + 2 = 2.5$.

5.4 Optimization Flow – Generating Predicated Switching Code

Figure 5.4 shows the complete iterative optimization flow of Predicated Switching, used to generate predicated switching code. 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.

After the initialization phase is performed, the algorithm enters an iterative phase. First it decides on the next best candidate operation for speculation. The ranking function used during this phase has already been described in detail in Section 5.3. Once an operation has been selected, it is speculated by deleting the edge from its predecessor predicate define operation (related to the branching condition) and by creating a new successor predicated move operation for reconciliation, if necessary. Note that reconciliation is

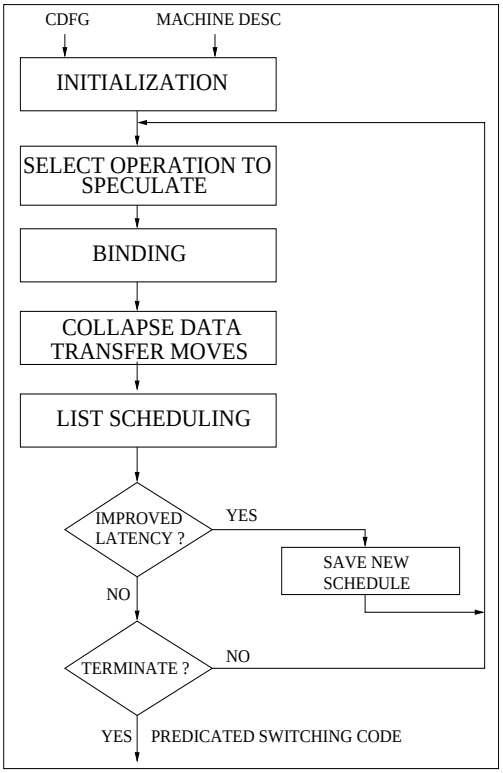


Figure 5.4: Overview of proposed optimization flow.

unnecessary if the speculated operation defines a temporary variable, i.e. if it has been speculated by direct predicate promotion (conservative speculation). In summary, depending on the specific operation being speculated, we rely on the principles of SSA-PS or direct predicate promotion to transform the Data Flow Graph (DFG).

After speculation is done, the resulting DFG containing predicated code is bound to the clusters in the datapath, using a modified version of the al-

gorithm proposed in [42].[2] The modified version of this algorithm addresses the fact that our framework has the ability to leverage data transfers across clusters. Specifically, the original algorithm adds a penalty for every predecessor of an operation that is bound to a cluster different from the cluster to which the operation itself is being tentatively bound. In our version of the algorithm, such penalty is not added if the predecessor is a predicated move operation. The reason for this modification is the fact that predicated move operations realizing the reconciliation functions may themselves be used to perform required (i.e. binding related) data transfers across clusters and thus may actually not lead to a performance penalty.

The next optimization phase applies the critical transformation of collapsing binding related move operations with reconciliation related predicated moves. Finally, a list scheduler schedules the resulting DFG. A two level priority function that ranks operations first by lower ALAP and next by lower mobility is used by the list scheduler. The list scheduler also deletes any unnecessary reconciliation operations, i.e. reconciliation operations corresponding to speculated operations that end up being scheduled *after* their corresponding predicate define instructions. Note that this deletion will not be performed if the reconciliation operation has been collapsed with a data transfer operation, since now the reconciliation operation also performs useful work (i.e. implements the required data transfer function) and cannot be deleted. Deletion

[2]Note that the computation of the ranking function at the start of an iteration uses the binding derived on the previous iteration.

will also not be performed if changes to the original binding function need to be made, as this would require a computationally intensive rebinding of operations to be performed by the list scheduler.

If execution latency is improved with respect to the previous best result, then the corresponding schedule is saved. The process continues iteratively, greedily speculating operations, until a 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.

Since the estimation of cluster loads, as well as the binding algorithm, depend on the assumed profile latency (pr), we found experimentally that is was important to search over different such profile latencies. Thus, the iterative process is repeated for various pr values, starting from the ASAP latency of the original CDFG and incrementing it upto a given percentage of the critical path (not exceeding four steps in our current version of the algorithm).

The complete algorithm has cubic asymptotic complexity, although in practice it is much lower since in general only a few operations are speculated in the final schedule. Even though such complexity is significant, we find it acceptable, particularly in the context of code generation for embedded systems (see Chapter 1).

5.5 Experimental Results

In this section, we provide experimental evidence of the improved efficiency of EPIC code generated using our proposed predicated switching optimization phase. We compare the results of Predicated Switching (denoted PS) to two baseline solutions, one generated by if-conversion based predication with conservative speculation (denoted *conservative speculation*, or CS), and the other generated by aggressively speculating all operations, including those requiring the use of SSA-PS (denoted *maximum speculation*, or MS).

Kernels extracted from the MediaBench suite of programs [43], the TI Benchmark Suite [33], the SUN VSDK Suite [32], as well as from the Mpeg4 core routines in OpenDivx [31], were used in these experiments. These select kernels consist mostly of deeply nested loop bodies that were possible to fully or partially if-convert, so as to generate hyperblocks. Most notably in the MediaBench suite of programs, we did find kernels that reused variable names on a surprisingly careless/poor way, introducing unnecessary/spurious data dependences throughout the code. Since the conservative speculation baseline approach would be penalized in those cases, and those penalties would not correspond to semantically meaningful data dependences in the actual algorithm/computations, we eliminated such poor code quality kernels from consideration.

Table 5.1 provides details on the sources of the selected kernels/hyperblocks. Characteristics of these kernels, including total number of operations, critical path (in number of time steps) and Inherent Instruction Level Parallelism (de-

ID #	Kernel	Benchmark	Main Inner Loop / Code from Function
1	Jquant2	Jpeg/Mediabench	find_nearby_colors()
2	Findpmv	Mpeg4dec/OpenDivx	Find_pmv()
3	Caldcscaler	Mpeg4enc/OpenDivx	cal_dc_scaler()
4	Collision	Collision/TI	Colldet()
5	Store	Mpeg2dec/Mediabench	Conv422to444()
6	Deblockvert	Mpeg4dec/OpenDivx	deblock_vert_default_filter()
7	Deblockhoriz	Mpeg4dec/OpenDivx	deblock_horiz_default_filter()
8	Viterbi	Viterbi/TI	Vitv32()
9	Vdkcthresh8	Vdkcthresh8/SunVSDK	vdk_c_thresh_8()
10	Lqsolve	Rasta(Lqsolve)/Mediabench	Eliminate()
11	Blockdequant	Mpeg4enc/OpenDivx	BlockDequantH263()
12	Intradcswitch	Mpeg4enc/OpenDivx	IntraDCSwitch_Decision()
13	Shortterm	Gsm/Mediabench	Fast_Short_term_synthesis_filtering()
14	Blockquant	Mpeg4enc/OpenDivx	BlockQuantH263()
15	Findcbp	Mpeg4enc/OpenDivx	FindCBP()

Table 5.1: Kernel Characteristics (1/2).

noted I-ILP) are shown in Table 5.2. The I-ILP value is computed by dividing a kernel's total number of operations by the number of operations on its critical path. The kernels in Table 5.2 have been ordered by decreasing I-ILP, since this number provides an indication of their resource demands.

The test kernels were manually compiled to a 3-address like intermediate representation that captured all dependences between instructions. For each experiment, this intermediate representation together with a parametrized description of the target EPIC machine datapath was input to our compiler optimization framework (see Figure 5.4). Six different EPIC machine configurations (four of them clustered) were used for the experiments. All the configurations were chosen to have relatively small clusters, since these ma-

ID #	Kernel	# Inst	Critical Path (# steps)	I-ILP
1	Jquant2	82	12	7.45
2	Findpmv	99	20	5.21
3	Caldcscaler	29	6	4.83
4	Collision	33	10	3.67
5	Store	39	14	3.55
6	Deblockvert	79	28	3.12
7	Deblockhoriz	69	27	3
8	Viterbi	44	17	2.93
9	Vdkcthresh8	12	7	2.4
10	Lqsolve	15	8	2.14
11	Blockdequant	31	22	2
12	Intradcswitch	12	7	1.71
13	Shortterm	16	12	1.7
14	Blockquant	13	10	1.63
15	Findcbp	14	10	1.56

Table 5.2: Kernel Characteristics (2/2).

chines are in general more attactive from a power/performance standpoint (see Chapter 1) and more challenging to compile to, as more data transfers may need to be scheduled. The FU's in each cluster include ALUs, Multipliers, 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 3 Clusters |1|1|1|1| specifies a datapath with three clusters, each cluster with 1 FU of each type.[3] As before,

[3]Although, for conciseness, we present results for machines with identical clusters, our framework is general and can handle machines with heterogeneous clusters as well (see Section 5.3).

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.

	1 Cluster |1|1|1|1|1| (Centralized)					2 Clusters |1|1|1|1|1|				
ID #	Kernel Latency	% Impr. of PS over		# Spec Ops/ Mergings		Kernel Latency	% Impr. of PS over		# Spec Ops/ Mergings	
	CS/MS/PS	CS	MS	MS	PS	CS/MS/PS	CS	MS	MS	PS
1	50/73/48	4.0	34.2	10	0	32/42/27	15.6	35.7	21/15	7/2
2	92/92/92	0.0	0.0	3	0	50/49/46	8.0	6.1	38/19	11/0
3	28/28/28	0.0	0.0	1	0	14/17/14	0.0	17.7	7/0	0/0
4	27/26/26	3.7	0.0	2	1	18/17/17	5.6	0.0	2/0	1/0
5	31/30/30	3.2	0.0	2	0	24/28/23	4.2	17.9	8/6	0/0
6	52/52/52	0.0	0.0	3	0	46/42/40	13.0	4.8	11/9	8/6
7	58/53/53	8.6	0.0	10	7	55/47/45	18.2	4.3	13/8	13/12
8	29/29/29	0.0	0.0	0	0	21/20/20	4.8	0.0	4/3	2/2
9	13/13/13	0.0	0.0	0	0	9/9/9	0.0	0.0	0/0	0/0
10	13/12/11	15.4	8.3	5	2	11/11/10	9.1	9.1	6/3	2/1
11	29/29/28	3.4	3.4	3	3	29/29/28	3.4	3.4	3/0	3/2
12	11/11/11	0.0	0.0	0	0	11/10/10	9.1	0.0	3/0	3/0
13	18/16/16	11.1	0.0	4	3	18/16/16	11.1	0.0	4/0	3/0
14	15/14/14	6.7	0.0	1	1	15/14/14	6.7	0.0	1/0	1/1
15	15/13/13	13.3	0.0	2	2	12/12/12	0.0	0.0	6/0	0/0

Table 5.3: Performance: Predicated Switching (1/4).

