HEP: Hidden-Explicit Predication

Abstract

Our new predication technique increases the performance of a typical computer design employing traditional control flow methods by realizing full predication. In other words, within the instruction window of a computer, branches only set predicate values; all non-branch instructions are predicated. Arbitrary control flow is allowed, such as nested and overlapped branches. The cost is moderate and is much less than competing techniques. The control flow conversion to predication is done transparently to the computer's immediate user, i.e., a compiler writer or an assembly language programmer. Thus, code compatibility is not a problem and HEP can be applied to any processor, with or without existing predication. With HEP minimal control dependencies are realized (this is orthogonal to control speculation). In this paper the new method is described, including the presentation of the novel concept of a *cancelling predicate*.

1 Introduction

One common method used to improve processor performance is to reduce the dependencies of machine instructions on each other, so that more of the instructions can execute in parallel, reducing program execution time. There are several types of dependencies extant in code; in this paper we concentrate on reducing *control* dependencies. Note that this is orthogonal to speculation methods. One way to reduce control dependencies is to use *predication*, the enabling or disabling of individual instructions based on single-bit predicate values; the predicates are typically set by comparison instructions; hence arbitrary control-flow can be achieved by predicating all instructions (*full predication*).

Most commercial computers have little or no predication. For code compatibility reasons, existing classic architecture designs such as Intel x86 processors cannot currently take advantage of full predication. Processors that do have full predication, or the equivalent, are typically unalterable or have a large amount of complex hardware. Common predication techniques are *visible*, that is the predicates appear in the instruction set of the processor. Typically predicated processors use *explicit* predicate registers, that is they have a predicate register associated with each predicatable instruction. In some research techniques predication is *implicit*; no single predicate register is associated with predicatable instructions. Therefore no existing technique achieves all desirable characteristics; no single processor exhibits full predication, invisible or *hidden* predicates, little hardware and simple hardware.

We propose *Hidden-Explicit Predication* or *HEP* hardware that realizes full predication without compiler or instruction set assistance and is relatively cheap and simple. Another major advantage of HEP is its applicability to all existing computer designs, including both those with and without predicates in their instruction set architectures. For example, common Intel x86 or IA-32 instruction set processors with no or little predication can have full predication added without affecting the instruction set. Also, machines containing predication operations in their instruction sets, such as many VLIW/EPIC (Very Long Instruction Word) computers (e.g., Intel Itanium) can also benefit, in that the number of predicates in the machine can be increased as much as desired without changing the existing predication operations or instruction set.

2 Background and Prior Work

The vast majority of computer programs use traditional control flow constructs to determine both when and if instructions in the program are executed. Such constructs include:

`if-then-else` statements and looping statements such as:
`while(condition is true){ ... }` or:
`for(i = 1; i < 10; i++){ ... }` or:
`do i=1 to 10 ... enddo`

The vast majority of such control statements are realized with machine-level instructions called *branches*, and most of these are *conditional* branches. The problem with the classic view of conditional branches is the assumption that every instruction dynamically or statically after a branch is control dependent on that branch and must wait for the branch to execute before executing itself. This has been well-demonstrated to be a significant barrier to realizing much parallelism within a program, keeping performance gains low[1, 2]. Control dependencies are not removed by control speculation techniques.

Predication is used to reduce classic control dependencies and therefore improve performance. A predicate is typically a one-bit variable having the values true or false; it is usually set by a comparison instruction. In this model ideally every instruction has a predicate as an additional input. The semantics are that the instruction is only effectively executed (its output state changed) if the predicate is true. An example of equivalent classic control flow and modern predication follows.

```
      Classic code:                  Predicated code:
1.  if (a==b) {                1. Pred = (a==b);  //Pred set to true if a equals b.
2.      z=x+y;                 2. IF (Pred) THEN z=x+y; //Operations performed only
3.      w=a+b; }               3. IF (Pred) THEN w=a+b; //    if Pred true.
4.  // later instructions:     4. // later instructions: NOT dependent on 1.
    //   all dependent on 1.
```

With predication, only the instructions having the equivalent predicate as an input are dependent on the branch-remnant (the comparison operation). In the example and in general, this means the instructions after the predicated instructions are now independent of the branch-remnant and may be executed in parallel with instructions before the branch-remnant, improving performance.

Current approaches to using predication use *visible* and *explicit* predicates[3, 4]; the predicates are controlled by the computer user and they use storage explicitly present in the computer's instruction set architecture (similar to regular data registers or main memory); they are explicit since there is at least one 1-bit predicate hardware register associated with each instruction. The most extreme example of this is the IA-64 (Intel Architecture-64 bits) architecture[5]. The first realization of this architecture is the Itanium (formerly Merced) processor, due to be on the market in the year 2001. Itanium has 64 visible-explicit predicate registers[6]. The predicates cannot be effectively used when the processor executes traditional IA-32 (x86) machine code. Therefore, existing software cannot take advantage of

Itanium without modification. Other types of microprocessors have similar constraints to x86 processors; predicates are not currently in their instruction set, so they cannot take advantage of predication techniques. A limited form of "automatic predication" was attempted in the HAL SPARC64 V [7].

The Transmeta Crusoe computer can have arbitrary predication in its underlying hardware. Our form of predication could also be used here to reduce hardware cost, reduce the load on the code *morphing* front end, help keep the design simple and hence reduce power consumption.

All of these features arise partially from the distributed nature of our full predication technique; the predicate registers are fully distributed among the instructions in the instruction window, and no central predication register file is needed. The distributed predicates and distributed predication operations also help keep the critical path delay low since their electrical signals need not travel far on the chip and there is no contention for accessing a central predication register file, as opposed to traditional predication methods.

It is of course possible to predicate just a subset of the instructions of a processor, as in the Intel P6 microarchitecture and Compaq Alpha. but then the benefits of predication are much less. Therefore *full predication* is preferred.

