Reducing the Ill Effects of Branches

subtitle: **- A Tutorial on Instruction Level Parallelism Enhancements -**

- To appear in: IEEE COMPUTER -

Augustus K. Uht, University of Rhode Island
Vijay Sindagi, Texas Instruments
Sajee Somanathan, ADE Corporation

April 26, 1996

Abstract

In order to significantly improve processor performance, the parallelism among machine instructions must be exploited. Conditional branches are the major restrictors of this parallelism, especially since they are widespread in general-purpose programs. Until recently, the best *Instruction Level Parallelism* (ILP) methods known realized only about a factor of 2 to 3 speedup over purely sequential computers. The uncertainties of which way branches will execute are the culprits, and are called *branch effects*. After an introduction to instruction level parallelism, current *Branch Effect Reduction Techniques* (BERT's) are described and compared, both qualitatively and quantitatively. Lastly, a new BERT developed by the authors, called *Disjoint Eager Execution* (DEE), is described and evaluated. Simulation results indicating potential order-of-magnitude speedups for one type of DEE are presented.

Keywords: high performance computing, instruction level parallelism, branches, branch effects, control flow, branch prediction, speculative execution, disjoint eager execution, control dependencies, superscalar processors.

Introduction:

There is an insatiable demand in the world for computers of ever-increasing performance. Old applications applied to more complex data and new, previously unknown applications demand the higher performance. Multiprocessor, multithreaded, vector and data-flow computers have been devised to improve performance, but they require reprogramming or recompilation, and are primarily useful for executing programs having much data parallelism or independent control flow.

General-purpose code does not fall into this category; rather, it is user or system code that is branch intensive with less data parallelism. The branches are relatively unpredictable, depend on runtime data, and the control flow is irregular. These code characteristics and their detrimental consequences are called *branch effects*, and traditionally have severely limited the parallelism obtainable from the code.

However, new *Branch Effect Reduction Techniques* (BERT's) are changing this. Order-of-magnitude or more performance speedups from parallelism are being demonstrated in research simulations, and may be realized in hardware within a few years. This article focuses on the reduction of branch effects in general-purpose code execution.

Background:

Computer performance can be improved in several ways. One of the most important is the enhancement of the internal hardware organization or *architecture* of a computer; this alone has accounted for about half of processor performance improvement over time. This fraction is expected to stay the same, if not increase.

Within the last 5 years, single-instruction-issue pipelined processors have "topped-out" their performance, executing slightly less than one instruction per cycle. Improving upon this requires that the *Instruction Level Parallelism* (ILP) implicit in programs be exploited. Indeed, *superscalar* processors doing this have become commonplace, e.g., the Intel Pentium and Motorola 68060. However, their ILP exploitation has begun to stagnate at about 2-3 instructions per cycle, on average during execution. This stagnation is due to the greatest impediment to ILP, *branches.*

Commonly, programs are composed of two kinds of instructions: *assignments* (e.g., `A=B+C`), and *branches* (used to realize high-level control flow statements such as: `if (a <= b) {....}`, or: `for (i=1; i<=10; i++) {....}`). In many cases nominally sequential instructions, for example: `A=B+C` and `D=E+F`, are *independent* (see the dependencies sidebar) and thus may be executed at the same time (in parallel). The performance improvement due to such ILP, or the ILP *speedup* S, is defined to be: *(time to execute a program sequentially)/(time to execute the program in parallel).* In the example code, then, $S = 2/1 = 2$.

In the classical model of standard or *imperative* computer operation the code after a branch cannot be executed until the branch has executed, and the program counter has been updated, restricting ILP. The code between executed branches, including the terminal or end branch, is called a *branch path.* The typical size of a branch path, the *Branch Path Length* (BPL), is 3 to 9 instructions. The ILP within a branch path has been found[2,3] to be only about a factor of 1.6, due to the presence of data dependencies.

On the face of it, then, we are stuck, and cannot create processors that execute more than one or two machine instructions per cycle. However, if branch effects could be completely eliminated, improvements in performance of factors of 25 to 158 times over sequential operation might be obtained.[2,4] Fortunately, *Branch Effect Reduction Techniques,* or *BERT's*, have been devised that allow much more ILP to be exploited. The hardware required for large-scale ILP is great, but the density of transistors per chip is increasing exponentially, estimated to reach 50-100 million transistors per chip by the year 2000.

In this article currently employed or researched BERT's are presented, discussed and compared, including examples from commercial machines. The final section presents one of our own BERT's. Simulation results of our methods indicate that order-of-magnitude ILP speedups (execution of 10 or more machine instructions, on average, per cycle) of general-purpose programs may be possible in turn-of-the-millennium processors.

Branch and Other Nomenclature:

The semantics of typical *conditional* branches are that if some condition is *true*, then control transfers to the instruction at the branch's *target* address. In this case the branch is said to be "Taken", and its branch *sign* is T or 1. If the condition is not true or *false*, then execution continues with the instruction immediately following the branch; in this case the branch is "Not taken" (sign is N or 0).