Note that the two baseline solutions can be also generated by Predicated Switching, if it decides not to speculate operations requiring the use of SSA-PS or decides to speculate all such operations. Predicated Switching can however generate a range of additional candidate solutions, speculating only a sub-set of operations. Thus, the experimental results presented in this section allow for an empirical assessment of the effectiveness of our proposed resource-aware selective speculation technique relative to two *resource-unaware*

	2 Clusters |1|1|1|1|					3 Clusters |1|1|1|1|				
ID #	Kernel Latency	% Impr. of PS over		# Spec Ops/ Mergings		Kernel Latency	% Impr. of PS over		# Spec Ops/ Mergings	
	CS/MS/PS	CS	MS	MS	PS	CS/MS/PS	CS	MS	MS	PS
1	32/42/27	15.6	35.7	21/15	7/2	32/42/25	21.9	40.5	21/15	9/3
2	50/49/46	8.0	6.1	38/19	11/0	50/49/41	18.0	16.3	38/19	22/11
3	14/17/14	0.0	17.7	7/0	0/0	12/16/12	0.0	25.0	6/0	0/0
4	18/17/17	5.6	0.0	2/0	1/0	18/17/17	5.6	0.0	2/0	1/0
5	24/28/23	4.2	17.9	8/6	0/0	24/28/23	4.2	17.9	8/6	2/2
6	46/42/40	13.0	4.8	11/9	8/6	46/42/40	13.0	4.8	11/9	8/6
7	55/47/45	18.2	4.3	13/8	13/12	55/47/45	18.2	4.3	13/8	13/12
8	21/20/20	4.8	0.0	4/3	2/2	21/20/20	4.8	0.0	4/3	2/2
9	9/9/9	0.0	0.0	0/0	0/0	9/9/9	0.0	0.0	0/0	0/0
10	11/11/10	9.1	9.1	6/3	2/1	10/11/10	0.0	9.1	6/3	0/0
11	29/29/28	3.4	3.4	3/0	3/2	29/29/28	3.4	3.4	3/0	3/2
12	11/10/10	9.1	0.0	3/0	3/0	11/10/10	9.1	0.0	3/0	3/0
13	18/16/16	11.1	0.0	4/0	3/0	18/16/16	11.1	0.0	4/0	3/0
14	15/14/14	6.7	0.0	1/0	1/1	15/14/14	6.7	0.0	1/0	1/1
15	12/12/12	0.0	0.0	6/0	0/0	12/12/12	0.0	0.0	6/0	3/0

Table 5.4: Performance: Predicated Switching (2/4).

baselines, each generating a solution in one of the boundaries of the range covered by Predicated Switching. In fact, in our flow we explicitly include the solutions generated at these boundaries. In order to fairly compare the techniques, the baseline CS and MS code was generated using the same modified binding algorithm and list scheduler used in our proposed flow incorporating the Predicated Switching phasing. Moreover, MS also performs the merging of reconciliation and data transfer operations discussed in Section 4.3. Detailed results of our experiments (15 kernels compiled for 6 different machine configurations) are given in Tables 5.3, 5.4, 5.5 and 5.6, and will be discussed in detail below.

	1 Cluster |1|1|2|2| (Centralized)					2 Clusters |1|1|2|2|				
ID #	Kernel Latency	% Impr. of PS over		# Spec Ops/ Mergings		Kernel Latency	% Impr. of PS over		# Spec Ops/ Mergings	
	CS/MS/PS	CS	MS	MS	PS	CS/MS/PS	CS	MS	MS	PS
1	50/73/48	4.0	34.2	10	0	32/39/27	15.6	30.8	29/16	12/5
2	92/92/92	0.0	0.0	3	0	49/49/46	6.1	6.1	43/32	11/0
3	28/28/28	0.0	0.0	1	0	14/17/14	0.0	17.6	7/0	4/0
4	19/18/18	5.3	0.0	2	1	15/14/14	6.7	0.0	2/0	1/0
5	29/29/29	0.0	0.0	2	0	23/24/22	4.3	8.3	10/7	0/0
6	51/51/51	0.0	0.0	2	0	41/36/36	12.2	0.0	12/11	12/11
7	47/44/44	6.4	0.0	12	6	47/39/38	19.1	2.6	13/8	13/10
8	29/29/29	0.0	0.0	0	0	19/20/19	0.0	5.0	5/5	0/0
9	8/8/8	0.0	0.0	0	0	8/8/8	0.0	0.0	0/0	0/0
10	13/11/11	15.4	0.0	4	2	11/11/9	18.2	18.2	6/3	6/3
11	24/24/23	4.2	4.2	3	3	24/24/23	4.2	4.2	3/0	3/0
12	11/11/11	0.0	0.0	0	0	11/10/10	9.1	0.0	3/0	3/0
13	16/14/14	12.5	0.0	4	3	16/14/14	12.5	0.0	4/0	3/0
14	12/11/11	8.3	0.0	1	1	12/11/11	8.3	0.0	1/0	1/0
15	15/13/13	13.3	0.0	2	2	12/12/12	0.0	0.0	6/0	0/0

Table 5.5: Performance: Predicated Switching (3/4).

Tables 5.3 and 5.4 summarize the first three sets of experiments performed for the 15 benchmarks. The left sub-table of Table 5.3 shows the results obtained for a target centralized machine with one FU of each type (that is, one ALU, one Multiplier, one Load/Store Unit and one Move Port). The second sub-table summarizes the results obtained for a 2-cluster machine, where each cluster has also one FU of each type. Thus, as we go left to right in Table 5.3, the width of the target machine increases, through the addition of one extra cluster. Similarly, the left sub-table of Table 5.4 shows the results for the 2-cluster machine (repeated for convenience) and the right sub-table those for the 3-cluster machine. The first column of each machine configu-

	2 Clusters |1|1|2|2|					3 Clusters |1|1|2|2|				
ID #	Kernel Latency	% Impr. of PS over		# Spec Ops/ Mergings		Kernel Latency	% Impr. of PS over		# Spec Ops/ Mergings	
	CS/MS/PS	CS	MS	MS	PS	CS/MS/PS	CS	MS	MS	PS
1	32/39/27	15.6	30.8	29/16	12/5	28/30/23	17.9	23.3	27/27	17/5
2	49/49/46	6.1	6.1	43/32	11/0	44/40/35	20.5	12.5	41/25	38/24
3	14/17/14	0.0	17.6	7/0	4/0	11/13/11	0.0	15.4	6/0	0/0
4	15/14/14	6.7	0.0	2/0	1/0	14/14/14	0.0	0.0	2/0	0/0
5	23/24/22	4.3	8.3	10/7	0/0	22/24/21	4.5	12.5	10/7	5/5
6	41/36/36	12.2	0.0	12/11	12/11	41/36/36	12.2	0.0	12/11	12/11
7	47/39/38	19.1	2.6	13/8	13/10	47/39/38	19.1	2.6	13/8	13/10
8	19/20/19	0.0	5.0	5/5	0/0	19/20/19	0.0	5.0	5/5	0/0
9	8/8/8	0.0	0.0	0/0	0/0	8/8/8	0.0	0.0	0/0	0/0
10	11/11/9	18.2	18.2	6/3	6/3	10/10/9	10.0	10.0	7/5	3/0
11	24/24/23	4.2	4.2	3/0	3/0	24/24/23	4.2	4.2	3/0	3/0
12	11/10/10	9.1	0.0	3/0	3/0	11/10/10	9.1	0.0	3/0	3/0
13	16/14/14	12.5	0.0	4/0	3/0	16/14/14	12.5	0.0	4/0	3/0
14	12/11/11	8.3	0.0	1/0	1/0	12/11/11	8.3	0.0	1/0	1/0
15	12/12/12	0.0	0.0	6/0	0/0	12/12/12	0.0	0.0	6/0	3/0

Table 5.6: Performance: Predicated Switching (4/4).

ration sub-table (with header *Kernel Latency*) gives the latency (in number of time steps) of the code generated via conservative speculation (CS), maximum speculation (MS), and predicated switching (PS), in this order. The second main column of each sub-table (with header *% Impr. of PS over*) gives the latency improvement of Predicated Switching code with respect to CS (i.e., conservative speculation) code and MS (i.e., maximum speculation) code. The third main column (with header *# Spec Ops / Mergings*) gives the total number of operations speculated via SSA-PS, and the number of such operations that were actually merged with inter-cluster data transfers, for both, maximum speculation (MS) code, and predicated switching (PS) code. Note that

similar results for the conservative speculation (CS) baseline are not explicitly presented, since they would be 0/0 for all experiments (see Chapters 2 and 4). Also note that there are no mergings on a *centralized* machine, as there are no data transfers across clusters.

Tables 5.5 and 5.6 are similar to Tables 5.3 and 5.4, except that each cluster contains one additional Load/Store Unit and one additional Move Port (represented as |1|1|2|2|).

5.5.1 Overall Result Statistics

The overall statistics for the 90 experiments summarized in Tables 5.3, 5.4, 5.5 and 5.6 (45 experiments reported in Tables 5.3 and 5.4, and 45 experiments reported in Tables 5.5 and 5.6) are as follows. Predicated switching improved performance with respect to CS (conservative speculation) in 68% of the experiments, and improved performance with respect to MS (maximum speculation) in 41% of the experiments. Predicated switching code improved performance with respect to *both* baseline solutions (i.e., MS *and* CS code) in 33% of the experiments and with respect to *at least one* of the baseline solutions in 76% of the experiments. The average improvement of predicated switching code over CS was 6.6% and over MS was 5.4%. The maximum performance improvement of predicated switching over CS was 21.9%, and the maximum performance improvement of predicated switching over MS was 40.5%, both achieved for the highest I-ILP benchmark (Kernel 1) when targeted to one of the 3-cluster machines. In the sequel, we discuss those results

in more detail.

