
Contents

1	Branch Predication	3
	<i>David August</i> Princeton University	
1.1	Introduction	3
1.1.1	Overcoming Branch Problems With Predication	4
1.2	A Generalized Predication Model	6
1.3	Compilation For Predicated Execution	9
1.3.1	If-Conversion	10
1.3.2	The Effect of Predication on ILP	11
1.3.3	Predicate Optimization and Analysis	11
1.3.4	The Predicated Intermediate Representation	13
1.4	Architectures with Predication	14
1.4.1	Hewlett-Packard Laboratories PlayDoh	14
1.4.2	Cydrome Cydra 5	15
1.4.3	ARM	16
1.4.4	Texas Instruments C6X	16
1.4.5	Systems with limited predicated execution support	17
1.5	In Depth: Predication in Intel IA-64	17
1.5.1	Predication in the Itanium 2 processor	18
1.6	Predication in Hardware Design	20
1.7	The Future of Predication	23
	References	25



Chapter 1

Branch Predication

David August

Princeton University

1.1	Introduction	3
1.2	A Generalized Predication Model	6
1.3	Compilation For Predicated Execution	9
1.4	Architectures with Predication	14
1.5	In Depth: Predication in Intel IA-64	17
1.6	Predication in Hardware Design	20
1.7	The Future of Predication	22

1.1 Introduction

The performance of instruction-level parallel (ILP) processors depends on the ability of the compiler and hardware to find a large number of independent instructions. Studies have shown that current wide-issue ILP processors have difficulty sustaining more than two instructions per cycle for non-numeric programs [1, 2, 3]. These low speedups are a function of a number of difficult challenges faced in extracting and efficiently executing independent instructions.

The extraction of ILP can be done by the compiler, the hardware, or both. The compiler and hardware have individual strengths and the task of extracting ILP should be divided accordingly. For example, the compiler has the ability to perform detailed analysis on large portions of the program, while the hardware has access to specific information which can only be determined at run time. While extracting ILP is necessary to achieve large speedups in ILP processors, the architecture must be able to execute the extracted ILP in an efficient manner. Performance gained through the extraction of ILP is often lost or diminished by stalls in the hardware caused by the long latency of operations in certain exceptional conditions. For example, a load from memory may stall the processor while the data is fetched from the relatively slow main memory. Hardware techniques which minimize these stalls, such as cache in case of memory, are essential for realizing the full potential of ILP. One of the major challenges to effectively extracting and efficiently executing ILP code is overcoming the limitations imposed by branch control flow.

At run time, the branch control flow introduces uncertainty of outcome and

non-sequentiality in instruction layout, which limit the effectiveness of instruction fetch mechanisms. Sophisticated branch prediction techniques, speculative instruction execution, advanced fetch hardware, and other techniques described in this book are necessary today to reduce instruction starvation in high-performance processor cores [4, 5, 6, 7].

For non-numeric benchmarks, researchers report that approximately 20% to 30% of the dynamic instructions are branches, an average of one branch for each three to five instructions. As the number of instructions executed concurrently grows, the sustained rate of branches executed per cycle must grow proportionally to avoid becoming the bottleneck. Handling multiple branches per cycle requires additional pipeline complexity and cost, which includes multi-ported branch prediction structures. In high issue rate processors, it is much easier to duplicate arithmetic functional units than to predict and execute multiple branches per cycle. Therefore, for reasons of cost, ILP processors will likely have limited branch handling capabilities which may limit performance in non-numeric applications. In those processors which do support multiple branches per cycle, the prediction mechanisms are subject to diminishing prediction accuracy created by the additional outcomes of execution. Even branch prediction mechanisms with good accuracy on a single branch per cycle, often have undesirable scaling characteristics.

1.1.1 Overcoming Branch Problems With Predication

Many forms of speculation have been proposed to alleviate the problems of branching control discussed above. A radically different approach taken in Explicitly Parallel Instruction Computing (EPIC) architectures allows the compiler to simply eliminate a significant amount of branch control flow presented to the processor. This approach, called *predication* [8, 9], is a form of compiler-controlled speculation in which branches are removed in favor of fetching multiple paths of control. The speculative trade-off the compiler makes involves weighing the performance cost of potential branch mispredictions with the performance cost associated with fetching additional instructions.

Specifically, predication is the conditional execution of instructions based on the value of a Boolean source operand, referred to as the predicate. If the value of the predicate is true (a logical **1**), a predicated instruction is allowed to execute normally; otherwise (a logical **0**), the predicated instruction is nullified, preventing it from modifying the processor state.

Figure 1.1 contains a simple example to illustrate the concept of predication. Figure 1.1(a), shows two if-then-else construct, called *hammocks*, arranged sequentially within the code. The outcome of each branch is determined by the evaluation of the independent branch conditions *Cond1* and *Cond2*, which may be $r3 < 10$ or any other register value comparison. Depending on the outcome of the first branch register *r1* is either incremented or decremented. Register *r2* is incremented only when *Cond2* is false. In order to respect the

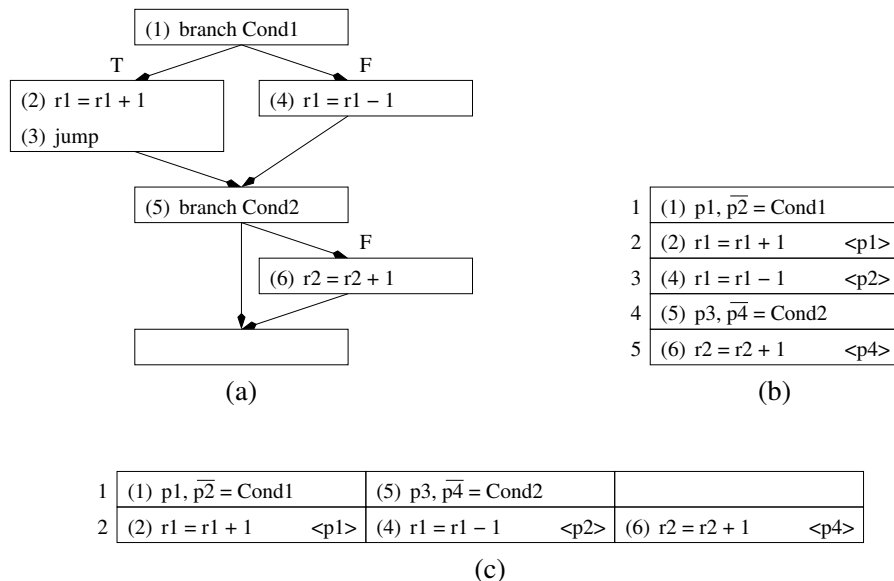


FIGURE 1.1: A simple code segment with and without predication.

branch control dependences, the target of the first branch must be selected before the second branch can execute.