Visible-explicit predication first appeared as "guards" in the supercomputing field in the 1960's and '70's. A limited form of hidden-explicit predication is presented in [8], but it is not truly hidden in that the compiler must delineate the domains. It also only predicates branches not part of nested or overlapped branch domains, relying on standard branching techniques for other branches. Our method allows full predication.

Reduced control dependencies were originally studied in [9]. Other methods used to reduce the ill effects of branches may be found in: [10] - determined the effect of eager execution (complete multipath execution) on performance and cost; [11, 12] - loop-based software methods for vector and multi-processors; [13, 14, 15, 16, 17] - software pipelining and/or VLIW approaches; [18] - a compendium of branch types and effect reduction methods; [19, 20] - collections of branch prediction methods and branch target buffer designs; [21] -

effect of branches on pipelining; [22] - a more recent and general treatment of branch effect reduction techniques; [23] - a graph-based technique; [24] - code percolation; [25] - instruction boosting in software; [26] - hardware-based trace scheduling, used in the Intel Pentium 4; and [27] - the multiscalar model - it executes multiple basic blocks concurrently, but does not reduce control dependencies (as we have defined them).

In prior work[1, 28, 29] a method for realizing an equivalent to full predication called *minimal control dependencies* or MCD was devised. It produced substantial performance gains[29, 30]; this was also shown by Lam and Wilson[2] with their CD-MF model (same as MCD).

Performance improves even more when MCD is coupled with another performance-enhancing technique called *disjoint eager execution*[31]. MCD can be considered to have *hidden* and *implicit* predicates, in that the predicates are not visible to the user, nor are they explicitly present in the processor.

However, MCD has disadvantages when compared to predication such as having a high hardware cost (more logic gates and storage needed) with relatively complex hardware. In particular, *j-by-j* diagonal bit matrices are required, where j is the number of instructions in the instruction window (those instructions currently under consideration for execution by the processor). In a high-ILP machine, j might be 256 or more, leading to a cumbersome 32,000 or more bit diagonal matrix. Further, all of the bits need to be accessed and operated on at the same time, leading to a very complex and potentially slow hardware layout. Lastly, setting the contents of the matrix when instructions are loaded into the processor is also costly and potentially slow.

HEP realizes the best features of both visible-explicit predicates and hidden-implicit predicates without the disadvantages.

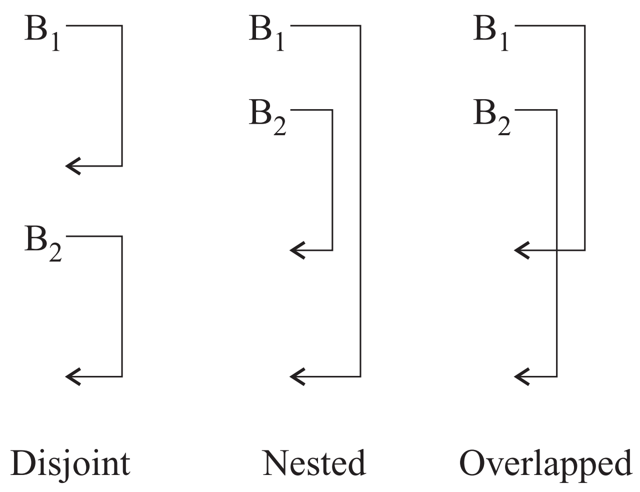


Figure 1: *Branch arrangements.* Any system realizing full predication must be able to handle any combination of these three basic branch arrangements.

3 Our New HEP Hardware for Full Predication

Hidden-explicit predication (HEP) realizes full predication; the predicates are not visible to the user and thus may be implemented in any processor architecture, and the predicates occupy explicit hardware register bits in the processor, reducing cost and complexity. There are two parts to HEP: the *predicate-assignment* part, taking place when instructions are loaded into the processor, and the *predicate-use* part, taking place at instruction execution time.

Nomenclature: A branch's *domain* consists of the instructions occurring between the branch and the branch's target. Thus, the branch controls the execution of the instructions within its domain. If the branch's condition evaluates true the branch is *taken*, and the instructions in its domain are not executed. If the branch's condition evaluates false, the branch is *not taken* and the instructions in the branch's domain are allowed to execute.

Multiple branch domains can be arranged in an unlimited number of ways, all of which are combinations of the three basic arrangements: *disjoint*, *nested* and *overlapped*, as shown in Figure 1. For full predication all possible combinations of these arrangements must be handled correctly. HEP does this.

Key Ideas: The predicate-assignment hardware detects the beginnings and ends of branch domains. The predicate-use hardware employs this information to realize the beginnings and ends of domains at execution time, performing the appropriate enabling and disabling of instructions in domains. Backwards branches are handled at instruction load time by converting them to forwards branches: the direction of a backwards branch target offset is reversed, then loading is resumed from the branch's target address.

In general, as each new domain is encountered during code execution, a new condition is placed on the execution of the code within the new domain. If the branch condition of the domain's branch is bc_i, and the predicate of the code before the branch is p_r, then the effective predicate p_e of the new code in the new branch's domain must be computed as:

$$p_e = \overline{bc_i} \cdot p_r$$

When a domain is exited (upon reaching the corresponding branch's target instruction) the effect of the corresponding branch must be nullified, in other words bc_i should have no effect on the execution of the following code. This is achieved by effectively OR-ing the branch condition with the current predicate; in other words, the following is effectively computed for the code after the branch domain:

$$p_{e2} = p_e + (bc_i \cdot p_r) = (\overline{bc_i} \cdot p_r) + (bc_i \cdot p_r) = p_r$$

This logic is realized by the combined operation of the predicate-assignment and predicate-use hardware.

3.1 Predicate-Assignment Hardware and Operation

The function of the predicate-assignment hardware is to assign predicate and *cancelling predicate* addresses to instructions as they are loaded into the processor's load buffer and before the buffer contents are sent to the instruction window for execution. The assignment is done by detecting domain entries (branches) and exits (targets). The basic hardware structure is a *branch tracking stack* or buffer as shown in Figure 2. (We use the term "stack" in its generic sense; any kind of temporary storage may likely be used.)