5.5.2 Analysis of Results for Centralized Machines

We start our analysis by considering the results achieved on the centralized machine with a |1|1|1|1| datapath configuration, which are summarized on the left sub-table of Table 5.3. Predicated switching improved the performance with respect to *both* baseline approaches in 20% of the benchmarks (i.e., in 3 out of the 15 kernels) and improved the performance with respect to at least one of the baselines in 60% of the benchmarks (i.e., in 9 kernels).

We consider first the three kernels where predicated switching did better than both baseline approaches. For these cases, the maximum performance improvement over CS was 15.4% and the maximum performance improvement over MS was 34.3%. Kernel 1 (Jquant2), which has an I-ILP of 7.45 operations (much higher than the actual width of the target centralized machine), and a critical path of 12 scheduling steps (see Table 5.2), is one such kernel. The small latency improvement of predicated switching over CS for this kernel (48 versus 50 steps) was achieved by rearranging/widening the scheduling ranges of only a few select operations, in this case via conservative predicate promotion, not SSA-PS. (Note that, as indicated in column 3 of this sub-table, predicated switching did not speculate operations via SSA-PS in this instance.) This experiment thus shows that, even when the final code latency is not critical-path bound (in this case, the critical path is only 12 steps), selective speculation can still lead to a better load distribution, that is, a load distribution that enables

and/or facilitates the derivation of code scheduling solutions that more effectively utilize a machine's resources.[4] In contrast, the latency improvement of predicated switching over MS for Kernel 1 (48 versus 73 steps, a 34.3% improvement) reveals that the excessive speculation (via SSA-PS) realized on this baseline solution has seriously hurt performance, with the extra necessary reconciliation operations (10 operations, in this case) actually amplifying the already severe resource bottlenecks.[5]

Indeed, the expected trend behavior for kernels whose I-ILP is much larger than the width of the target machine is to have MS doing worse than CS, as in the above experiment. In contrast, CS is expected to do worse than MS for critical-path bound kernels, that is, kernels for which the final code latency is defined (to a large extent) by the longest chain of data dependences between code operations. This is so because, in such cases, SSA-PS driven speculation may, both, eliminate/break some such dependences (thus decreasing the critical path), as well as alleviate resource bottlenecks in specific/narrow scheduling regions.

Naturally, such generic trends are based on a very simplified view of the

[4]Note that, as indicated in Section 2.2, the baseline CS approach predicate promotes (up-front) all operations that can be speculated with no added cost (i.e., that can be conservatively predicate promoted), while in our predicated switching approach, even such zero-cost speculation decisions are done incrementally, driven by the proposed ranking functions. Thus, final code scheduling/compaction tends to be simpler in predicated switching code, since the operation's scheduling ranges are not widened (by conservative or by SSA-PS speculation), when the current narrower ranges lead to a better load distribution. This makes the task of the heuristic resource constrained scheduler easier.

[5]Note that, as indicated in Section 5.4, the list scheduler removes any unnecessary reconciliation operations.

problem, and may not necessarily materialize in certain cases, since the actual *number* and *type* of operations speculatable via SSA-PS, as well as their *true data dependences* to other operations in the kernel, are also extremely important in terms of the actual profitability of speculating operations via SSA-PS. Indeed, the specific operations speculated by predicated switching can vary quite significantly with an increase in datapath resources. Namely, on machines with more resources, the TEL term in the speculation cost function (for each operation) will be different. Moreover, predicated switching speculates incrementally, in a greedy fashion, and uses an updated load profile each time a new operation is selected for speculation. This ensures that the next most suitable candidate for speculation is selected using updated information on previously speculated operations. Thus, the choice and sequence of operations to be speculated may vary quite significantly with increases/decreases in datapath resources.

Kernel 10 (Lqsolve) is another kernel for which predicated switching does better than both CS and MS, on the centralized |1|1|1|1| machine. Note that it has a much lower I-ILP than Kernel 1 (only 2.14 operations), and a relatively shorter critical path (8 steps). Thus, in contrast to the previous case (Kernel 1), predicated switching in this case did speculate 2 operations via SSA-PS, so as to better redistribute the load over the kernel's critical path. Accordingly, the latency penalty incurred by CS's inability to speculate such operations is now more severe than the penalty incurred by MS's excessive speculation (15.4% vs. 8.3%), which is consistent with the generic trend

alluded to above.

Kernel 11 (Blockdequant), yet another kernel for which predicated switching does better than both CS and MS on the centralized |1|1|1|1| machine, has a lower I-ILP than Kernel 10 (2 operations) with a much longer critical path (22 steps). The improvements for this kernel over both baselines was the same (3.5%). Note that although both MS and predicated switching speculated the same number of operations (three) by SSA-PS, they speculated different numbers of operations by direct speculation (conservative speculation) thus leading to predicated switching achieving a better schedule.

The remaining results for the |1|1|1|1| centralized machine are as follows. For six of the fifteen benchmarks (Kernels 4, 5, 7, 13, 14 and 15), predicated switching did better than CS (with 3.2% to 11.1% performance improvements), and yet achieved the same performance of MS. Note that for these six kernels, though MS achieved the same performance as predicated switching, it almost always speculated more operations than predicated switching. In other words, predicated switching achieved the same performance by speculating less operations leading to more power/energy efficient code.

Finally, for the remaining kernels, predicated switching and the two baseline techniques delivered the exact same performance. For all these kernels, predicated switching simply did not find it profitable to speculate any operations through SSA-PS. Also, the small number of operations speculated by MS in these kernels, did not affect performance. Kernel 9 (Vdkcthresh8) has no SSA-PS speculatable operations, and thus, the solutions generated by

the three techniques are likely to be similar for any target machine. Kernel 8 (Viterbi) has six SSA-PS speculatable operations (out of 44), yet predicated switching chose not to speculate any operations by SSA-PS, i.e. predicated switching was neither effective nor harmful for the centralized |1|1|1|1| machine. This will however change for some of the other target machines considered in our experiments.

The overall statistics on the results achieved for the centralized machine with a |1|1|2|2| datapath configuration (summarized on the left subtable of Table 5.5) are identical to those obtained for the |1|1|1|1| machine. In summary, on centralized machines, our proposed technique delivers consistent gains over CS, and frequently the same performance as MS, but does so by speculating less operations. However, for clustered machines, we see consistent performance gains over both baselines, suggesting that predicated switching is particularly suited for such machines, as we shall see in the next section.

5.5.3 Analysis of Results for Clustered Machines

Overall, for the 2-cluster machines (30 experiments shown in the right sub-tables of Tables 5.3 and 5.5), predicated switching improved performance with respect to *both* baseline approaches in 43% of the cases (that is, in 13 out of the 30 experiments) and improved performance with respect to at least one of the baselines in 87% of the cases (i.e., in 26 out of the 30 experiments). For the 3-cluster machines (30 experiments shown in the right sub-tables of Tables 5.4 and 5.6), predicated switching improved performance with respect

to *both* baseline approaches in 40% of the cases, and with respect to at least one of the baselines in 83% of the cases. Note that both such statistics are much better than those obtained for the centralized machines, thereby suggesting that predicated switching is particularly effective in the context of high-ILP multicluster machines. The maximum performance improvement of predicated switching over CS on a multicluster machine was 21.9%, and over MS was 40.5%, both obtained for Kernel 1 (the kernel with the highest I-ILP) on the 3-cluster |1|1|1|1| machine. Over all multicluster machines, the average improvement of predicated switching code over CS was 7.7% and over MS was 6.6%. In nearly 40% of the experiments with clustered machines, predicated switching was able to merge SSA-PS related reconciliation functions with data transfers aimed at distributing load among clusters, thus amortizing the latency cost of both such types of operations.[6]

We start our detailed analysis, by contrasting the results obtained for the centralized |1|1|1|1| target machine with the results obtained for two wider *multicluster* machine configurations, namely, the machine with two |1|1|1|1| clusters, and the machine with three |1|1|1|1| clusters (see Tables 5.3 and 5.4). In terms of generic trends, predicated switching should do comparatively better than CS on wider (multicluster) machines (when compared to the original centralized machine), because a wider machine offers more opportunities to

[6]Note that MS baseline solutions typically merge more operations than predicated switching, because they contain more operations speculated via SSA-PS, yet the cost of such excessive speculation cannot, for most cases, be fully absorbed just by performing such mergings.

profitably explore aggressive ILP extraction. Similarly, the relative performance degradation due to MS's excessive (SSA-PS driven) speculation should decrease with respect to that observed on the centralized (lower ILP) machine, since higher ILP machines have more resources to alleviate potential bottlenecks created by SSA-PS's reconciliation operations. As mentioned above, though, this trend is not necessarily guaranteed to manifest in all examples, since many other complex factors contribute to the profitability of speculating operations via SSA-PS, including (now) the ability to merge SSA-PS's reconciliation functions with data transfers between clusters.

Kernel 1 (Jquant2), the benchmark with the highest I-ILP (7.45), partially exhibits the expected trend behavior discussed above. Specifically, the latency penalty of CS with respect to predicated switching increases as we move from the centralized to the 3-cluster machine (i.e. from 4% to 15.6% to 21.9%). Note that predicated switching speculates more operations as the width of the machine increases, suggesting that the proposed technique is able to take advantage of the increased machine resources provided. Also note that, on the 2-cluster machine as well as on the 3-cluster machine, predicated switching collapses roughly one third of the SSA-PS's reconciliation operations with required data transfers.

However, for Jquant2 (Kernel 1), the relative penalty of MS increases from 34.3% to 35.7% to 40.5% as the number of clusters is increased. This relative performance degradation of MS with respect to predicated switching on the wider machine can be explained as follows. On the wider 3-cluster

machine, the predicated switching technique was able to improve performance with respect to the smaller (2-cluster) machine by speculating only a few more operations (specifically, two). In contrast, the MS baseline technique (with all operations speculated) was unable to improve performance on the 3-cluster machine due to the added complexity and asymmetries created by the large number of reconciliation functions. Similar considerations explain the slight increase in relative penalty of MS w.r.t. predicated switching when moving from the centralized machine to the 2-cluster machine.