The basic compiler transformation to exploit predicated execution is known as *if-conversion* [10, 11]. If-conversion replaces conditional branches in the code with comparison instructions that define one or more predicates. Instructions control dependent on the branch are then converted to predicated instructions, utilizing the appropriate predicate value. In this manner, control dependences are converted to data dependences.

Figure 1.1b shows the code segment after if-conversion, and Figure 1.1c shows the code after scheduling. Here the two branch conditions are now part of two predicate defines. Four predicates are defined to be true or false depending on the evaluation of the conditions. For example, if *Cond1* is satisfied, P_1 is set to true and P_2 is set to false. In the next cycle, instructions 2, 4, and 6 are executed based on the values stored in the predicate registers. Notice that these instructions do not need to respect positional dependence with any branches. They are now free to be scheduled simultaneously as they are here. As a result, real ILP (ILP discounting the nullified instructions) can be increased over this original code segment even assuming perfect branch prediction. Another side benefit is the potential reduction in instruction count. Instruction 3 is no longer necessary since within this block of code execution conditions are no longer determined solely by fetch pattern. Further, fetch hardware now has a single very simple fetch pattern to perform instead of four patterns of which only one is correct for every execution of this code segment.

Beyond these scheduling benefits, the removal of branches with predication improves performance by eliminating any branch misprediction penalties. In particular, the removal of frequently mispredicted branches can yield large performance gains [12, 13, 14].

The remainder of this chapter covers predication from several perspectives. The next section presents a generalized model of predicated architectures. Using this model as a point of reference, Section 1.3 details issues in the compilation for predication. Following this, Section 1.4 surveys several research and commercial predicated architectures, and Section 1.5 provides a detailed analysis of predication in action on the Itanium 2 processor. Section 1.6 outlines the challenges predication presents in future hardware design. Finally, the chapter concludes with a look toward predication's role in the future.

1.2 A Generalized Predication Model

This section presents the *IMPACT EPIC* architecture model which serves as a generalized model of architectures with predicated execution support and serves as a basis for the discussion of predication in the remainder of the chapter [15]. The *IMPACT EPIC* architecture model, a statically scheduled, in-order issue, EPIC architectural originated as a generalization of the Cydra 5 and the HPL PlayDoh architectures [9],[16] described later.

Any architecture supporting predicated execution must be able to conditionally nullify the side effects of selected instructions based on the value of its predicate source register. Additionally, the architecture must support efficient computation of predicate values. To do this, the *IMPACT EPIC* model of predicated execution contains N independently addressable single-bit predicate registers. Predicate register 0 is defined always to hold the value **1**. All instructions in the model take a guard predicate source register. The architectural also includes a set of predicate defining instructions. These predicate defining instructions are classified broadly as predicate comparison instructions, predicate clear/set instructions, and predicate save/restore instructions.

Predicate register file. As previously mentioned, an N by 1-bit register file to hold predicates is added to the baseline architecture. The choice of introducing a new register file to hold predicate values rather than using the existing general purpose register file was made for several reasons. First, it is inefficient to use a single general register to hold each one bit predicate. Second, register file porting is a significant problem for wide-issue processors. By keeping predicates in a separate file, additional port demands are not added to the general purpose register file. Within the architecture, the predicate register file behaves no differently than a conventional register file. For exam-

P_{guard}	C	UT	UF	OT	OF	AT	AF	CT	CF	$\vee T$	$\vee F$	$\wedge T$	$\wedge F$
0	0	0	0	-	-	-	-	-	-	-	1	0	0
0	1	0	0	-	-	-	-	-	-	1	-	0	0
1	0	0	1	-	1	0	-	0	1	1	1	0	-
1	1	1	0	1	-	-	0	1	0	1	1	-	0

FIGURE 1.2: IMPACT EPIC predicate deposit types.

ple, the contents of the predicate register file must be saved during a context switch. Furthermore, the predicate file is partitioned into caller and callee save sections based on the chosen calling convention.

Predicate comparison instructions. The most common way to set predicate register values is with a new set of predicate comparison instructions. Predicate comparison instructions compute predicate values using semantics similar to those for conventional comparison instructions. There is one predicate comparison instruction for each integer, unsigned, float, and double comparison instruction in an unpredicated ISA. Unlike traditional comparison instructions, the predicate comparison instructions have up to two destination operands and these destination operands refer to registers in the predicate register file. The predicate comparison instruction format is as shown here.

$$\left(P_{dest_1} \xleftarrow{type_1}, P_{dest_2} \xleftarrow{type_2} \right) (src_0 [cmp] src_1) \langle P_{guard} \rangle$$

This computes the condition $C = src_0 [cmp] src_1$ and optionally assigns a value to two destination predicates, P_{dest_1} and P_{dest_2} , according to the predicate types ($type_1$ and $type_2$, respectively), C , and the guarding predicate P_{guard} , as shown in Table 1.2. (In the table, a “-” indicates that no value is deposited.) The IMPACT EPIC Architecture defines six deposit types which specify the manner in which the result of a condition computation and the guard predicate are deposited into a destination predicate register. The (U)nconditional, (O)r, (A)nd, and (C)onditional types are as defined in the HP Labs PlayDoh Specification [16]. The and- (A), or- (O), conjunctive- (\wedge) and disjunctive- (\vee) type predicate deposits are termed parallel compares since multiple definitions of a single one of these types can commit simultaneously to the same predicate register. While simple if-conversion generates only unconditional and or-type defines [11], the other types are useful in optimization of predicate define networks, so an effective predicate analysis system should fully support their general use.

Unconditional destination predicate registers are always defined, regardless of the value of P_{guard} and the result of the comparison. If the value of P_{guard} is **1**, the result of the comparison is placed in the predicate register (or its complement for \overline{U}). Otherwise, a **0** is written to the predicate register. Unconditional predicates are utilized for blocks that are executed based on a single condition, i.e., they have a single control dependence.

The OR-type predicates are useful when execution of a block can be enabled by multiple conditions, such as logical AND (`&&`) and OR (`||`) constructs in C. OR-type destination predicate registers are set if P_{guard} is **1** and the result of the comparison is **1** (**0** for \overline{OR}); otherwise, the destination predicate register is unchanged. Note that OR-type predicates must be explicitly initialized to **0** before they are defined and used. However, after they are initialized multiple OR-type predicate defines may be issued simultaneously and in any order on the same predicate register. This is true since the OR-type predicate either writes a **1** or leaves the register unchanged which allows implementation as a wired logical OR condition. This property can be utilized to compute an execution condition with zero dependence height using multiple predicate define instructions.

The AND-type predicates are analogous to the OR-type predicates. AND-type destination predicate registers are cleared if P_{guard} is **1** and the result of the comparison is **0** (**1** for \overline{AND}); otherwise, the destination predicate register is unchanged. The AND-type predicate is particularly useful for transformations such as control height reduction [17],[18].