Each entry (row) in the buffer corresponds to one branch. Typically, but not necessarily, a branch is in the buffer only while the instruction load pointer *ilptr* value is within the branch's domain. The following fields compose each entry:

1. address of predicate corresponding to the branch p_b
2. address of cancelling predicate corresponding to the branch cp_b; in practice this may be derived from the branch's predicate address, so no explicit entry would be needed for cancelling predicate addresses
3. target address of the branch ta_b
4. valid bit flag v_b; true while the target of the corresponding branch has not yet been reached; the buffer entry may be reclaimed and reused when the valid bit is false

A branch is placed in the buffer when it is encountered by the *ilptr* and is removed when its target is reached. In the case of overlapped branches, the target for a branch may be reached before a prior branch's target has been reached; in this case the overlapped branch has its v bit cleared and is removed from the buffer when convenient.

The comparators look for a match between the *ilptr* and the target addresses. If there is a match, it means that the instruction just loaded is the target of the matching branch (multiple matches will be considered later). The current cancelling predicate address cp_T is set equal to the cancelling predicate address of the matching branch. cp_T is entered into the cancelling predicate address field of the instruction being loaded.

Out-of-Bounds Branches: So far only branches with targets inside the window have been considered. It is also possible that a branch in the window could jump to a point not yet encountered by the predicate-assignment hardware. Therefore the hardware shown in Figure 2 is augmented with some more circuitry to handle these *out-of-bounds* branches.

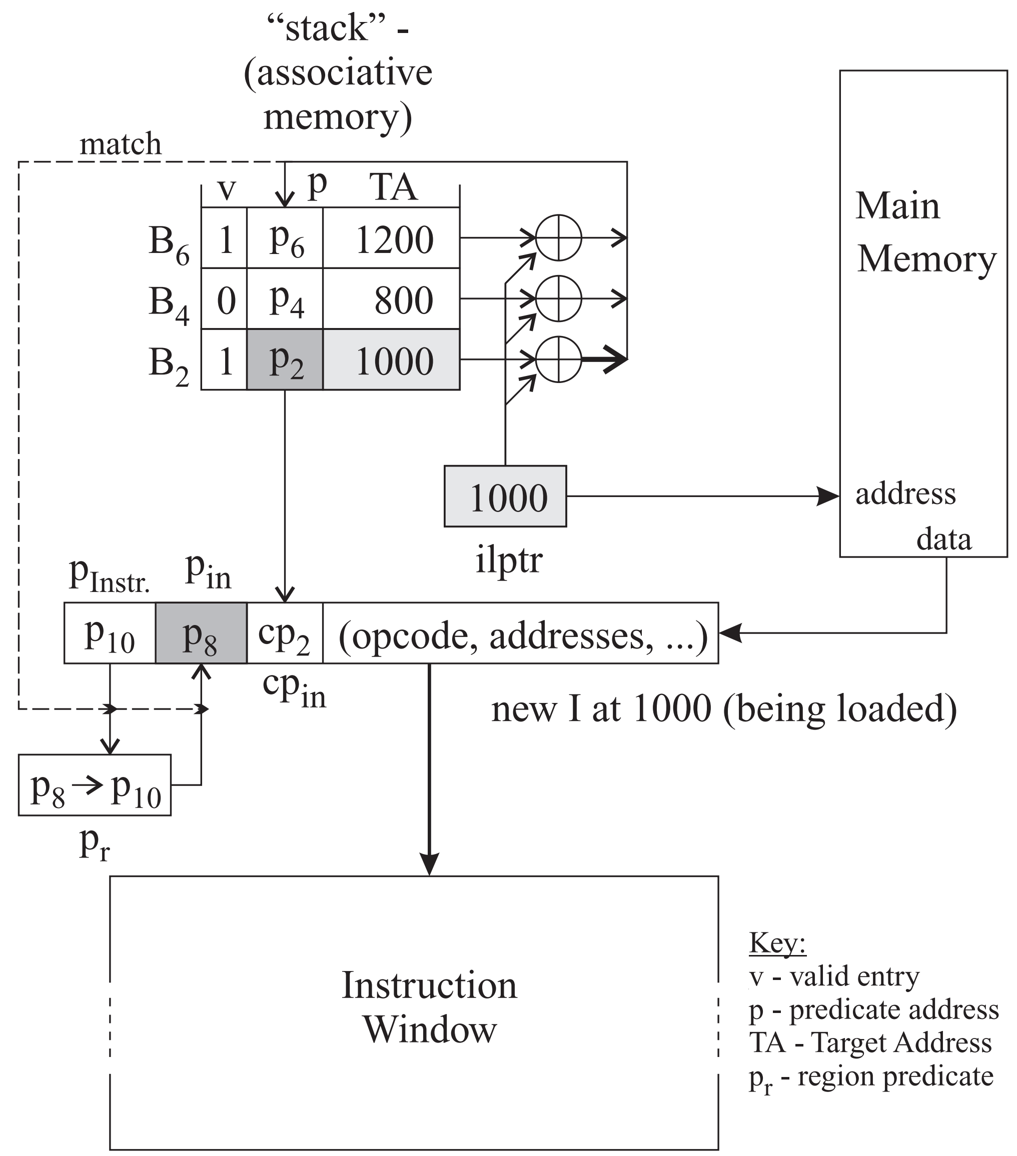


Figure 2: *Predicate-assignment hardware.* The snapshot is taken at load time 9+ of Example 4; see Figure 6. So far three branches have been loaded. The end of branch B_4's domain, its target, has also been loaded, as is indicated by the branch's valid bit being cleared. The buffer is associatively addressed by the current value of the *ilptr* (instruction load pointer). At this time the instruction at 1000 is loaded; its address matches a target address in the buffer. The predicate address of the branch corresponding to the target address match with the *ilptr* is used for the loading instruction's input cancelling predicate address, cp_{in}. The p_r register holds the address of the current region's predicate; at this time it is loaded into the input predicate address field p_{in} of the new instruction. Simultaneously p_r is updated with the predicate address corresponding to the loading instruction itself, p_{10}. (In general, p_r may point to the predicate from either a branch or a branch target.)