There is yet another important observation to be made with respect to Kernel 1. Note that it has 82 operations, and yet a critical path of only 12 operations. Thus, as expected, when doubling the width of the machine (i.e., moving from a centralized to a 2-cluster machine), the schedule latency of the resulting predicated switching code decreased by about 44% (namely, from 48 to 27 steps). However, when the width of the machine was increased by an additional 33% (i.e., when a third cluster was added to the machine), the latency decreased by only 7% (namely, from 27 to 25 steps). This latency saturation is partially due to the fact that Kernel 1 has a substantial number of early read operations which, due to the limited number of load/store units in the machine, must be serialized, thus precluding a more aggressive code compaction[7]. This prompted us to consider a second set of target machine configurations (shown in Tables 5.5 and 5.6), where each cluster has now two

[7]Recall that Read operations take two steps in our target machines, being thus particularly critical to performance.

load/store units and two move ports. As can be seen, while the latency of Kernel 1 is identical to that achieved for the centralized and the 2-cluster machine configurations considered previously (i.e., 48 and 27 steps), the effect of the extra load/store unit (and of the extra move port) is clearly felt on the 3-cluster machine. Namely, an increase on 33% on the machine width delivers now a performance improvement of nearly 15% on the predicated switching code.

We proceed with our detailed analysis of the results obtained for clustered machines, by considering Kernel 2 (Findpmv). Kernel 2 has the second highest I-ILP value (5.21), and illustrates very sharply the "cluster-friendliness" of predicated switching. Indeed, for this kernel, the relative performance improvement of predicated switching code over the two baselines improves substantially as we move from centralized to multicluster machines. Specifically, for the three machines in Tables 5.3 and 5.4 (with |1|1|1|1| clusters), the improvement of standard predication over conservative speculation and maximum speculation is 0% / 0% for the centralized machine, 8% / 6.1% for the 2-cluster machine, and 18% / 16.3% for the 3-cluster machine. Similarly, for the three machines in Tables 5.5 and 5.6 (with |1|1|2|2| clusters), the improvement of standard predication over conservative speculation and maximum speculation is 0% / 0% for the centralized machine, 6.1% / 6.1% for the 2-cluster machine, and 20.5% / 12.5% for the 3-cluster machine. The results for Kernel 1 (discussed previously) show similar "cluster-friendly" trends.

Kernel 3 (Caldcscaler), with the third highest I-ILP value (4.83) shows

considerable improvement in performance over MS for all the multicluster machine configurations but no improvement over CS in any of the cases. This is explained by the fact that this kernel has very few SSA-PS speculatable operations and a large number of operations that can be direct speculated (conservatively speculated) and so the solution found by predicated switching is essentially the same as the CS solution.

Consider now the benchmark with the lowest I-ILP (1.56), Kernel 15 (Findcbp).The relative effectiveness of predicated switching with respect to the two baselines actually decreases for this benchmark, as we move from centralized to multicluster machines. This behavior is however easily explained by the fact that, while the |1|1|1|1| and |1|1|2|2| centralized machines do not have enough parallelism/resources to deliver the minimum (critical path) latency for this kernel, the 2-cluster machines already provide sufficient ILP, and thus, our resource-aware speculation technique becomes less critical to performance for the 3-cluster machines. Indeed the latency attained by this kernel on the multicluster machines is within 2 steps of its critical path latency. These extra two steps required by the clustered machines represent the *performance overhead* typically incurred by clustered architectures [36]. Similar ILP considerations explain the fact that the performance achieved for several of the kernels on 2-cluster machine configurations could not be further improved on the corresponding 3-cluster machine configurations.

In summary, the best and most consistent gains resulting from resource-aware selective speculation are delivered for kernels that have high I-ILP with

respect to the issue-width of target machine (be it centralized or multicluster) and a large number of speculatable operations, some of which are on the critical path. Kernels Jquant2, Findpmv, Store, Deblockvert, and Deblockhoriz exhibit such broad characteristics, thus enabling predicated switching to deliver good gains across most of the target machines considered in the experiments. For kernels with less I-ILP or less speculatable operations, the gains achieved by predicated switching may be also substantial, at least for certain machines, see e.g., Lqsolve, which has an I-ILP of only 2.14, and yet allows 6 out of its 15 operations to be speculated via SSA-PS, thus enabling an efficient load redistribution for certain machine configurations.

5.5.4 Impact on Code Size and Register Pressure

The increase in harmonic mean of the code size[8] produced by predicated switching with respect to CS, over all experiments, was determined to be 7.1%. This increase is extremely small and can be explained by the fact that, in general, predicated switching needs to selectively speculate only a small number of operations in order to aggressively improve performance on the target machine. Moreover, on multicluster machines, code size was actually reduced or remained the same (as compared to CS code) in 48% of the experiments, upto a maximum reduction of 8.5%. Such code size increases in CS (with respect to predicated switching) result from additional data transfer operations introduced by the binding algorithm, so as to distribute the load

[8]Code size was measured in number of micro-instructions (operations).

over the clusters on the more congested regions of the schedule. Finally, predicated switching produces smaller code than MS for almost all cases, as would be expected, since MS speculates aggressively, and hence consistently adds the largest number of reconciliation operations to the code.

Moreover, for 42% of the experiments, register pressure on the predicated switching code was decreased or remained unchanged with respect to CS code. Register pressure was determined by computing the maximum number of live webs[9] on the register file of each of a machine's clusters. These results strongly suggest that predicated switching is able to use the distributed register spaces present in the clustered datapath effectively. In other words, the cost incurred by predicated switching in creating extra live variables through renaming is largely compensated for by efficiently distributing load (and hence register demand) over the clustered datapath. Also note that MS produces typically the highest register pressure among all competing techniques. In summary, thus, predicated switching gives significant performance gains over both CS and MS, with negligible impact on both register pressure and code size.

5.5.5 Impact of Predicated Switching on Overall Program Performance

Kernel Lqsolve, extracted from the least squares solver in the *Rasta* distribution available in Mediabench, is one of the kernels where predicated

[9] A web is a maximal set of def-use chains. A def-use chain connects the definition of a variable to all its uses (see [50] for details).

switching performed the best, with performance improvements with respect to CS and MS of 18.2%, 15.4%, 10.0%, and 9.1% (depending on the target machine). We profiled the solver and found that, in average, more than 48% of the solver's total execution time is spent on Kernel Lqsolve. Thus, the program speed-up that would be obtained just by applying predicated switching to this single kernel, given the above performance improvements, would be between 1.087 (i.e., a 8.7% speed-up) and 1.043 (i.e., a 4.3% speedup). We have also profiled the *gsm decoder* program available in Mediabench, and found that on average more than 39% of the program's total execution time is spent on Kernel Shortterm (Kernel 13 in Table 5.1). For all machine configurations, except the |2|2|2|2| centralized machine, predicated switching code for Kernel 13 outperformed the CS code by 12.5% to 11.1%, which would give a corresponding program speedup of 1.048 (4.8%) to 1.043 (4.3%) with respect to conservative speculation code. We have also profiled the *Jpeg decompresser* program (djpeg) distributed with Mediabench, and found that Kernel Jquant2 (Kernel 1 in Table 5.1) takes on average about 12% of the program's total execution time. Note that this is the kernel where predicated switching performs the best. Namely, it delivers relative performance improvements (with respect to the baselines) as high as 40.5%, which would correspond to a program speedup of 1.048 (4.8%). Finally, we have also profiled the *Mpeg2 decoder* program distributed with Mediabench, and found that Store (Kernel 6 in Table 5.1) takes on average about 1% of the program's total execution time, and thus, the result of optimizing just this single kernel via predicated switching would not

be noticeable in the performance of the decoder.

	3 Clusters |1|1|1|1|1|				
ID	Time (secs)			Ratio PS ov.	
#	CS	MS	PS	CS	MS
1	2.1	4.2	84.9	40.8	20.3
2	3.7	5.9	137.5	37.2	23.2
3	0.5	0.8	26.2	58.1	34.9
4	0.6	0.7	26.6	46.7	38.0
5	0.7	1.2	42.1	56.9	34.2
6	2.7	3.3	108.3	39.5	33.2
7	3.1	3.2	120.9	39.2	37.7
8	1.0	1.2	45.7	44.8	37.1
9	0.2	0.3	10.5	47.9	37.6
10	0.2	0.4	16.2	70.5	38.6
11	1.0	1.3	43.7	43.7	35.0
12	0.2	0.3	12.0	49.9	36.3
13	0.4	0.5	19.6	53.0	38.5
14	0.2	0.3	12.5	54.4	39.1
15	0.3	0.4	17.0	62.9	43.6
HM				53.3	37.4

Table 5.7: Compilation Time.

The remaining kernels were extracted from libraries of functions intended to facilitate the development of application programs. Thus, no profiling data was possible to derive for such kernels, since those libraries can be used in many different ways by different application programs. However, as indicated previously, most of such kernels are found on deeply nested loops, being thus likely to have a substantial impact in performance when their corresponding functions are used in application programs.

5.5.6 Compilation Time Statistics

Table 5.7 shows the compilation times taken by the techniques being compared for the 3-cluster |1|1|1|1| set of experiments. The machine used for all experiments was a Sun Ultra 5 workstation with a clock speed of 266 MHz. We provide the compilation times for a target multicluster machine as these times are more indicative of the scalability of the techniques. Note that the actual ratio of execution time with respect to CS is roughly constant with large variations in the number of operations in the kernel, suggesting that, for reasonable sized kernels, the execution time of our method will be approximately one and a half orders of magnitude larger than the baseline techniques. We conclude by noting that our current implementation of the algorithm could be largely optimized, to produce shorter compile times.

Chapter 6

Predicated Switching incorporating Software Pipelining

This chapter describes how software pipelining, a powerful ILP extraction technique, can be seamlessly incorporated in our ILP extraction flow [58] [57]. First, we describe software pipelining and the benefits that result from this technique. We then discuss the various tradeoffs that can be explored when speculation and software pipelining are integrated in a combined framework, and describe how software pipelining is incorporated in our optimization framework. Finally, we present our experimental results.

6.1 Software Pipelining – Background

Software pipelining [44] [41] is an effective ILP extraction technique that increases the throughput of loop bodies by overlapping multiple iterations in a single execution cycle. Software pipelining allows for "breaking" of true data dependences in a loop body by moving/retiming operations that are connected by such dependences to different iterations, thus enabling their concurrent execution.