The conditional type predicates have semantics similar to regular predicated instructions, such as adds. If the value of P_{guard} is **1**, the result of the comparison is placed in the destination predicate register (or its complement for \overline{C}). Otherwise, no actions are taken. Under certain circumstances, a conditional predicate may be used in place of an OR-type predicate to eliminate the need for an initialization instruction. However, the parallel issue semantics of the OR-type predicates are lost with conditional predicates.

Two predicate types primarily facilitate generating efficient code using Boolean minimization techniques by allowing predicate defines to behave functionally as logical-and and logical-or gates [19]. These are referred to as *disjunctive-type* ($\vee T$ or $\vee F$) and *conjunctive-type* ($\wedge T$ or $\wedge F$). Table 1.2 (right-hand portion) shows the deposit rules for the new predicate types. The $\wedge T$ -type define clears the destination predicate to **0** if either the source predicate is FALSE or the comparison result is FALSE. Otherwise, the destination is left unchanged. Note that this behavior differs from that of the and-type predicate define, in that the and-type define leaves the destination unaltered when the source predicate evaluates to FALSE. The conjunctive-type thus enables a code generator to easily and efficiently form the logical conjunction of an arbitrary set of conditions and predicates.

The disjunctive-type behavior is analogous to that of the conjunctive-type. With the $\vee T$ -type define, the destination predicate is set to **1** if either the source predicate is TRUE or the comparison result is TRUE (FALSE for $\vee F$). The disjunctive-type is thus used to compute the disjunction of an arbitrary set of predicates and compare conditions into a single predicate.

Predicate clearing and setting. The stylized use of OR-type and AND-type predicates described previously requires that the predicates be precleared and preset, respectively. While preclearing and presetting can be performed by unconditional predicate comparison instructions, a set of instructions are

defined to make this process more efficient.

First, instructions to clear and set groups of registers using a mask are provided. These instructions are aptly called *pclr* and *pset*. These instructions set a contiguous group of predicate registers to zero or one using a mask. Thus, any combination of the predicates can be cleared or set using these instructions. For architectures with many predicate registers and limited instruction encoding, the mask may refer only to a subset of the predicate register file.

Second, an integer to predicate collection register move is provided. A predicate collection register is special-purpose integer register which holds the value of a collection of consecutive predicate registers. For example, a machine with 64 predicate registers may have single 64-bit predicate collection register whose value is tied to the state of the entire predicate register file. The predicate collection register move instruction can move the value of the predicate collection register to and from an integer general purpose register. While this value is in a general purpose register, its value can be manipulated in the usual manner.

Predicate saving and restoring. During procedure calls, predicate registers need to be saved and restored according to calling convention. Such saving and restoring can be performed using moves to and from the predicate collection register. A similar operation can be performed by the operating system during context switches.

Should the number of predicate registers needed by the compiler exceed the number of architectural predicate registers, special predicate spill and fill operations should be used. These operations, called *pspill* and *pfill*, allow an individual predicate register to be loaded from and stored to stack memory. In this manner, the compiler has the freedom to handle predicate registers in the same way as the conventional register types.

1.3 Compilation For Predicated Execution

Since predication is completely compiler inserted and managed, compilers for predicated architectures are necessarily more complex. A compiler with predication support must simultaneously handle codes with branches and predicated instructions since programs will contain both. To be effective, optimizations and analyses must be able to handle predicate and branch control of instructions with equal strength. As we will see in this section, the complexity added by predication is a challenge to compiler writers, yet it is not without benefits.

1.3.1 If-Conversion

At the most basic level, compilers support predicated execution by performing if-conversion for some set of branches at some point in the compilation process. The decision of what branches to if-convert and when in the compilation process to if-convert them can have large effects on overall code performance.

As mentioned earlier, predication is a trade-off between the benefits of removing a branch and the cost of processing multiple paths. Unfortunately, these factors are hard to estimate at the time during compilation that the decision is made. For example, the compiler must estimate the misprediction rate for each branch under consideration. Further, the compiler must also estimate the scheduling and optimization effects if-conversion will have on subsequent compiler optimizations. Profile information may help with characterizing the misprediction rate, but estimating the effect removing the branch will have on subsequent optimizations is a real problem which has implications on when in the compilation process if-conversion is best applied [20].

Previous work has demonstrated the value of introducing predication in an early compilation phase to enhance later optimization; in particular the *hyperblock* compilation framework has been shown to generate efficient predicated code [21, 22]. The hyperblock compilation framework is able to deliver better performance than late if-conversion because all subsequent optimizations have more freedom to enhance the predicated code. Unfortunately, early if-conversion makes the decision of what to if-convert very difficult. The over-application of if-conversion with respect to a particular machine will result in the over-saturation of processor fetch and potentially execution resources. A delicate balance between control flow and predication must be struck in order to maximize predication's potential. Early if-conversion decisions are made based on the code's characteristics prior to further optimization. Unfortunately, code characteristics change dramatically due to optimizations applied after if-conversion, potentially rendering originally compatible traces incompatible. In cases where estimates of final code characteristics are wrong, the final code may perform worse for having had predication introduced. This problem may be alleviated by *partial reverse if-conversion* techniques which operate at schedule time to balance the amount of control flow and predication present in the generated code. Partial reverse if-conversion replaces some predicated code with branch control flow by the re-introduction of branches [20].

Predicating late in the compilation process, or post-optimization if-conversion, has the advantage of not having to predict the effect of subsequent optimizations. This leads to more consistent, but less impressive performance gains [23]. This is the consequence of not capitalizing on optimization opportunities exposed during if-conversion.

1.3.2 The Effect of Predication on ILP

Typically, if-conversion is the only transformation dealing with predicated codes which considers branch misprediction behavior. Other optimizations operating on predicated codes generally focus on enhancing the level of ILP in the static schedule. Therefore, understanding predication's impact on the ILP in the static schedule provides a foundation for understanding analysis and optimization of predicated codes.

Branches in the code present barriers to efficient extraction and expression of ILP. Branches impose control dependences which by definition sequentialize the execution of surrounding instructions, reducing the number of naturally independent instructions. One type of control dependence occurs when an instruction is located such that the outcome of a branch determines whether or not that instruction will execute. This type of control dependence, from a branch to other instructions (including other branches), is determined by positional relationships within the code. The set of possible layouts for a program is limited by the need to respect these proper control dependence connections. In many cases, respecting these control dependences will limit the creation of schedules with desirable ILP characteristics.

To see how branch control dependences can limit ILP, consider two instructions which both have independent conditions of execution (that is they are both control dependent on separate branches with mutually independent conditions) and which both execute frequently. Instructions 4 and 6 in Figure 1.1 serve as an example of this. Since the condition of execution for each instruction is determined by its position relative to the branches which control it, these instructions must remain in separate parts of the code (Figure 1.1a). Predication eliminates the positional nature of instruction control, and allows instructions to execute in parallel (cycle 2 of Figure 1.1c).