The new circuitry consists primarily of another set of comparators for performing associative lookups on field "p".

The technique is as follows. A candidate branch for execution supplies its predicate address to the tracking buffer circuitry. The address is used as a key to perform a lookup on the "p" field. If a branch's domain is wholly contained in the window, then the branch will not have a valid entry in the buffer. Therefore if the candidate branch does obtain a valid match, it is an out-of-bounds branch; further, the branch's target address is then read from the corresponding TA tracking buffer entry. The latter reduces storage costs, as target addresses need not be stored in the window, and also simplifies operation because target addresses do not need to be read from the window.

3.2 Predicate-Use Hardware and Operation

The Predicate-Use (PredU) hardware augments the state and operations of instructions held in the processor's instruction window. None of the PredU hardware is visible to the user, i.e., it does not appear in the processor's instruction set architecture, and thus may be applied to any type of processor.

The overall effect of the PredU hardware is to chain predicate sources and sinks so as to both enforce the functionality of the system and to keep the hardware cost low. The alternative to chaining the predicates is to have many predicate inputs for each instruction, which would be costly in terms of additional instruction state and therefore also more complex in operation.

The PredU hardware and operations differ depending on whether the instruction is a branch or an assignment statement. Both cases are now considered.

Branch PredU Hardware and Operations: The output predicates are evaluated or re-evaluated whenever the input predicate or branch condition becomes available or changes value, resp.

Input: p_r - predicate of region, same as input predicate p_{in}

Outputs:

branch predicate:

$$p_{out} = \overline{bc} \cdot p_r$$

branch cancelling predicate:

$$cp_{out} = bc \cdot p_r$$

bc is the Branch Condition of this branch. bc has the values true (1) and false (0). It is set as the result of some comparison test operation such as: $A < B$. The comparison may be performed either as part of the branch's execution or as part of a prior instruction, depending on the processor architecture.

Execution Enabling Predicate: Typically none. The branch executes whenever its inputs are available or change value. Therefore all branches in the instruction window may execute in parallel and out-of-order.

Assignment Statement PredU Hardware and Operation: Assignment statements also have predicate inputs and outputs. These are used both for predicate-chaining and predicate-cancelling. Recall that predicate-cancelling occurs when a branch domain is exited.

Inputs:
p_r - predicate of region; same as input predicate p_{in}
cp_T - cancelling predicate of targeting branch, if any; same as cp_{in}

Output:

$$p_{out} = p_r + cp_T = p_{in} + cp_{in}$$

p_{out} is computed independently of the rest of the assignment statement's execution and computations.

Execution or Assignment Enabling Predicate: p_I - same as output:

$$p_I = p_{in} + cp_{in}$$

The assignment instruction may modify its traditional sinks when p_I is true. Such sinks are the results of the regular operations of the assignment statement, e.g., if the instruction is: $A = B + C$ then A is a traditional sink and is written if the instruction's predicate evaluates true.

3.3 Special Case: Multiply-Targeted Instructions

There is a not-so-special case that can often arise in code and that we have not yet addressed. This is the case when an instruction is the target of more than one branch. In this scenario the hardware as described so far will not work, it is only suitable for an instruction being the target of no more than one branch.

There are two orthogonal solutions that can be employed to handle the multiple-target case. The first is to provide multiple cancelling predicate fields for each instruction. This will cost more, but may be suitable for a small number of cancelling predicates. However, we must handle the case when an instruction is the target of many, many branches (this is possible in many machines, although perhaps not likely).

The second solution is to insert a dummy No-Op instruction into the window after the current instruction if the instruction runs out of cancelling predicate fields. The No-Op's cancelling predicates can then be used in addition to the original instruction's. Since any number of No-Ops can be inserted, any number of branches targeting the same instruction can be handled. Of course, a price is paid for a "wasted" slot in the instruction window for each No-Op instruction added.

Experimentation is needed to determine a suitable number of cancelling predicate fields for one instruction. It is likely that both solutions will be used in a typical processor.

3.4 Special Case: Branch is a Target of Another Branch

It is also quite possible that code will contain a branch that is the target of another branch. This scenario is readily handled by employing all of the predicate and cancelling predicate logic in the target branch, such that it appears as BOTH a branch and an assignment statement. The cancelling predicate output of such an instruction is the same as that of an un-targeted version of the branch. The predicate output combines the functions of the branch predicate and the assignment statement predicate, with the branch portion using the assignment portion as its region predicate input:

$$p_{out} = \overline{bc} \cdot (p_r + cp_T) = \overline{bc} \cdot (p_{in} + cp_{in})$$

This works because the assignment portion effectively (logically) takes place before the branch.

3.5 Examples

We now present four examples to illustrate the operation of the hidden-explicit predicate system. The examples cover the following cases:

1. Two disjoint branches, Figure 3. (Also covers the cases of straightline code and a single branch.)

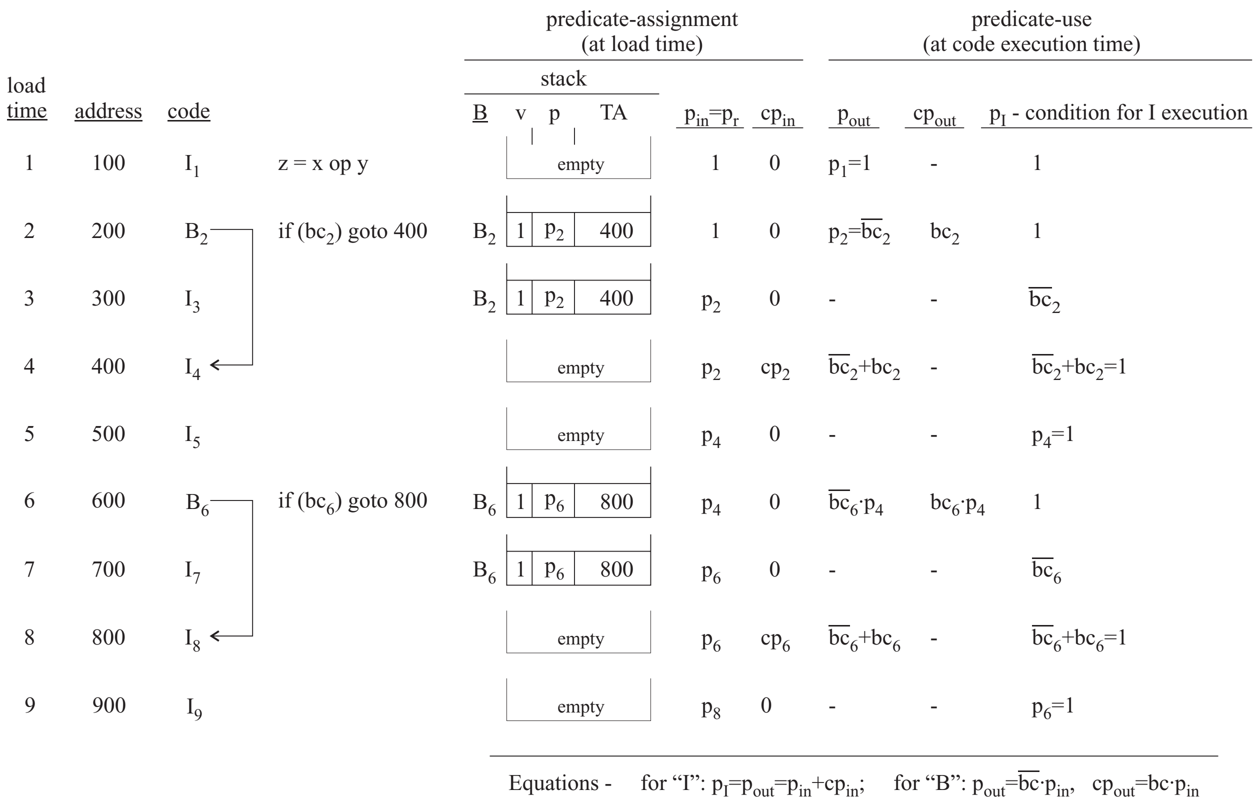
2. Two nested branches, Figure 4.

3. Two overlapped branches, Figure 5.

4. Three branches with a combination of nesting and overlapping, Figure 6.

All of the examples have the same format. In the code column: “I” instructions are assignment statements, and “B” instructions are branches. The branch domains are shown with arrowed lines. Each example can be followed by first going down the *predicate-assignment* table entries, in order, as the instructions would be loaded. Using the tracking buffer, this results in the predicate addresses shown in the p_{in} and cp_{in} columns being generated and entered into the corresponding instruction’s fields in the instruction window.

Next, the predicate-use table entries may be examined to see how the predicates are evaluated at run-time, how their values are chained and how branch domains are effectively exited. For an example of the latter, refer to Figure 3 and look at the p_I entry for the assignment instruction at address 400. Although it has predicate inputs, their values cancel each other out, p_I is effectively “1” and thus the instruction is always enabled for execution,



				predicate-assignment (at load time)						predicate-use (at code execution time)		
				stack								
load time	address	code		B	v	p	TA	$p_{in}=p_r$	cp_{in}	p_{out}	cp_{out}	p_I - condition for I execution
1	100	I_1	z = x op y			empty		1	0	$p_1=1$	-	1
2	200	B_2	if (bc_2) goto 400	B_2	1	p_2	400	1	0	$p_2=\overline{bc_2}$	bc_2	1
3	300	I_3		B_2	1	p_2	400	p_2	0	-	-	$\overline{bc_2}$
4	400	I_4				empty		p_2	cp_2	$\overline{bc_2}+bc_2$	-	$\overline{bc_2}+bc_2=1$
5	500	I_5				empty		p_4	0	-	-	$p_4=1$
6	600	B_6	if (bc_6) goto 800	B_6	1	p_6	800	p_4	0	$\overline{bc_6}\cdot p_4$	$bc_6\cdot p_4$	1
7	700	I_7		B_6	1	p_6	800	p_6	0	-	-	$\overline{bc_6}$
8	800	I_8				empty		p_6	cp_6	$\overline{bc_6}+bc_6$	-	$\overline{bc_6}+bc_6=1$
9	900	I_9				empty		p_8	0	-	-	$p_6=1$

Equations - for "I": $p_I=p_{out}=p_{in}+cp_{in}$; for "B": $p_{out}=\overline{bc}\cdot p_{in}$, $cp_{out}=bc\cdot p_{in}$