The number of different iterations of the loop body executing concur-

rently is called the number of *pipe stages* in the schedule and the rate at which a new execution cycle is initiated is called the *initiation interval* of the schedule.

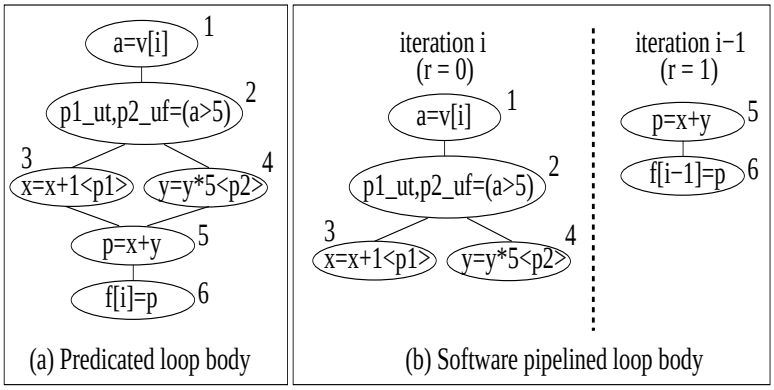


Figure 6.1: Example predicated loop body (a) and its software pipelined version (b).

We illustrate software pipelining using the example predicated code segment, shown in Figure 6.1(a), which represents a loop body, with loop index i that is incremented from 1 to N. A software pipelined version of this loop is shown in Figure 6.1(b) – in this example, operations 5 and 6 were retimed (i.e., delayed) by one iteration. Thus, two pipe stages were created, one corresponding to iteration i and the other to iteration $i-1$. The corresponding retiming function (r) is also indicated in Figure 6.1(b).

The corresponding software pipelined schedule for a target centralized machine with 2 ALUs and 2 Load/Store units, is shown in Figure 6.2. The resulting schedule attains a steady state initiation interval of 4 time steps, which is 57% of the critical path (and thus the best attainable initiation interval) of the original loop body. This reduction of critical path is possible due to the

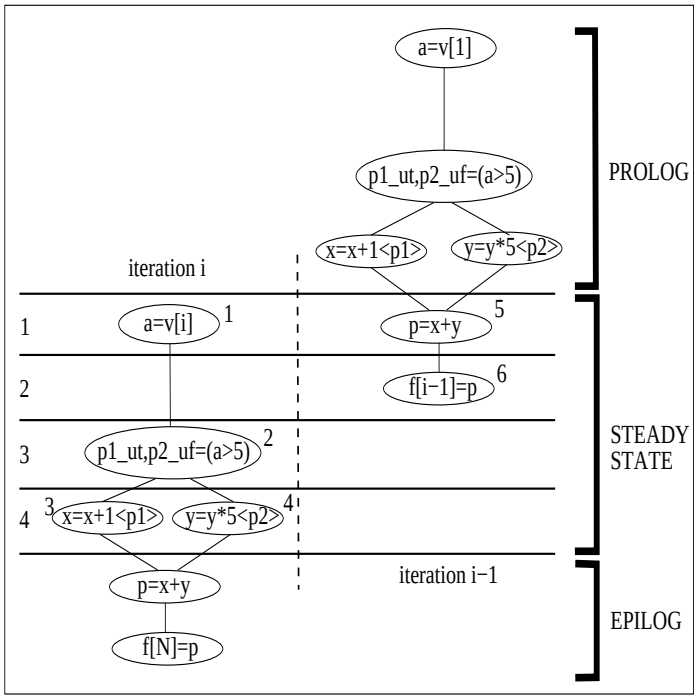


Figure 6.2: Schedule for predicated loop body software pipelined for two pipe stages.

"breaking" of the true data dependences between operation 3 and 4 and operation 5. Specifically, operation 5 now computes p using the values of x and y defined in the previous iteration, and thus can (and should) be scheduled before operations 3 and 4 (which define the new values for iteration i).

Although software pipelining can provide significant performance enhancements, it comes at the cost of increased code size. Namely, extra code is needed to fill up the pipe (called prolog) before steady state execution is reached, and drain the pipe (called epilog) after steady state execution completes. Code size is increased by a factor of P, where P is the number of

pipe stages in the software pipelined schedule. This extra code (shown in Figure 6.2 for the example loop body), can be detrimental to performance, especially in the context of embedded systems, where program memories are small. Moreover, software pipelined loops cause an increase in register pressure due to variables being live across different iterations [2]. These deleterious effects become more pronounced when the number of pipe stages in the software pipelined schedule is increased. We will examine these issues in more detail in the next section.

6.2 Speculation and Software Pipelining

Speculation and software pipelining are ILP extraction techniques with very different characteristics and associated costs. In this section, we will discuss these costs and show how our proposed framework can trade-off the relative merits of each technique.

The costs associated with speculation are the introduction of reconciliation operations. On the other hand, as seen in the previous section, software pipelining costs include increase in code size due to the prolog and epilog. Note that the prolog and epilog can be eliminated on EPIC architectures by the use of steady state predicates. However, the complexity of such predicates increases with an increase in the number of pipe stages in the software pipelined schedule. Therefore, keeping the number of pipe stages in the schedule to a minimum is clearly desirable [2].

Our proposed strategy utilizes software pipelining to aggressively in-

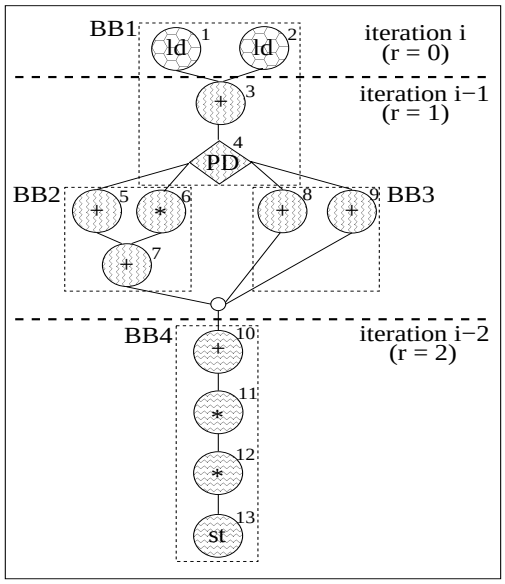


Figure 6.3: Example predicated loop body retimed for 3 pipe stages.

crease the ILP of loops by "breaking" critical true data dependences – note that speculation alone cannot achieve this. Speculation is then used to further extract additional ILP from the kernel. The integration of speculation and software pipelining in a combined framework thus allows us to generate low initiation interval solutions, potentially using fewer pipe stages than would be possible if software pipelining were used alone.

To illustrate this point consider the example predicated code segment shown in Figure 6.3, which represents a loop body. The example code segment is software pipelined for a maximum of three pipe stages, using the retiming function indicated in Figure 6.3. For clarity, the operations on each pipe stage are shaded differently (see Figure 6.3). A schedule for this example code

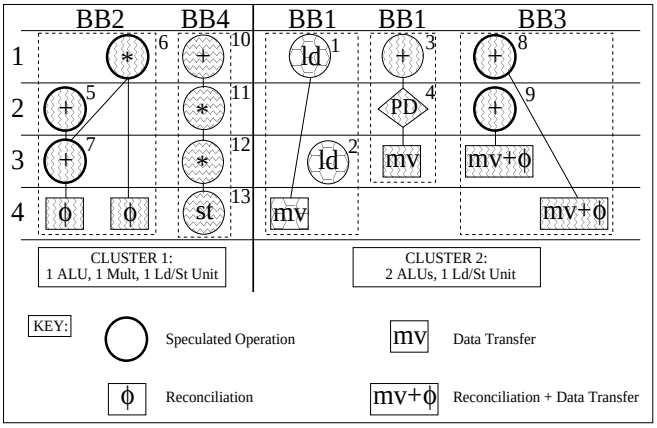


Figure 6.4: Software pipelined schedule for example predicated loop body (with speculation).

segment, possible to generate using our technique, is shown in Figure 6.4.[1] Operations that are speculated (operations 5, 6, 7, 8 and 9) are shown with a thick border. An initiation interval of 4 time steps has been achieved for the target 2-cluster machine (see resource description of machine in Figure 6.4).

Note that an initiation interval of 4 *could not* have been achieved (using the same number of pipe stages) without speculation, because the operations shown with a thick border (i.e. the speculated operations) would have to be

[1]Note that this example shows all the various types of data transfer operations that are possible to be generated by our technique. Namely, the two operations marked mv illustrate the explicit data transfer operations required to transfer the predicate values and the loaded values from cluster 2 to cluster 1. The two operations marked $mv + \phi$ show the collapsed data transfer and reconciliation functions that are generated by the SSA-PS transformation. In this example, they perform the dual functions of data transfer (from cluster 2 to cluster 1) and reconciliation of the speculated addition operations of basic block 3. Finally, the two operations marked ϕ perform the reconciliation of the speculated operations of basic block 2 in cluster 1.

delayed till after the predicate values are computed, thus lengthening the initiation interval of the schedule. This is illustrated in Figure 6.5, which shows the same software pipelined predicated loop body, now scheduled without speculation. Note that its initiation interval has increased to 5 time steps.

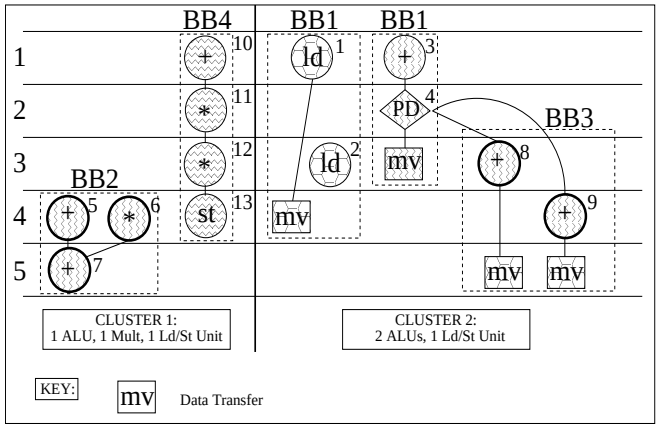


Figure 6.5: Software pipelined schedule for example predicated loop body (without speculation).