1.3.3 Predicate Optimization and Analysis

Predicate optimizations fall into three categories. These are:

1. optimizations made predicate-aware
2. optimizations enhanced by predication
3. predicate-specific optimizations

Most traditional optimizations fall into the first category. Dead code elimination, loop invariant code removal, common subexpression elimination, register allocation, and others simply need to understand that control is no longer just positional. This is typically accomplished providing by these optimizations with dataflow and control flow analyses which factor predication into their results. This analysis problem is, however, not a simple one.

Several predicate dataflow analysis systems have been proposed to address this issue [24, 25, 26, 27]. The simplest method to deal with predication is

to augment dataflow analyses with the notion of a conditional writer. For example, in liveness analysis predicated instructions simply do not kill their definitions. This is however extremely conservative and leads to suboptimal results. In the case of live variable analysis, the degradation in accuracy dramatically reduces the effectiveness of several optimizations including register allocation [28].

More sophisticated predicate dataflow engines consider the relationship among predicates. Some make approximations for efficiency [24, 26, 27] while others represent the relationship completely for accuracy [25]. This information about the predicates relationships is then used to perform the dataflow analysis itself. To do this, some analysis engines simply create a shadow control flow graph encoding the predicate information upon which traditional dataflow analysis can be performed [27, 26]. Other dataflow analysis systems customize the dataflow analysis itself for predication [24].

In addition to those optimizations which are simply made predicate-aware, there are those which are enhanced by predication. Scheduling and ILP optimizations are encumbered by branches. Generally, these transformations must handle control dependences separately from data dependences. Typically, compilers avoid this complication by limiting these transformations to within a single basic block. Optimization of a code window containing multiple basic blocks requires either transformation of the code, usually with duplication, or application of complex and expensive global optimization techniques. Global basic block techniques often need to make trade-offs between the performance of different paths. The uncertainty of branch outcome forces the compiler to make path selection decisions based on estimates which can often be inaccurate. Predication eliminates these control dependences, in effect turning optimizations designed to handle a single basic block into optimizations which can handle predicated regions formed from a portion of the control flow graph.

Beyond expanding the scope of optimization and scheduling, predicated execution also provides an efficient mechanism by which a compiler can explicitly present the overlapping execution of multiple control paths to the hardware. In this manner, processor performance is increased by the compiler's ability to find ILP across distant multiple program paths. Another, more subtle, benefit of predicated execution is that it facilitates the movement of redundant computation to less critical portions of the code [17].

Predication also allows more freedom to code motion in loops. By controlling the execution of an instruction in a loop with predication, that instruction can be set to execute in some subset of the iterations. This is useful for example for loop invariant code insertion. While loop invariant code insertion may seem like a pessimization, it does in fact produce more compact schedules for nested loops in which the inner loop iterates a few times for each invocation. This is a common occurrence in many codes. This freedom is also useful in removing partial dead code elimination from loops. It also allows for kernel-only software pipelined loops.

Finally, several optimizations only exist to operate on predicated codes. One such technique, called Predicate Decision Logic Optimization (PDLO), operates directly on the predicate defines [29]. PDLO extracts program semantics from the predicate computation logic. Represented as Boolean expressions in a binary decision diagram (BDD), which can also be used for analysis, the predicate network is optimized using standard factorization and minimization techniques. PDLO then reformulates the program decision logic back as more efficient predicate define networks. Combined with aggressive early if-conversion and sophisticated late partial reverse if-conversion, PDLO can be used to optimize predicate defines as well as branches.

Another predicate-specific optimization is called *predicate promotion*. Predicate promotion is a form of compiler-controlled speculation that breaks predicate data dependences. Using predicate promotion to break predicate data dependences is analogous to using compiler-controlled control speculation to break control dependences. In predicate promotion, the predicate used to guard an instruction is *promoted* to a predicate with a weaker condition. That is, the instruction is potentially nullified less often, potentially never, during execution. Since any extra executions are not identified until later, all executions of the instruction must be treated as speculative. Promotion must be performed carefully with potentially excepting instructions as the extra executions may through a spurious exception, an exception not found in the original program. To address this problem, several architectures provide a form of delayed exception handling which is only invoked for the non-spurious exceptions. Predicate promotion is essential to realizing the full potential of predication to extract ILP [27, 19].

1.3.4 The Predicated Intermediate Representation

Even when predicated execution support in the targeted hardware is minimal or nonexistent, a predication may be useful in the compiler's intermediate representation. The predicated intermediate representation is obtained by performing if-conversion early in the compilation process, before optimization. This predicated representation provides a useful and efficient model for compiler optimization and scheduling. The use of predicates to guard instruction execution can reduce or even completely eliminate the need to manage branch control dependences in the compiler, as branches are simply removed through the if-conversion process. Control dependences between branches and other instructions are thus converted into data dependences between predicate computation instructions and the newly predicated instructions. In the predicated intermediate representation, control flow transformations can be performed as traditional data flow optimizations. In the same way, the predicated representation allows scheduling among branches to be performed as a simple reordering of sequential instructions. Removal of control dependences increases scheduling scope and affords new freedom to the scheduler [30]. Before code emission for a machine without full predication support, the predi-

cated instructions introduced must be converted to traditional branch control flow using a process called *reverse if-conversion*.

1.4 Architectures with Predication

1.4.1 Hewlett-Packard Laboratories PlayDoh

The basis for predication in the IMPACT EPIC architecture is the PlayDoh architecture developed at Hewlett-Packard Laboratories [16].

PlayDoh is a parameterized Explicitly Parallel Instruction Computing (EPIC) architecture intended to support public research on ILP architectures and compilation [16]. PlayDoh predicate define instructions generate two Boolean values using a comparison of two source operands and a source predicate. A PlayDoh predicate define instruction has the form:

$$pD_0_type_0, pD_1_type_1 = (src_0 \mathbf{cond} src_1) \langle pSRC \rangle.$$

The instruction is interpreted as follows: pD_0 and pD_1 are the destination predicate registers; $type_0$ and $type_1$ are the predicate types of each destination; $src_0 \mathbf{cond} src_1$ is the comparison, where *cond* can be *equal* ($==$), *not equal* (\neq), *greater than* ($>$), etc.; $pSRC$ is the source predicate register. The value assigned to each destination is dependent on the predicate type. PlayDoh defines three predicate types, *unconditional* (UT or UF), *wired-or* (OT or OF), and *wired-and* (AT or AF). Each type can be in either normal mode or complement mode, as distinguished by the T or F appended to the type specifier (U, O, or A). Complement mode differs from normal mode only in that the condition evaluation is treated in the opposite logical sense.