Figure 3: *Hidden-explicit predication example: disjoint branches.*

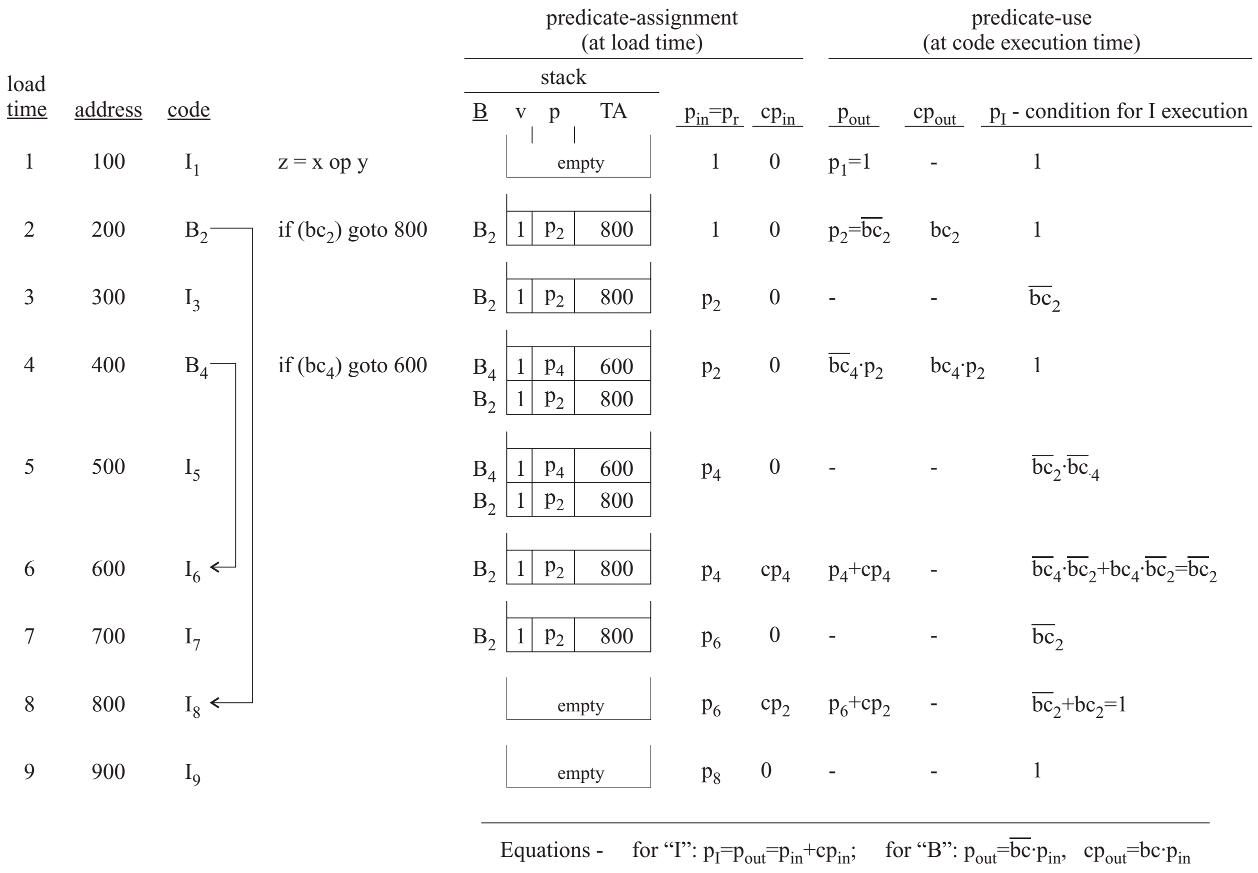
as far as branches are concerned. This is correct, since it is outside the domains of all of the branches in the code example.

4 Detailed Hardware Considerations

HEP uses more hardware than original processors but not a substantially larger amount, especially given current hardware density trends (the number of transistors on a chip doubles every 18 months - Moore's Law).

While the tracking buffer was described in a sequential fashion for ease of exposition, nothing precludes it from being realized in parallel form, that is servicing many instructions at once. This is done in the Levo machine[32]. Therefore the tracking buffer is not a bottleneck.

A concern that immediately jumps to mind when viewing the examples given earlier



				predicate-assignment (at load time)						predicate-use (at code execution time)		
				stack								
load time	address	code		B	v	p	TA	$p_{in}=p_r$	cp_{in}	p_{out}	cp_{out}	p_I - condition for I execution
1	100	I_1	z = x op y	empty				1	0	$p_1=1$	-	1
2	200	B_2	if (bc_2) goto 800	B_2	1	p_2	800	1	0	$p_2=\overline{bc}_2$	bc_2	1
3	300	I_3		B_2	1	p_2	800	p_2	0	-	-	$\overline{bc}_2$
4	400	B_4	if (bc_4) goto 600	B_4	1	p_4	600	p_2	0	$\overline{bc}_4 \cdot p_2$	$bc_4 \cdot p_2$	1
				B_2	1	p_2	800					
5	500	I_5		B_4	1	p_4	600	p_4	0	-	-	$\overline{bc}_2 \cdot \overline{bc}_4$
				B_2	1	p_2	800					
6	600	I_6		B_2	1	p_2	800	p_4	cp_4	p_4+cp_4	-	$\overline{bc}_4 \cdot \overline{bc}_2 + bc_4 \cdot \overline{bc}_2 = \overline{bc}_2$
7	700	I_7		B_2	1	p_2	800	p_6	0	-	-	$\overline{bc}_2$
8	800	I_8		empty				p_6	cp_2	p_6+cp_2	-	$\overline{bc}_2 + bc_2 = 1$
9	900	I_9		empty				p_8	0	-	-	1

Equations - for "I": $p_I=p_{out}=p_{in}+cp_{in}$; for "B": $p_{out}=\overline{bc} \cdot p_{in}$, $cp_{out}=bc \cdot p_{in}$

Figure 4: *Hidden-explicit predication example: nested branches.*

				predicate-assignment (at load time)						predicate-use (at code execution time)		
				stack								
load time	address	code		B	v	p	TA	$p_{in}=p_r$	cp_{in}	p_{out}	cp_{out}	p_I - condition for I execution
1	100	I_1	z = x op y	empty				1	0	$p_1=1$	-	1
2	200	B_2	if (bc_2) goto 600	B_2	1	p_2	600	1	0	$p_2=\overline{bc}_2$	bc_2	1
3	300	I_3		B_2	1	p_2	600	p_2	0	-	-	$\overline{bc}_2$
4	400	B_4	if (bc_4) goto 800	B_4	1	p_4	800	p_2	0	$\overline{bc}_4\cdot p_2$	$bc_4\cdot p_2$	1
				B_2	1	p_2	600					
5	500	I_5		B_4	1	p_4	800	p_4	0	-	-	$\overline{bc}_4\cdot\overline{bc}_2$
				B_2	1	p_2	600					
6	600	I_6		B_4	1	p_4	800	p_4	cp_2	p_4+cp_2	-	$(\overline{bc}_4\cdot\overline{bc}_2)+bc_2=\overline{bc}_4+bc_2$
				B_2	0	p_2	600					
7	700	I_7		B_4	1	p_4	800	p_6	0	-	-	$\overline{bc}_4+bc_2$
				B_2	0	p_2	600					
8	800	I_8		empty				p_6	cp_4	p_6+cp_4	-	$\overline{bc}_4+bc_2+(bc_4\cdot\overline{bc}_2)=1$
9	900	I_9		empty				p_8	0	-	-	1