In fact, for this example, a three pipe stage schedule with an initiation interval of 4 time steps cannot be found by using software pipelining alone, no matter what retiming function is used. Figure 6.6 shows the same predicated loop body, software pipelined for three pipe stages, using a different retiming function. The corresponding software pipelined schedule (for the same three pipe stages), without speculation, is shown in Figure 6.7 – as in the previous case, the initiation interval is 5 time steps. In other words, if software pipelining alone were used, then the same initiation interval (of 4 time steps) can be achieved only by increasing the number of pipe stages.

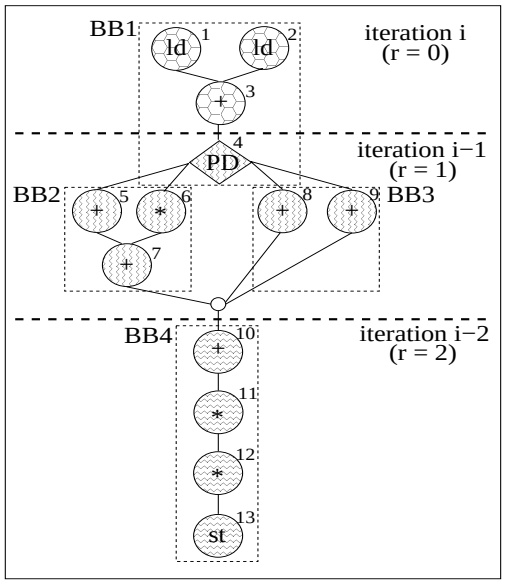


Figure 6.6: Example predicated loop body with different retiming function.

In summary, our resource aware ILP extraction scheme allows us to combine the strengths of speculation and software pipelining. This synergistic combination exposes additional optimization opportunities to the software pipelining algorithm thus enabling performance enhancement at possibly lower cost. We describe our proposed optimization flow in the next section.

6.3 Optimization Flow – Predicated Switching incorporating Software Pipelining

Figure 6.8 shows the complete iterative optimization flow of Predicated Switching incorporating software pipelining. Obviously, in this flow the input CDFG must correspond to a predicated loop body. As can be seen, the op-

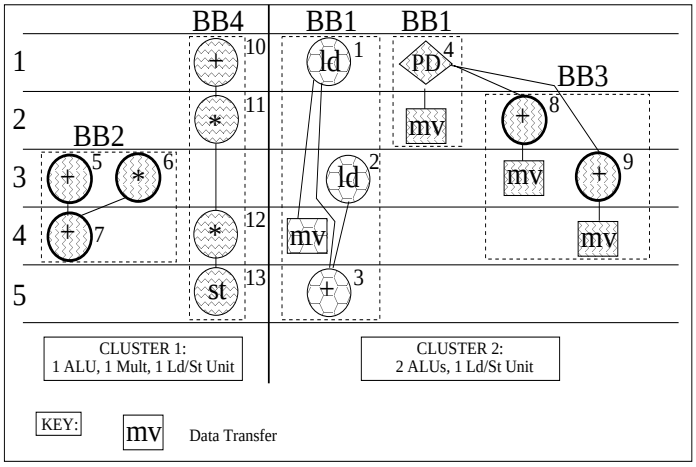


Figure 6.7: Software pipelined schedule for example predicated loop body with different retiming function (without speculation).

timization flow described in Section 5.4, is now augmented with a modulo scheduling phase replacing the list scheduler. The flow shown in Figure 6.8 attempts to find the minimum initiation interval for the input CDFG (on the given target machine) subject to a constraint on the maximum number of pipe stages. For all our experiments, we allowed a maximum of five pipe stages in the software pipelined schedule, so as to minimize the negative effects of using an excessive number of pipe stages (see Sections 6.1 and 6.2). Different retiming functions targeting different initiation intervals are consecutively attempted until a termination condition is satisfied (see Section 5.4). The best resulting solution has the least number of pipe stages, for the least initiation interval found by the scheduler.

Placing the modulo scheduler late in the optimization flow enables it

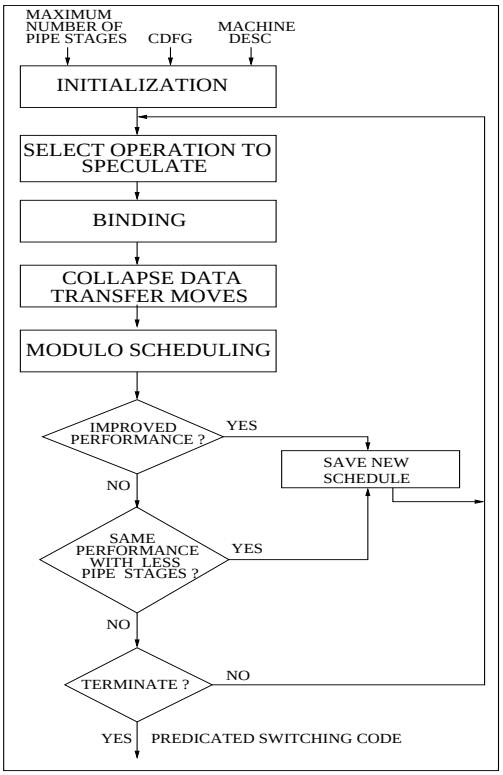


Figure 6.8: Overview of optimization flow incorporating software pipelining.

to take advantage of two important earlier phases, namely: (1) the resource aware speculation phase, which extracts additional ILP that can be profitably exploited by the modulo scheduler; and (2) the SSA-PS induced collapsing of reconciliation operations with data transfer operations, since these can dilute the overhead resulting from clustering. In other words, the additional optimization opportunities unlocked by these earlier two phases, is now made available to the modulo scheduler for further performance enhancement.

An important difference between the selection phase in this optimiza-

tion flow and the flow described in the previous chapter is that the load profiles are calculated on a "folded" basis. The rationale behind using "folded" load profiles is to account for the increased resource utilization by the kernels when they are software pipelined. Specifically, depending on the initiation interval targeted at a given phase of the iteration process, the original load profile is used to calculate the "folded" load profile, which is then used to select the best candidate for speculation. If the target profile latency is pr and an initiation interval of ii is targeted, then the "folded" load at time step t is given by:

$$Folded\_Load(t) = \sum_{\forall i \in [1,pr] \text{ s.t. } t=(i) mod (ii)} (Load(i)), \forall t \in [1, ii]$$

Finally, a modulo scheduling algorithm software pipelines and schedules the resulting DFG. A simple modulo scheduler that implements a two level priority function ranking operations first by lower ALAP and next by lower mobility was used. The algorithm attempts to reduce the number of pipe stages by scheduling as many operations as possible in pipe stage 1, followed by the next pipe stage 2, and so on. It is important to note that any software pipelining algorithm, e.g. [1], can be used in our optimization framework, to take advantage of the increased opportunities for performance enhancement that are made possible through resource-aware speculation. The optimization flow is repeated for target initiation intervals that are factors of the target profile latency, starting from the minimum initiation interval achievable on the target machine. We search over a large number of different such profile latencies, starting from the ASAP latency of the original CDFG.

As before, if the initiation interval is improved with respect to the previous best result, then the corresponding schedule is saved. However, we also save schedules when the same initiation interval is attained using a smaller number of pipe stages, as these solutions are superior to those with a larger number of pipe stages (see Sections 6.1 and 6.2).

6.4 Experimental Results

In this section, we provide experimental evidence of the performance enhancements that our proposed ILP extraction framework, predicated switching incorporating software pipelining, can potentially deliver. The subset of kernels used in Section 5.5 that are loop bodies were considered in these experiments. Details of these kernels are provided in Table 6.1

For the experiments in this chapter, we use slightly wider issue machines since software pipelined loops have increased resource demands. Furthermore, we only consider *multicluster* machines, since single cluster (centralized) machines do not provide enough resources to usefully exploit the increased ILP that software pipelining provides. Accordingly, four clustered machine configurations with resource configurations of |1|1|2|2| and |2|2|2|2| per cluster (using the same notation as in Section 5.5) were used for the experiments in this chapter.

Tables 6.2 and 6.3 show the performance results for all the kernels over all the EPIC machine configurations considered, as well as the number of pipe stages used by each technique. The format of the tables is the same as

ID #	Kernel	# Inst	Critical Path (# steps)	I-ILP
1	Jquant2	82	12	7.45
2	Collision	33	10	3.67
3	Store	39	14	3.55
4	Deblockvert	79	28	3.12
5	Deblockhoriz	69	27	3
6	Viterbi	44	17	2.93
7	Vdkcthresh8	12	7	2.4
8	Lqsolve	15	8	2.14
9	Blockdequant	31	22	2
10	Shortterm	16	12	1.7
11	Blockquant	13	10	1.63
12	Findcbp	14	10	1.56

Table 6.1: Kernel Characteristics.

explained in Section 5.5, augmented with additional information about pipe stages used.

6.4.1 Overall Result Statistics

Comparing the results of software pipelining with and without speculation would consistently reveal that better results can be achieved by combining the two techniques. Instead, we chose to show the results of software pipelining combined with the three speculation techniques discussed before – ours and the baselines. Thus, this set of experiments not only illustrates the effects of combining speculation with software pipelining (see number of speculated operations in Tables 6.2 and 6.3), but also re-emphasizes the need to perform such speculation on a resource aware basis.