For each destination predicate register, a predicate define instruction can either deposit a 1, deposit a 0, or leave the contents unchanged. The predicate type specifies a function of the source predicate and the result of the comparison that is applied to derive the resultant predicate. The eight left-hand columns of Table 1.2 show the deposit rules for each of the PlayDoh predicate types in both normal and complement modes, indicated as “T” and “F” in the table, respectively.

The major limitation of the PlayDoh predicate types is that logical operations can only be performed efficiently amongst compare conditions. There is no convenient way to perform arbitrary logical operations on predicate register values. While these operations could be accomplished using the PlayDoh predicate types, they often require either a large number of operations or a long sequential chain of operations, or both.

With traditional approaches to generating predicated code, these limitations are not serious, as there is little need to support logical operations

amongst predicates. The Boolean minimization strategy described in Section 1.3, however, makes extensive use of logical operations on arbitrary sets of both predicates and conditions. In this approach, intermediate predicates are calculated that contain logical subexpressions of the final predicate expressions to facilitate reuse of terms or partial terms. The intermediate predicates are then logically combined with other intermediate predicates or other compare conditions to generate the final predicate values. Without efficient support for these logical combinations, gains of the Boolean minimization approach are diluted or lost.

1.4.2 Cydrome Cydra 5

The Cydra 5 system is a VLIW, multiprocessor system utilizing a directed-dataflow architecture [9, 31]. Each Cydra 5 instruction word contains seven operations, each of which may be individually predicated. An additional source operand added to each operation specifies a predicate located within the predicate register file. The predicate register file is an array of 128 Boolean (one bit) registers.

The content of a predicate register may only be modified by one of three operations: *stuff*, *stuff_bar*, or *brtop*. The *stuff* operation takes as operands a destination predicate register and a Boolean value as well as an input predicate register. The Boolean value is typically produced using a comparison operation. If the input predicate register is one, the destination predicate register is assigned the Boolean value. Otherwise, destination predicate is assigned to 0. The *stuff_bar* operation functions in the same manner, except the destination predicate register is set to the inverse of the Boolean value when the input predicate value is one. The semantics of the unconditional predicates are analogous to the those of the *stuff* and *stuff_bar* operations in the Cydra 5. The *brtop* operation is used for loop control in software pipelined loops and sets the predicate controlling the next iteration by comparing the contents of a loop iteration counter to the loop bound.

Figure 1.3 shows the previous example after if-conversion for the Cydra 5. To set the mutually exclusive predicates for the different execution paths shown in this example requires three instructions. First, a comparison must be performed, followed by a *stuff* to set the predicate register for the true path (predicated on P_1) and a *stuff_bar* to set the predicate register for the false path (predicated on P_2). This results in a minimum dependence distance of 2 from the comparison to the first possible reference of the predicate being set.

In the Cydra 5, predicated execution is integrated into the optimized execution of modulo scheduled inner loops to control the prologue, epilogue, and iteration initiation [32],[33]. Predicated execution in conjunction with rotating register files eliminates almost all code expansion otherwise required for modulo scheduling. Predicated execution also allows loops with conditional branches to be efficiently modulo scheduled.

```

        mov r1,0
        mov r2,0
        ld.i r3,A,0
L1:
        ld.i r4,r3,r2
        gt r6,r4,50
        stuff p1,r6
        stuff_bar p2,r6
        add r5,r5,1 (p2)
        add r6,r6,1 (p1)
        add r1,r1,1
        add r2,r2,4
        blt r1,100,L1

```

FIGURE 1.3: Example of if-then-else predication in the Cydra 5.

1.4.3 ARM

The Advanced RISC Machines (ARM) processors consist of a family of processors, which specialize in low cost and very low power consumption [34]. They are targeted for embedded and multi-media applications. The ARM instruction set architecture supports the conditional execution of all instructions. Each instruction has a four bit condition field that specifies the context for which it is executed. By examining the condition field of an instruction and the condition codes in a processor status register, the execution condition of each instruction is calculated. The condition codes are typically set by performing a compare instruction. The condition field specifies under what comparison result the instruction should execute, such as equals, less than, or less than or equals. When the compare instruction result contained in the processor status register matches the condition field, the instruction is executed. Otherwise, the instruction is nullified. With this support, the ARM compiler is able to eliminate conditional branches from the instruction stream.

1.4.4 Texas Instruments C6X

Texas Instruments' TMS320C6000 VelociTI Architecture supports a form of predicated execution. Five general purpose registers, two in the A bank and three in the B bank, may be used to guard the execution of instructions, either in the positive (execute when the register contains a non-zero value) or negative (execute when the register contains zero) sense. These values may be written using any ordinary instruction, and a set of comparison instructions which deposit into these registers is also provided.

1.4.5 Systems with limited predicated execution support

Many other contemporary processors offer some form of limited support for predicated execution. A conditional move instruction is provided in the DEC Alpha, SPARC V9, and Intel Pentium Pro processor instruction sets [35],[36],[37]. A conditional move is functionally equivalent to that of a predicated move. The move instruction is augmented with an additional source operand which specifies a condition. As with a predicated move, the contents of the source register are copied to the destination register if the condition is true. Otherwise, the instruction does nothing. The DEC GEM compiler can efficiently remove branches utilizing conditional moves for simple control constructs [38]. The HP PA-RISC instruction set provides all branch, arithmetic, and logic instructions the capability to conditionally nullify the subsequent instruction [39].

The Multiflow Trace 300 series machines supported limited predicated execution by providing *select* instructions [40]. Select instructions provide more flexibility than conditional moves by adding a third source operand. The semantics of a select instruction in C notation are as follows:

```
select dest,src1,src2,cond
dest = ( (cond) ? src1 : src2 )
```

Unlike the conditional move instruction, the destination register is always modified with a select instruction. If the condition is true, the contents of *src1* are copied to the destination; otherwise, the contents of *src2* are copied to the destination register. The ability to choose one of two values to place in the destination register allows the compiler to effectively choose between computations from “then” and “else” paths of conditionals based upon the result of the appropriate comparison.

Vector machines have had support for conditional execution using mask vectors for many years [41]. A mask of a statement S is a logical expression whose value at execution time specifies whether or not S is to be executed. The use of mask vectors allows vectorizing compilers to vectorize inner loops with if-then-else statements.

Although there are a great variety of architectural styles for predicated execution, most can be mapped to the generalized IMPACT EPIC model presented earlier.

1.5 In Depth: Predication in Intel IA-64

Perhaps the mostly widely known architecture employing predication is Intel’s IA-64 [42]. The IA-64 architecture is implemented by Intel’s Itanium

<i>ctype</i>	<i>p</i> ₁	<i>p</i> ₂
none	CT	CF
unc	UT	UF
or	OT	OT
and	AT	AT
or.andcm	OT	AF

FIGURE 1.4: The comparison types available with IA-64 compare instructions.

line of processors. The Itanium processors make good candidates to evaluate predication given the ongoing effort to continually improve its design and its compiler technology.