Equations - for "I": $p_I=p_{out}=p_{in}+cp_{in}$; for "B": $p_{out}=\overline{bc}\cdot p_{in}$, $cp_{out}=bc\cdot p_{in}$

Figure 5: *Hidden-explicit predication example: overlapped branches.*

				predicate-assignment (at load time)						predicate-use (at code execution time)		
				stack								
load time	address	code		B	v	p	TA	$p_{in}=p_r$	cp_{in}	p_{out}	cp_{out}	p_I - condition for I execution
1	100	I_1	z = x op y			empty		1	0	$p_1=1$	-	1
2	200	B_2	if (bc_2) goto 1000	B_2	1	p_2	1000	1	0	$p_2=\overline{bc}_2$	bc_2	1
3	300	I_3		B_2	1	p_2	1000	p_2	0	-	-	$\overline{bc}_2$
4	400	B_4	if (bc_4) goto 800	B_4 B_2	1 1	p_4 p_2	800 1000	p_2	0	$\overline{bc}_4\cdot p_2$	$bc_4\cdot p_2$	1
5	500	I_5		B_4 B_2	1 1	p_4 p_2	800 1000	p_4	0	-	-	$\overline{bc}_4\cdot\overline{bc}_2$
6	600	B_6	if (bc_6) goto 1200	B_6 B_4 B_2	1 1 1	p_6 p_4 p_2	1200 800 1000	p_4	0	$\overline{bc}_6\cdot p_4$	$bc_6\cdot p_4$	1
7	700	I_7		B_6 B_4 B_2	1 1 1	p_6 p_4 p_2	1200 800 1000	p_6	0	-	-	$\overline{bc}_6\cdot\overline{bc}_4\cdot\overline{bc}_2$
8	800	I_8		B_6 B_4 B_2	1 0 1	p_6 p_4 p_2	1200 800 1000	p_6	cp_4	p_6+cp_4	-	$(\overline{bc}_6\cdot\overline{bc}_4\cdot\overline{bc}_2)+(bc_4\cdot\overline{bc}_2)$ $=(\overline{bc}_6+bc_4)\overline{bc}_2$
9	900	I_9		B_6 B_4 B_2	1 0 1	p_6 p_4 p_2	1200 800 1000	p_8	0	-	-	$(\overline{bc}_6+bc_4)\overline{bc}_2$
10	1000	I_{10}		B_6	1	p_6	1200	p_8	cp_2	p_8+cp_2	-	$((\overline{bc}_6+bc_4)\overline{bc}_2)+bc_2$ $=\overline{bc}_6+bc_4+bc_2$
11	1100	I_{11}		B_6	1	p_6	1200	p_{10}	0	-	-	$\overline{bc}_6+bc_4+bc_2$
12	1200	I_{12}				empty		p_{10}	cp_6	$p_{10}+cp_6$	-	$\overline{bc}_6+bc_4+bc_2+(bc_6\cdot\overline{bc}_4\cdot\overline{bc}_2)$ $=1$
13	1300	I_{13}				empty		p_{12}	0	-	-	1

Equations - for "I": $p_I=p_{out}=p_{in}+cp_{in}$; for "B": $p_{out}=\overline{bc}\cdot p_{in}$, $cp_{out}=bc\cdot p_{in}$

Figure 6: *Hidden-explicit predication example: mixed branches.*

is: doesn't the sequential chaining of the predicates and cancelling predicates necessarily sequentialize the code? In practice, it does not since a predicate with a given value is not likely to propagate all the way through the chain. It is very likely that somewhere along the way it will be ANDed with a signal having a '0' value, for example, and thus the predicate's value will not affect later code. Further, with today's accurate branch predictors predicate values are not likely to change.

5 Real Code Performance Example

An example of HEP on real code is given in Figure 7. The code snippet is from the kernel of the SPEC92 compress benchmark, as compiled into MIPS assembly code. We assume perfect knowledge of the memory addresses (they are all different - as in the actual code trace), one cycle per instruction, and the first branch is mispredicted with a penalty of 9 cycles. The baseline microarchitecture has basic ILP (register renaming, etc.) but unreduced control dependencies. The HEP microarchitecture is the same but has minimal control dependencies.

With the branch misprediction the baseline machine takes 8 more cycles to execute than the HEP machine since the HEP need not re-execute the code after the branch domain. For this limited example, this is 5 times faster than the baseline.

A cycle-accurate simulator is nearing completion. Results of simulating the SPEC2000 benchmarks will be presented in the final version of the paper.

6 Conclusions

We have presented a technique to realize full predication in hardware without any compiler or other software assistance. Several novel ideas and subsystems were presented, including: cancelling predicates, predicate address assignment with a buffer or stack, predicate chaining, and using augmented assignment statements to help realize the control flow. A hand simulation demonstrated HEP's performance gain potential.

baseline time	HEP time		MIPS instruction	
1	1		sb	v1,20960(a6)
1	1		beq	at,zero,tgt
1,(2)	1,(2)		sb	a0,0(s0)
1,10	1	tgt:	ld	s0,24(sp)
1,10	1		sd	s4,8(sp)
1,10	1		lw	a2,-32440(gp)
1,10	1		addu	a5,a3,a5
1,10	1		sll	v0,a3,3
1,10	1		bne	...

Figure 7: *HEP Code execution sample.* The HEP column gives the time of execution of the code using HEP; the baseline version without HEP, and assuming unreduced control dependencies. The "beq" branch is predicted not taken, but is actually taken. The "sb" instruction is squashed in cycle 2. With HEP, the p_I values of the instructions after "tgt:", that is those instructions not in "beq"s domain, do not change, so these instructions need not re-execute upon the misprediction. With the baseline version these instructions must re-execute.