	2 Clusters |1|1|2|2|					3 Clusters |1|1|2|2|				
ID #	Kernel Initiation Interval (Pipe Stages)	% Impr. of PS over		# Spec Ops/ Mergings		Kernel Initiation Interval (Pipe Stages)	% Impr. of PS over		# Spec Ops/ Mergings	
	CS/MS/PS	CS	MS	MS	PS	CS/MS/PS	CS	MS	MS	PS
1	21/37/21 (3/1/3)	0.0	43.2	26/15	1/0	17/29/15 (3/1/3)	11.8	48.3	25/20	1/1
2	8/8/8 (3/3/3)	0.0	0.0	2/0	0/0	6/6/6 (4/4/3)	0.0	0.0	2/0	1/0
3	14/14/14 (4/3/3)	0.0	0.0	10/7	2/1	10/10/9 (4/4/5)	10.0	10.0	10/8	4/4
4	40/35/35 (1/1/1)	12.5	0.0	11/12	12/13	40/35/35 (1/1/1)	12.5	0.0	11/12	12/13
5	22/21/21 (4/3/4)	4.6	0.0	13/8	7/5	20/20/16 (3/4/4)	20.0	20.0	14/15	6/5
6	17/16/16 (2/3/2)	5.9	0.0	6/4	2/0	10/10/10 (4/4/4)	0.0	0.0	6/6	0/0
7	3/3/3 (3/3/3)	0.0	0.0	0/0	0/0	3/3/3 (4/4/4)	0.0	0.0	0/0	0/0
8	6/6/6 (3/3/2)	0.0	0.0	6/2	2/1	4/4/4 (3/3/3)	0.0	0.0	8/2	0/0
9	8/8/7 (4/4/5)	12.5	12.5	4/3	3/2	7/8/7 (5/5/5)	0.0	12.5	4/5	3/2
10	6/6/6 (4/4/4)	0.0	0.0	4/2	0/0	5/5/5 (4/4/4)	0.0	0.0	4/3	0/0
11	4/4/4 (4/5/4)	0.0	0.0	2/1	0/0	3/4/3 (5/4/5)	0.0	25.0	2/2	0/0
12	7/7/7 (3/3/2)	0.0	0.0	6/0	1/0	5/5/5 (4/4/3)	0.0	0.0	6/0	1/0

Table 6.2: Performance: Predicated Switching incorporating Software Pipelining (1/2).

The overall statistics for the 48 experiments summarized in Tables 6.2 and 6.3 (24 experiments reported in Table 6.2 and 24 experiments reported

in Table 6.3) are as follows. Predicated switching improved performance with respect to CS (conservative speculation) in 35.4% of the experiments, and improved performance with respect to MS (maximum speculation) in 37.5% of the experiments. Predicated switching code improved performance with respect to *both* baseline solutions (i.e., MS *and* CS code) in 27.1% of the experiments and with respect to *at least one* of the baseline solutions in 45.8% of the experiments. The average improvement in the initiation interval of predicated switching code over CS was 5.2% and over MS was 8.1%. The maximum performance improvement of predicated switching over CS was 28.6%, and that over MS was 50%. When delivering the same performance, predicated switching reduced the number of pipe stages w.r.t. CS in 35.5% of the cases and w.r.t. MS in 31% of the cases by upto a maximum of 33%. We discuss these results in more detail below.

6.4.2 Analysis of Results

Analysis of the results obtained for the 2-cluster machines show that predicated switching improved performance with respect to *both* baseline approaches in 12.5% of the cases and improved performance with respect to at least one of the baselines in 37.5% of the cases. For the 3-cluster machines, predicated switching improved performance with respect to *both* baseline approaches in 41.7% of the cases, and with respect to at least one of the baselines in 54.2% of the cases. Moreover, in 48% of all the experiments, predicated switching was also able to merge SSA-PS related reconciliation functions with

load distributing data transfer operations, indicating that the SSA-PS transformation was highly effective.

	2 Clusters					3 Clusters				
ID #	Kernel Initiation Interval (Pipe Stages)	% Impr. of PS over		# Spec Ops/ Mergings		Kernel Initiation Interval (Pipe Stages)	% Impr. of PS over		# Spec Ops/ Mergings	
	CS/MS/PS	CS	MS	MS	PS	CS/MS/PS	CS	MS	MS	PS
1	11/22/11 (4/1/3)	0.0	50.0	22/8	1/1	10/15/8 (3/4/5)	20.0	46.7	27/18	1/0
2	5/5/5 (4/4/4)	0.0	0.0	2/0	1/0	4/4/4 (4/5/4)	0.0	0.0	2/0	0/0
3	7/8/7 (5/5/4)	0.0	12.5	9/7	3/3	7/7/5 (4/4/5)	28.6	28.6	10/8	1/0
4	35/31/30 (1/1/1)	6.5	3.3	12/12	14/13	35/31/30 (1/1/1)	6.5	3.3	12/12	14/13
5	15/13/12 (4/4/4)	20.0	7.7	14/8	6/6	15/13/12 (4/4/4)	20.0	7.7	14/8	6/6
6	8/8/8 (4/4/4)	0.0	0.0	6/4	0/0	8/8/7 (4/3/4)	12.5	12.5	6/7	1/1
7	3/3/3 (4/4/4)	0.0	0.0	0/0	0/0	2/2/2 (5/5/5)	0.0	0.0	0/0	0/0
8	3/3/3 (4/4/3)	0.0	0.0	8/2	7/2	3/3/3 (4/4/3)	0.0	0.0	8/2	3/0
9	7/7/7 (5/5/5)	0.0	0.0	4/1	3/2	7/7/7 (5/5/5)	0.0	0.0	4/1	3/3
10	5/5/5 (5/4/4)	0.0	0.0	4/2	3/2	5/5/4 (4/4/5)	20.0	20.0	4/3	1/1
11	3/3/3 (5/5/5)	0.0	0.0	2/2	0/0	3/3/3 (4/4/4)	0.0	0.0	2/2	1/1
12	4/4/4 (4/3/3)	0.0	0.0	6/0	4/0	4/4/3 (4/5/5)	25.0	25.0	6/0	1/0

Table 6.3: Performance: Predicated Switching incorporating Software Pipelining (2/2).

For both the 2-cluster machines, we notice that in most cases, predicated switching improved performance with respect to at least one of the comparison techniques whenever it decided to speculate at least one operation. Note that the correlation between actual initial ILP of the kernel and its performance is uninformative since software pipelining increases a kernel's ILP. Moreover, in most of the cases when speculation was performed, collapsing of data transfers with reconciliation operations was also possible. Over the 2-cluster machines, whenever predicated switching achieved the same performance as CS, it did so using a smaller number of pipe stages in 44.4% of the cases. The reduction in number of pipe stages was as much as 33% achieved for Kernel 12 on the |1|1|2|2| machine (see left sub-table of Table 6.2). Also, whenever predicated switching achieved the same performance as MS, it used smaller number of pipe stages in 27.8% of the experiments on the 2-cluster machines. These statistics show that even when predicated switching achieves the same performance (i.e., does not improve the initiation interval of a kernel), it is capable of reducing the cost overhead of the solutions it generates.

Both the 3-cluster machines show improvements that are significantly larger than those obtained for the configurations using a smaller number of clusters, strongly suggesting that predicated switching incorporating software pipelining is more effective with increase in the degree of clustering. For both these 3-cluster machines, predicated switching usually produced better schedules whenever it decided to speculate operations. As before, a good percentage of reconciliation operations are merged with data transfers. Specifically, for

the 3-cluster |1|1|2|2| machine (right sub-table of Table 6.2), Kernels 1, 3 and 5 improved over both CS and MS by speculating 1, 4 and 6 operations respectively, and collapsing 1, 4 and 5 data transfers respectively. Similarly, for the 3-cluster |2|2|2|2| configuration (right sub-table of Table 6.3), Kernels 1, 3, 4, 5, 6, 10 and 12 improved over both the comparison techniques by speculating 1, 1, 14, 6, 1, 1 and 1 operations respectively, and collapsing a significant percentage of data transfers. Note that in a few cases more collapses are performed compared to the number of operations actually speculated (example Kernel 4 on the 3-cluster |1|1|2|2| machine). This is because during collapsing certain operations may need to be duplicated in order to maintain program correctness. Furthermore, whenever predicated switching generated the same initiation interval w.r.t. CS, it did so using fewer pipe stages in 23.1% of the cases; w.r.t. to MS this statistic was 33.3%.

In summary, predicated switching with software pipelining provides significant performance improvements on wide issue multicluster machines. The synergistic combination of software pipelining with speculation as enabled by our optimization framework results in increased opportunities for performance enhancement at typically lower cost (captured by the number of pipe stages in the generated schedules).

Chapter 7

Conclusions and Future Work

EPIC processors are a very attractive platform for many of today's critical applications, especially those with stringent performance requirements and a significant degree of instruction level parallelism (ILP). In particular, clustered machines can take aggressive advantage of the available ILP, while maintaining high energy-delay efficiency. In this thesis we have presented a resource aware ILP extraction technique specifically geared towards EPIC machines. In particular, we have proposed: (1) a novel compiler transformation, called Static Single Assignment - Predicated Switching (SSA-PS), which provides compilers/code-optimizers with maximum flexibility to selectively speculate individual/critical kernel operations, while simultaneously enabling the use of required data transfers across datapath clusters for an efficient realization of the speculated code; (2) a static speculation algorithm to decide which specific kernel operations should actually be speculated in the final code, so as to maximize execution performance on the target EPIC processor; and (3) an ILP extraction flow incorporating several code generation phases critical to profitable ILP extraction by the compiler. Experimental results presented in Chapters 5 and 6 have shown that predicated switching can deliver performance improvements of up to 50%, when compared to resource-unaware

baseline techniques. Moreover, we have shown that code size and register pressure are not adversely affected by our technique.

Future work in the area of performance enhancing ILP extraction for clustered machines could include the consideration of data speculation explicitly in the optimization process. This type of speculation has the ability to hide the long latency penalties of load instructions, thereby improving performance significantly. Another area of future research might be in incorporating register pressure considerations during the binding phase. Our preliminary work in this area suggests that such considerations may not be very critical in determining final code quality, particularly for homogeneous clustered machines, since the existing binding algorithm already seems to perform an adequate balance of register utilization across clusters, through its distribution of operation load. Still, this topic requires further study in architectures where heterogeneous clusters are considered, since the register file sizes in these architectures may not be proportional to the number of FUs connected to them, leading to explicit estimation of register load being required.

The current optimization process assumes that instruction selection occurs prior to the speculation, binding and scheduling phases. Including the instruction selection phase within the current optimization framework could be investigated as future work.

Bibliography

[1] Cagdas Akturan and Margarida F. Jacome. CALiBeR: A Software Pipelining Algorithm for Clustered Embedded VLIW Processors. In *ICCAD*, pages 112–118, 2001.

[2] Cagdas Akturan and Margarida F. Jacome. RS-FDRA: A Register Sensitive Software Pipelining Algorithm for Embedded VLIW Processors. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 2002.