The Intel IA-64 architecture supports full predication with a bank of 64 predicate registers and a limited set of two-destination predicate defining instructions. Predicate defines are simply called *compares* in IA-64, a consequence of the fact that the only comparison instructions that exist all have predicate, not integer, destination registers. The compare instructions have the format:

$$(qp) \text{ cmp.crel.ctype } p_1, p_2 = r_2, r_3$$

where the *qp* is the qualifying predicate, the *crel* is the comparison relation (eq, ne, lt, le, gt, ge, ltu, leu, gtu), and the *ctype* is the comparison type. IA-64 supports the four basic comparison types proposed in the HP PlayDoh model in its compare instructions, although only in limited combinations within a single instruction. These are shown in Figure 1.4.

Since not all combinations of deposit types can be combined into a single predicate defining instruction, the compiler is forced to generate sub-optimal comparison code in some situations when compared to the PlayDoh or IMPACT EPIC models. Further, some compilation techniques presented for predicate optimization are only applicable with significant modification likely to lead to inefficiencies [29]. Nevertheless, it is interesting to explore the role predication plays in a commercial product such as the Itanium 2.

1.5.1 Predication in the Itanium 2 processor

As of this writing, the Itanium 2 processor represents the most advanced implementation of Intel's IA-64 ISA. The Itanium 2 is a six-issue processor with six ALUs, two FPUs, and three branch units. The two of the six ALUs also serve as load units, and another two serve as store units. Itanium 2 issues instructions in order and does not perform register renaming. The Itanium 2 employs an aggressive branch predictor and instruction fetch is decoupled from the back end by a 48 operation buffer [43]. Conditional branches in Itanium 2

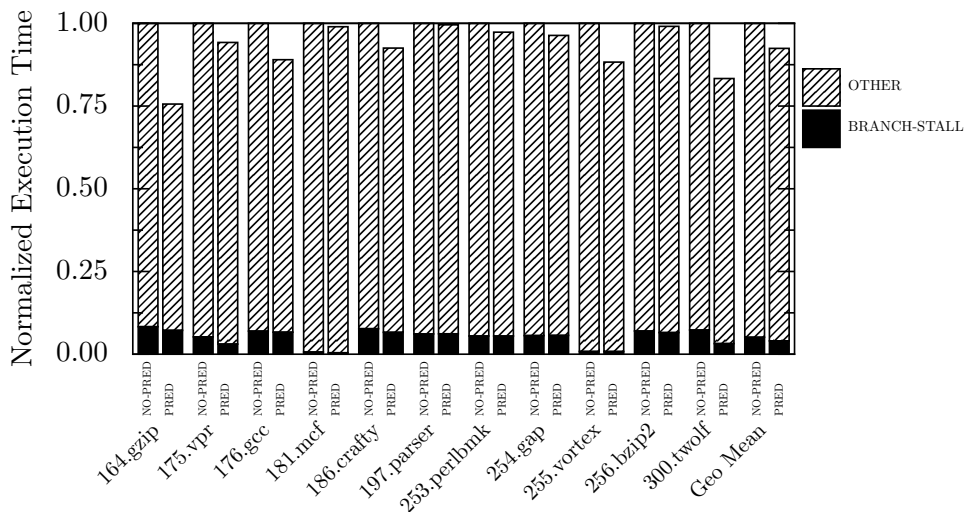


FIGURE 1.5: Execution time of SPEC benchmarks with (PRED) and without (NO-PRED) predication (normalized to NO-PRED). Total time is subdivided into branch misprediction stalls and other.

All ILP achieved is expressed in advance by the compiler in the form of instruction groups. These groups are delimited by the compiler using stop bits in the executable. As an implementation of IA-64, almost all instructions in the Itanium 2 are guarded by a qualifying predicate. In addition, compiler controlled data and control speculation are supported.

Figures 1.5 and 1.6 show the impact predication has on performance of the Itanium 2. These figures show characteristics of SPEC 2000 codes generated by version 7.1 of Intel’s Electron compiler using aggressive optimization settings running on 900Mhz Itanium 2 processors. Breakdown of the performance is obtained using the hardware performance monitoring features of the Itanium 2, collected using Perfmon kernel support and the Pfmmon tool developed by Hewlett-Packard Laboratories [44].

Figure 1.5 shows the performance of the benchmarks without and with predication. Overall, predication helps to achieve a 8.2% reduction in cycle count. 164.gzip, the benchmark responding best to predication, shows a 25% reduction in cycles is achieved. Interestingly, the change in the amount of time spent in branch stalls is only slightly reduced. This suggests that, contrary to expectations, the performance enhancement derived from predication is not due to a reduction in total branch mispredictions. This is consistent with measurements by others [45].

Figure 1.6 shows that predication improves performance primarily by reducing load-use stalls and by increasing utilization of empty issue slots. Both of these effects are a consequence of the increased scheduling freedom and

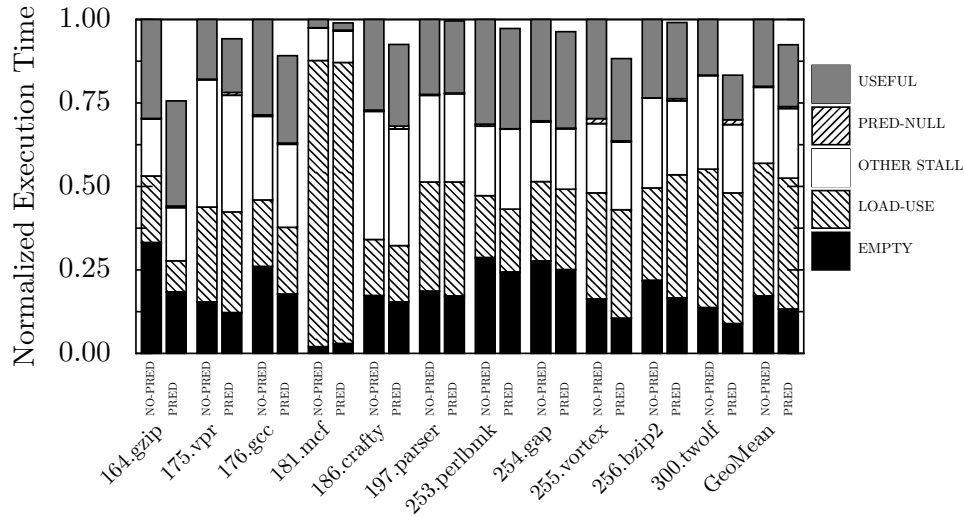


FIGURE 1.6: Execution time of SPEC benchmarks with (PRED) and without (NO-PRED) predication (normalized to NO-PRED). Total time is subdivided into time wasted performing no work (empty execution slots), stalls due to load-use (cache misses), other types of stalls, nullifying instructions, and executing useful work.

optimization opportunities provided by prediction. In the case of load-use stalls, this scheduling freedom enables the compiler to find useful work to hide cache misses. Since the Itanium 2 stalls on use, work can be performed on code following a load missing in the cache so long as it doesn't use the value. Scheduling freedom provided by predication frees up more instructions for this purpose. In the case of empty issue slots, the added scheduling freedom allows instructions to be issued in cases where execution resources would otherwise go unused. This increase in ILP exposed by the compiler translates directly to performance during execution.