References

[1] A. K. Uht, *Hardware Extraction of Low-Level Concurrency from Sequential Instruction Streams.* PhD thesis, Carnegie-Mellon University, December 1985. Available from University Microfilms International, Ann Arbor, Michigan, U.S.A.

[2] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 46–57, IEEE and ACM, May 1992.

[3] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu, "A Comparison of Full and Partial Predicated Execution Support for ILP Processors," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 138–149, IEEE and ACM, May 1995.

[4] W. W. Hwu and D. I. August, "Predicated Execution Architectures: New Research Results and Directions,", October 1995. Talk given at the First Intel Microprocessor Research Forum,

Santa Clara, CA, USA.

[5] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir, "Introducing the IA-64 Architecture," *IEEE Micro*, vol. 20, pp. 12–23, September/October 2000.

[6] Intel Staff, "The IA-64 Architecture Software Developer's Manual Vol. 1-4," Technical Report 24531{7,8,9}-001, Intel Corporation, 2000.

[7] M. C. Shebanow, "SPARC64 V, a High Performance System Processor,", November 1999. HAL Computer Systems, Inc. Talk given at Microprocessor Forum 1999.

[8] A. Klauser, T. Austin, D. Grunwald, and B. Calder, "Dynamic Hammock Predication for Non-predicated Instruction Set Architectures," in *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT), Paris, France*, pp. 278–285, IFIP, October 1998.

[9] G. S. Tjaden, *Representation and Detection of Concurrency Using Ordering Matrices.* PhD thesis, The Johns Hopkins University, 1972.

[10] E. M. Riseman and C. C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," *IEEE Transactions on Computers*, vol. C-21, pp. 1405–1411, December 1972.

[11] U. Banerjee and D. Gajski, "Fast Execution of Loops With IF Statements," *IEEE Transactions on Computers*, vol. C-33, pp. 1030–1033, November 1984.

[12] R. G. Cytron, "Doacross: Beyond Vectorization for Multiprocessors (Extended Abstract)," in *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 836–844, Pennsylvania State University and the IEEE Computer Society, August 1986.

[13] A. Aiken and A. Nicolau, "Perfect Pipelining: A New Loop Parallelization Technique," in *Proceedings of the 1988 European Symposium on Programming*, 1988. Also available as Dept. of Computer Science Technical Report Number 87-873, Cornell University, Ithaca, N.Y. 14853.

[14] K. Ebcioğlu, "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps," in *Proceedings of the Twentieth Annual Workshop on Microprogramming (MICRO-20)*, pp. 69–79, Association of Computing Machinery, December 1987.

[15] K. Ebcioğlu and E. R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," IBM Research Report RC 20538, IBM Research Division, August 5, 1996.

[16] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures.* PhD thesis, Yale University, New Haven, CT, 1985.

[17] B. Su, S. Ding, J. Wang, and J. Xia, "GURPR - A Method for Global Software Pipelining," in *Proceedings of the Twentieth Annual Workshop on Microprogramming (MICRO-20)*, pp. 88–96, Association of Computing Machinery, December 1987.

[18] H. G. Cragon, *Branch Strategy Taxonomy and Performance Models.* Los Alamitos, California: IEEE Computer Society Press, 1992.

[19] J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *COMPUTER*, vol. 17, pp. 6–22, January 1984.

[20] J. E. Smith, "A Study of Branch Prediction Strategies," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, pp. 135–148, IEEE and ACM, 1981.

[21] D. J. Lilja, "Reducing the Branch Penalty in Pipelined Processors," *COMPUTER*, vol. 21, pp. 47–55, July 1988.

[22] A. K. Uht, V. Sindagi, and S. Somanathan, "Branch Effect Reduction Techniques," *COMPUTER*, vol. 30, pp. 71–81, May 1997.

[23] S. Dutta and M. Franklin, "Control Flow Prediction with Tree-Like Subgraphs for Superscalar Processors," in *Proceedings of the 28th International Symposium on Microarchitecture (MICRO-28)*, pp. 258–263, IEEE and ACM, November/December 1995.

[24] C. C. Foster and E. M. Riseman, "Percolation of Code to Enhance Parallel Dispatching and Execution," *IEEE Transactions on Computers*, vol. C-21, pp. 1411–1415, December 1972.

[25] M. D. Smith, M. S. Lam, and M. A. Horowitz, "Boosting Beyond Static Scheduling in a Superscalar Processor," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 344–354, IEEE and ACM, May 1990.

[26] E. Rotenberg and J. Smith, “Control Independence in Trace Processors,” in *Proceedings of the 32nd International Symposium on Microarchitecture (MICRO-32)*, IEEE and ACM, November 1999.

[27] G. S. Sohi, S. Breach, and T. N. Vijaykumar, “Multiscalar Processors,” in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, IEEE and ACM, June 1995.

[28] A. K. Uht, “An Efficient Hardware Algorithm to Extract Concurrency From General-Purpose Code,” in *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, University of Hawaii, in cooperation with the ACM and the IEEE Computer Society, January 1986.

[29] A. K. Uht, “A Theory of Reduced and Minimal Procedural Dependencies,” *IEEE Transactions on Computers*, vol. 40, pp. 681–692, June 1991.

[30] A. K. Uht, “Concurrency Extraction via Hardware Methods Executing the Static Instruction Stream,” *IEEE Transactions on Computers*, vol. 41, pp. 826–841, July 1992.

[31] A. K. Uht and V. Sindagi, “Disjoint Eager Execution: An Optimal Form of Speculative Execution,” in *Proceedings of the 28th International Symposium on Microarchitecture (MICRO-28)*, pp. 313–325, IEEE and ACM, November/December 1995.

[32] (Authors names omitted for anonymity), “Levo: A Resource-Flow Computer,” in *The 28th International Symposium on Computer Architecture*, IEEE and ACM, June/July 2001. Submitted.