[3] J. R. Allen, K. Kennedy, C. Portfield, and J. Warren. Conversion of control dependence to data dependence. In *Conference Record of the 10th Annual Symposium on Principles of Programming Languages*, pages 177–189, Austin,Texas, January 1983.

[4] Analog Devices. ADSP-TS001M TigerSHARC DSP product description, 2001. Available online http://www.analog.com/products/descriptions/ADSP-TS001.html.

[5] D. August, D. Connors, S. Mahlke, J. Sias, K. Crozier, B. Cheng, P. Eaton, Q. Olaniran, and W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, July 1998.

[6] Chris Basoglu, Karl Zhao, Keiji Kojima, and Atsuo Kawaguchi. The MAP-CA VLIW-based media processor, March 2000. Equator Technologies Inc. and Hitachi Ltd. Available online http://equator.com.

[7] P. Briggs, T. Harvey, and T. Simpson. Static single assignment construction, 1995.

[8] Mihai Budiu and Seth C. Goldstein. Optimizing Memory Accesses For Spatial Computation. In *International Symposium on Code Generation and Optimization (CGO'03)*, March 2003.

[9] R. Camposano. Path-Based Scheduling for Synthesis. In *IEEE Trans. on Computer-Aided Design Integrated Circuits and Systems*, 1991.

[10] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Predicated static single assignment. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

[11] P. Chang and W. Hwu. Trace selection for compiling large c applications to microcode. In *Proceedings of the 21st International Workshop on Microprogramming and Microarchitecture*, pages 188–198, November 1988.

[12] Pohua P. Chang, Scott A. Mahlke, and Wen mei W. Hwu. Using profile information to assist classic code optimizations. In *Software Practice and Experience*, 1991.

[13] J. D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of 18th Annual ACM Symposium on POPL*, pages 55–66, January 1991.

[14] R.P. Colwell, W.E. Hall, C.S. Joshi, D.B. Papworth, P.K. Rodman, and J.E. Tornes. Architecture and implementation of a VLIW supercomputer. In *Proceedings of Supercomputing '90*, pages 910 – 919, Branford, CT, November 1990.

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

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

[17] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. In *ACM Transactions on Programming Languages and Systems*, pages 451–490, 1991.

[18] Kemal Ebcioğlu, Jason Fritts, Stephen Kosonocky, Michael Gschwind, Eric Altman, Krishnan Kailas, and Terry Bright. An Eight-Issue Tree-VLIW Processor for Dynamic Binary Translation. In *Proceedings of International Conference on Computer Design (ICCD'98)*, pages 488–495. IBM Thomas J. Watson Res. Center, Yorktown Heights, NY, USA, IEEE Press, October 1998.

[19] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, and Giuseppe Desoli. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, British Columbia, Canada, June 2000.

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

[21] J. Fritts. Architecture and Compiler Design Issues in Programmable Media Processors, 2000.

[22] Jason Fritts, Zhao Wu, and Wayne Wolf. Parallel media processors for the billion-transistor era. In *Proceedings of the International Conference on Parallel Processing*, Aizu, Japan, September 1999.

[23] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen mei W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 183–193, San Jose, California, 1994.

[24] G. Goossens, J. Vandewalle, and H. De Man. Loop optimization in register-transfer scheduling for DSP systems. In *Proc. Design Automation Conference*, pages 826–831, 1989.

[25] S. Gupta, N. Saviou, S. Kim, N. D. Dutt, R. K. Gupta, and A. Nicolau. Speculation techniques for high level synthesis of control intensive designs. In *Design Automation Conference*, 2001.

[26] Sumit Gupta, Nick Savoiu, Nikil D. Dutt, Rajesh K. Gupta, and Alexandru Nicolau. Conditional Speculation and its Effects on Performance and Area for High-Level Synthesis. In *International Symposium on System Synthesis*, 2001.

[27] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 2003.

[28] U. Holtmann and R. Ernst. Experiments with low-level speculative computation based on multiple branch prediction. In *IEEE Trans. VLSI Systems*, pages 262–267, 1993.

[29] U. Holtmann and R. Ernst. Combining MBP-speculative computation and loop pipelining in high-level synthesis. In *Proceedings of the ED and TC*, pages 550–556, 1995.

[30] P. Y. Hsu and E. S. Davidson. Highly concurrent scalar processing. In *Proc. 13th International Symposium on Computer Architecture*, pages 386–395, June 1986.

[31] http://www.projectmayo.com/projects/detail.php?projectId=1.

[32] http://www.sun.com/processors/vis/vsdk.html.

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

[34] W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. In *The Journal of Supercomputing*, pages 229–248, January 1993.

[35] Margarida F. Jacome, Gustavo de Veciana, and Satish Pillai. Clustered VLIW architectures with predicated switching. In *Design Automation Conference*, pages 696–701, 2001.

[36] Krishnan Kailas, Kemal Ebcioğlu, and Ashok Agrawala. CARS: A new code generation framework for clustered ILP processors. In *Seventh International Symposium on High Performance Computer Architecture*, Monterrey, Mexico, January 2001.

[37] Vinod Kathail, Mike Schlansker, and Bob Rau. HPL PlayDoh architechture specification: Version 1.0. Technical Report HPL-93-80, Hewlett-Packard Laboratories, February 1993.

[38] T. Kim, J. W. S. Liu, and C. L. Liu. A Scheduling Algorithm for Conditional Resource Sharing. In *Proc. IEEE International Conference on Computer-Aided Design*, pages 84–87, 1991.

[39] Artur Klauser, Todd Austin, Dirk Grunwald, and Brad Calder. Dynamic Hammock Predication for Non-predicated Instruction Set Architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 1998.

[40] Ganesh Lakshminarayana, Anand Raghunathan, and Niraj K. Jha. Incorporating speculative execution into scheduling of control-flow intensive behavioral descriptions. In *Design Automation Conference*, pages 108–113, 1998.

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

[42] Viktor S. Lapinskii, Margarida F. Jacome, and Gustavo de Veciana. High-quality operation binding for clustered VLIW datapaths. In *Design Automation Conference*, pages 702–707, 2001.

[43] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

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

[45] Rainer Leupers. Exploiting Conditional Instructions in Code Generation for Embedded VLIW Processors. In *Design, Automation, and Test in*

Europe, March 1999.

[46] S. Mahlke, R. Hank, James McCormick, D. August, and W. M. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 138–149, June 1995.

[47] Scott A. Mahlke, William Y. Chen, Wen-Mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 27, pages 238–247, New York, NY, 1992. ACM Press.

[48] Scott A. Mahlke, Richard E. Hank, Roger A. Bringmann, John C. Gyllenhaal, David M. Gallagher, and Wen mei Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 217–227, San Jose,California, 1994.

[49] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, Roger A. Bringmann, and Wen mei W. Hwu. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, December 1992.

[50] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[51] A. Nicolau. Percolation Scheduling: A Parallel Compilation Technique. In *Cornell University, Dept. of Computer Science Technical Report*, 1985.

[52] Alexandru Nicolau and Steven Novak. Trailblazing: A Hierarchical Approach to Percolation Scheduling. In *Proceedings of the 1993 International Conference on Parallel Processing*, 1993.

[53] M. Palkovic, M. Miranda, F. Catthoor, and D. Verkest. High-level condition expression transformations for design exploration. In *System Design Automation - Fundamentals, Principles, Methods, Examples*, pages 56–64. Kluwer Academic Publishers, 2001.

[54] 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 Computer-Aided Design*, pages 356–368, March 1988.

[55] P. G. Paulin and J. P. Knight. Force-Directed Scheduling in Automatic Data Path Synthesis. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 195–202, Miami Beach, FL, June 1987. IEEE Computer Society Press.

[56] Satish Pillai and Margarida Jacome. Symbolic Binding for Clustered VLIW ASIPs. In *International Conference on Computer Design*, 2000.

[57] Satish Pillai and Margarida F. Jacome. Compiler-Directed ILP Extraction for Clustered VLIW/EPIC machines. In *Embedded Software for SOC*. Kluwer Academic Publishers, August 2003.

[58] Satish Pillai and Margarida F. Jacome. Compiler-Directed ILP Extraction for Clustered VLIW/EPIC machines: Predication, Speculation and Modulo Scheduling. In *Proceedings of IEEE/ACM Design Automation and Test in Europe (DATE)*, 2003.

[59] B. Rau, D. Yen, W. Yen, and R. Towle. The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs. In *IEEE Computer*, pages 12–35. January 1989.

[60] Scott Rixner, William J. Dally, Brucek Khailany, Peter Mattson, Ujival J. Kapasi, and John D. Owens. Register Organization for Media Processing. In *Proc. of 26th International Symposium on High-Performance Computer Architecture*, May 1999.

[61] Texas Instruments. TMS320C6000 CPU and instruction set reference guide, October 2000. Literature Number: SPRU226.

[62] C. J. Tseng, R. W. Wei, S. G. Rothweiler, M. Tong, and A. K. Bose. Bridge: A Versatile Behavioral Synthesis System. In *Proc. 25rd ACM/IEEE Design Automation Conference*, pages 415–420, 1988.

[63] Gary Scott Tyson. The effects of predicated execution on branch prediction. In *Proceedings of the 27th Annual International Symposium on*

Microarchitecture, pages 196–206, San Jose,California, 1994.

[64] K. Wakabayashi and H. Tanaka. Global Scheduling Independent of Control Dependencies Based on Condition Vectors. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pages 112–115, 1992.

[65] S. W. White and S. Dhawan. POWER2: Next generation of the RISC System/6000 family. *IBM Journal of Research and Development*, 38(5):493–502, September 1994.

[66] P. F. Yeung and D. J. Rees. Resources Restricted Global Scheduling. In *VLSI 1991*, pages 287–296, 1991.

Vita

Satish Pillai was born in Pune, India on 10th June 1975, the son of Soman Pillai and Vijaya Pillai. He received the Bachelor of Engineering degree in Electrical and Electronics Engineering from P.S.G. College of Technology in 1997. In the same year, he started his graduate studies in Electrical and Computer Engineering at the University of Texas at Austin. He received the Master of Science degree in Electrical and Computer Engineering in 1999 and began his doctoral research in the same year.

Permanent address: 2106 San Gabriel #219
Austin, Texas 78705

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.