Sias et al. perform a thorough analysis of predication and other ILP enhancing techniques using the IMPACT compiler on Itanium 2 [22].

1.6 Predication in Hardware Design

As described earlier in this chapter, by removing branches, predication reduces the need to speculate early in the pipeline. Instead, in using predication, the compiler speculates that the cost of fetching additional instructions with predication is offset by the savings obtained by removing branch mispredic-

1	(1) $p1, \bar{p2} = \text{Cond1}$	
2	(2) $r1 = 0$	$\langle p1 \rangle$ (3) $r1 = 1$ $\langle p2 \rangle$
3	(4) $r2 = r1 + 5$	

FIGURE 1.7: A simple code segment illustrating the multiple definition problem.

tions, by reducing branch resource consumption, and by enhancing optimizations in the compiler. There are however new hardware design issues raised by predication which must also be considered.

While predication eliminates the need to make decisions that necessitate speculation early in the pipeline, predication still requires decisions to be made later in the pipeline. These decisions may ultimately require speculation in more aggressive architectures. These decisions stem from the *multiple-definition problem*.

Figure 1.7 shows a very simple example of the multiple-definition problem. In this example, the predicate define instruction 1 in cycle 1 sets the value of predicates P_1 and its complement, P_2 . In cycle 2, only one of the two instructions issued concurrently (2 or 3) will execute while the other is nullified. Finally, in cycle 3, instruction 4 will either get the value of $r1$ from instruction 2 or from instruction 3 depending on the value of the predicates P_1 and P_2 . At this point, $r2$ will either have the value 5 or 6. A relatively simple in-order machine has no problem with this sequence of events. This, however, is not true for more aggressive architectures.

Consider this code example in an extremely aggressive in-order pipeline. In such a case, the value produced in cycle 2 may not have time to be recorded in the single architectural register file. Instead, register bypass logic may be necessary to deliver the value of $r1$ to instruction 4. Since the bypass either forwards the data from the execution unit handling instruction 2 or the execution unit handling instruction 3 depending on the value of the predicates, the value of the predicate must be known in time. This may be problematic, however, as the value of these predicates are only known after the execution of the predicate define. For instance, in a machine with incomplete bypass, the issue logic may need to know the value of the predicate before issuing instructions 2 and 3. This information unlikely to be available as there is ideally no time between the completion of instruction 1 and the start of execution of instructions 2 and 3.

In a machine with a renaming stage, such as is found in architectures supporting out-of-order execution, the problem is much worse. In such machines, a renaming stage must give physical register names to the instances of $r1$ in instructions 2, 3, and 4. This renaming must respect the actual dependence at run-time, a dependence determined by the value of the predicate. Since the rename stage is before the execute stage, the predicate information is typically unavailable at the time registers are renamed, as is the case in this

1	(1) $p1, \overline{p2} = \text{Cond1}$	
2	(2) $r1 = 0$	<p1>
3	(3) if (p2) $r1 = 1$ else $r1 = r1$	
4	(4) $r2 = r1 + 5$	

FIGURE 1.8: Serialization to solve the multidef problem.

example. The naive solution, to stall the processor at rename until predicates are computed is in general unsatisfactory.

Several solutions have been proposed in the literature to address this problem. Perhaps the simplest is to serialize predicated instructions 2 and 3 in the compiler or hardware [40, 46]. This solution is illustrated in Figure 1.8. With serialization, instruction 3 consumes the value of instruction 2. In cases where instruction 3 would be nullified, the instruction now is still executed. Instead of doing nothing, the instruction now performs a physical register move of the result of instruction 2 into the register destined to be sourced by instruction 4. While this eliminates the need to stall in the rename stage, the cost of this solution is still high as it causes the serialization of predicated instructions. The solution also requires an additional source operand, as can be seen in instruction 3, that adds complexity to the design of the datapath. A similar technique is used to deal with the CMOV, conditional move instruction, in the Alpha 21264 [47].

To reduce this serialization, Wang et al. proposed an approach based upon the notion of static-single-assignment (SSA) [48]. The idea defers renaming decisions by inserting additional instructions, called *select- μ ops* which resemble phi-nodes, before instructions which need the computed value. These additional instructions are responsible for selecting between the set of definitions based on predicate values. The presence of the *select- μ ops* allows destination registers of predicated instructions, instructions 2 and 3 in the figure, to be named without regard to the predicate relationship between the multiple definitions and the original use.

Another solution employs a form of speculation, called predicate prediction, has been proposed [49]. The proposed predicate prediction mechanism speculatively assumes a value for the predicate to be used in the renaming process, and provides a clever means to recover from mispredictions which avoids resource consumption by known nullified instructions. Consequently, the penalty due to mispredicting a predicate was shown to be less severe than mispredicting a branch. In the compiler, predicate mispredictions influence factors used the compiler heuristics, but leave the basic principles unchanged.

1.7 The Future of Predication

While predication has realized significant performance gains in actual processors, predication's true potential has not yet been achieved. As described earlier in Section 1.3, predication enhances and augments ILP optimizations even in the presence of perfect branch prediction. While researchers working on predicated compilations techniques see many untapped opportunities today, there likely exist many more unidentified opportunities.

As seen in Section 1.5, predication's effect on ILP is more significant than its effect removing stalls due to branch misprediction in Itanium 2. This, however, is likely to change. The longer pipelines necessary to enhance performance exacerbate the branch misprediction problem. Novel architectural and compiler techniques which speculatively execute larger and disjoint regions of code may also increase reliance on good branch predictions. Predication is likely to play a role in addressing problems introduced by these new techniques by avoiding the need to make such speculative predictions.



References

- [1] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 272–282, April 1989.
- [2] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple instruction issue," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 290–302, April 1989.
- [3] M. A. Schuette and J. P. Shen, "An instruction-level performance analysis of the Multiflow TRACE 14/300," in *Proceedings of the 24th International Workshop on Microprogramming and Microarchitecture*, pp. 2–11, November 1991.
- [4] T. M. Conte, K. Menezes, P. Mills, and B. Patel, "Optimization of instruction fetch mechanisms for high issue rates," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 333–344, June 1995.
- [5] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: A low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 24–34, December 1996.
- [6] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135–148, May 1981.
- [7] T. Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 51–61, November 1991.
- [8] P. Y. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 386–395, June 1986.
- [9] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, vol. 22, pp. 12–35, January 1989.

- [10] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pp. 177–189, January 1983.
- [11] J. C. Park and M. S. Schlansker, "On predicated execution," Tech. Rep. HPL-91-58, Hewlett Packard Laboratories, Palo Alto, CA, May 1991.
- [12] D. N. Pnevmatikatos and G. S. Sohi, "Guarded execution and branch prediction in dynamic ILP processors," in *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 120–129, April 1994.
- [13] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu, "Characterizing the impact of predicated execution on branch prediction," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 217–227, December 1994.
- [14] G. S. Tyson, "The effects of predicated execution on branch prediction," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 196–206, December 1994.
- [15] D. I. August, K. M. Crozier, J. W. Sias, D. A. Connors, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu, "The IMPACT EPIC 1.0 Architecture and Instruction Set reference manual," Tech. Rep. IMPACT-98-04, IMPACT, University of Illinois, Urbana, IL, February 1998.
- [16] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: Version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA, February 1994.
- [17] M. Schlansker, V. Kathail, and S. Anik, "Height reduction of control recurrences for ILP processors," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 40–51, December 1994.
- [18] M. Schlansker and V. Kathail, "Critical path reduction for scalar programs," in *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 57–69, December 1995.
- [19] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu, "Integrated predication and speculative execution in the IMPACT EPIC architecture," in *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 227–237, June 1998.
- [20] D. I. August, W. W. Hwu, and S. A. Mahlke, "A framework for balancing control flow and predication," in *International Symposium on Microarchitecture*, pp. 92–103, 1997.
- [21] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann, and W. W. Hwu, "Effective compiler support for predicated execution using

- the hyperblock,” in *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 45–54, December 1992.
- [22] J. W. Sias, S. zee Ueng, G. A. Kent, I. M. Steiner, E. M. Nystrom, and W. mei W. Hwu, “Field-testing IMPACT EPIC research results in Itanium 2,” in *Proceedings of the 31st Annual International Symposium on Computer Architecture*.
- [23] A. Klauser, T. Austin, D. Grunwald, and B. Calder, “Dynamic hammock predication for non-predicated instruction set architectures,” in *Proceedings of the 18th Annual International Conference on Parallel Architectures and Compilation Techniques*, pp. 278–285, October 1998.
- [24] R. Johnson and M. Schlansker, “Analysis techniques for predicated code,” in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 100–113, December 1996.
- [25] J. W. Sias, W. W. Hwu, and D. I. August, “Accurate and efficient predicate analysis with binary decision diagrams,” in *Proceedings of 33rd Annual International Symposium on Microarchitecture*, pp. 112–123, December 2000.
- [26] D. I. August, *Systematic Compilation for Predicated Execution*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois, 2000.
- [27] S. A. Mahlke, *Exploiting Instruction Level Parallelism in the Presence of Conditional Branches*. PhD thesis, University of Illinois, Urbana, IL, 1995.
- [28] D. M. Gillies, D. R. Ju, R. Johnson, and M. Schlansker, “Global predicate analysis and its application to register allocation,” in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 114–125, December 1996.
- [29] D. I. August, J. W. Sias, J. Puiatti, S. A. Mahlke, D. A. Connors, K. M. Crozier, and W. W. Hwu, “The program decision logic approach to predicated execution,” in *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 208–219, May 1999.
- [30] N. J. Warter, *Modulo Scheduling with Isomorphic Control Transformations*. PhD thesis, University of Illinois, Urbana, IL, 1993.
- [31] G. R. Beck, D. W. Yen, and T. L. Anderson, “The Cydra 5 minisupercomputer: Architecture and implementation,” *The Journal of Supercomputing*, vol. 7, pp. 143–180, January 1993.
- [32] J. C. Dehnert and R. A. Towle, “Compiling for the Cydra 5,” *The Journal of Supercomputing*, vol. 7, pp. 181–227, January 1993.

- [33] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt, "Overlapped loop support in the Cydra 5," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 26–38, April 1989.
- [34] A. V. Someren and C. Atack, *The ARM RISC Chip, A Programmer's Guide*. Reading, MA: Addison-Wesley Publishing Company, 1994.
- [35] D. L. Weaver and T. Germond, *The SPARC Architecture Manual*. SPARC International, Inc., Menlo Park, CA, 1994.
- [36] Digital Equipment Corporation, *Alpha Architecture Handbook*. Maynard, MA: Digital Equipment Corporation, 1992.
- [37] Intel Corporation, Santa Clara, CA, *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, 1997.
- [38] D. S. Blickstein, P. W. Craig, C. S. Davidson, R. N. Faiman, K. D. Glosop, R. B. Grove, S. O. Hobbs, and W. B. Noyce, "The GEM optimizing compiler system," *Digital Technical Journal*, vol. 4, no. 4, pp. 121–136, 1992.
- [39] Hewlett-Packard Company, Cupertino, CA, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1990.
- [40] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg, "The Multiflow Trace scheduling compiler," *The Journal of Supercomputing*, vol. 7, pp. 51–142, January 1993.
- [41] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*. Reading, MA: Addison-Wesley Publishing Company, 1991.
- [42] Intel Corporation, *IA-64 Application Developer's Architecture Guide*, May 1999.
- [43] C. McNairy and D. Soltis, "Itanium 2 processor microarchitecture," *IEEE MICRO*, vol. 23, pp. 44–55, March.
- [44] S. Eranian, "Perfmon: Linux performance monitoring for IA-64." Downloadable software with documentation, <http://www.hpl.hp.com/research/linux/perfmon/>, 2003.
- [45] Y. Choi, A. Knies, L. Gerke, and T.-F. Ngai, "The impact of if-conversion and branch prediction on program execution on the intel Itanium processor," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pp. 182–191, IEEE Computer Society, 2001.
- [46] P. Y. Chang, E. Hao, Y. Patt, and P. P. Chang, "Using predicated execution to improve the performance of a dynamically scheduled machine

- with speculative execution,” in *Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques*, June 1995.
- [47] R. Kessler, “The Alpha 21264 microprocessor,” *IEEE Micro*, vol. 19, pp. 24–36, March/April 1991.
- [48] P. H. Wang, H. Wang, R.-M. Kling, K. Ramakrishnan, and J. P. Shen, “Register renaming and scheduling for dynamic execution of predicated code,” in *The 7th International Symposium on High-Performance Computer Architecture*, January 2001.
- [49] W. Chuang and B. Calder, “Predicate prediction for efficient out-of-order execution,” in *Proceedings of the 17th annual international conference on Supercomputing*, pp. 183–192, ACM Press, 2003.