The sign of branches is often predicted in BERT's (*branch prediction*). A predictor's *Branch Prediction Accuracy* (BPA) is its average fraction of correct predictions. The path predicted to be followed is called the *Predicted* path, while the path predicted not to be followed is the *Not-Predicted* path. Note that the predicted path can be of either branch sign. Code in the CPU that is in the process of being executed is in the processor's execution *window*. A branch is commonly predicted after entering the window; it is then *pending*. When the branch's condition is evaluated, the branch is *resolved*, and may then be executed.

The *resource* usage of a *branch path* consists of the hardware needed to execute the code in the path as fast as possible, i.e., exploiting as much ILP as exists in the path. Typically, the hardware required includes 1 or 2 processing elements on average per branch path.

Static code is code in the order written by the programmer. *Dynamic* code consists of the different instances of static instructions arising during execution, ordered by their time of execution.

SIDEBAR: Dependencies, limiters of ILP:

Two instructions <u>must</u> be executed sequentially if there are one or more *dependencies* between them. A *resource* dependency arises if there are insufficient resources, such as adders, to execute all possible pending instructions at the same time.

The other class of dependencies is *semantic* dependencies. These are relationships between instructions which require the instructions to execute sequentially to ensure correct program results. Within this class, there are two categories of dependencies, *data* and *control* (or "procedural"). Both categories consist of a set of classical dependency types which restrict ILP. Research has determined, for both, a smaller and <u>minimal</u> subset of dependency types, allowing more ILP.

The classical data dependencies are shown in Table 1. In each case, the common use of memory or register variable `A` in instructions `1` and `2` creates the corresponding type of dependency. The set of minimal

Dependency name:	*flow* or *true*	*anti-*	*output*
Alternate (*hazard*) name:	(*read-after-write*)	(*write-after-read*)	(*write-after-write*)
Example:	`1. A = b + c`	`1. z = A + c`	`1. A = b + c`
	`2. z = A * y`	`2. A = y * x`	`2. A = z * y`

Table 1: *Data dependencies.* The common use of the variable `A` in each pair of instructions creates the corresponding type of dependency between the two instructions of each pair.

data dependencies consists of only the *true* dependency. The other two types of data dependencies can be eliminated by having multiple copies of instruction sinks, e.g., `A`. This is called *renaming*, and may be accomplished in either hardware or software. We assume that renaming is used throughout this article.

Classically, all instructions dynamically after a branch are control dependent on the branch. Therefore, in the example below, instructions `2` through `7` are all dependent on instruction `1`, a branch.

```
1:  if (a == b) {      //(or, with a branch format: if (a != b) goto 3;)
2:      z = y + x; }
3:  d = e * f;
4:  g = d - h;
5:  if (x == y) {      //(or: if (x != y) goto 7;)
6:      u = y + e; }
7:  j = k - m;
```

With *Minimal Control Dependencies* (MCD),[5] however, more ILP is possible. Now the execution of instructions `3-7` does not depend on whether instruction `1` is taken or not taken. Thus, instructions `3-7` can execute concurrently with instruction `1`, increasing the realizable ILP. Note that this results in the semantically-allowable parallel execution of branches. The set of necessary and sufficient control dependencies is given elsewhere.[5] All commercial and most research processors use the classical model of control dependencies, since it is cheap. However, the minimal control dependency model is necessary to realize significant ILP.[4]

BERT's:

The BERT's are summarized and compared in Table 2. There and elsewhere, we have used data obtained with the *de facto* standard general-purpose benchmarks, the SPECint92 suite; if those data are not available, we have used data closest to them. In the Table, BERT performance is given in terms of ILP speedup or BPA. Hardware attributes include typical hardware cost, whether the instruction set is impacted, and a ranking of the BERTs' hardware costs. Software characteristics include whether object code size increases, whether program profiling or recompilation are necessary, and a ranking of the BERTs' software complexities.

The overall BERT goal is to reduce the ill effects of branches in order to expose as much inherent ILP as possible. There are three general devices modern BERT's use to achieve this end. BERT's within and among different device types may be combined, often to great advantage.

BERT Devices: The first device is to conditionally execute code after a branch, before the branch has resolved, even if the code is control dependent on the branch. Hence, such code is executed before it is known whether it should be executed; this is *speculative execution*; typically, *branch prediction* is used. We call speculative execution methods that only execute down one path from a branch *Single-Path* (SP) speculative execution. Considering SP execution, if a prediction is correct, the code available for ILP extraction increases beyond a branch path; the available code is then that between branch *mispredictions*. Improving the *Branch Prediction Accuracy* (BPA) increases the distance between mispredictions, increasing ILP. Improved speculative execution methods seek to increase the effective distance between mispredictions.

The second BERT device decreases the effect or range of branches. This is done via the use of *Minimal Control Dependencies* (MCD) (see the "Dependencies" sidebar), or *predication*, e.g., the HEP and Cydra 5 machines.[6]

The last general category of device is to increase the size of the average basic block, and thereby increase its ILP. This is done via compiler-based methods, e.g., code *percolation* or *motion*, or *trace scheduling*.[7]

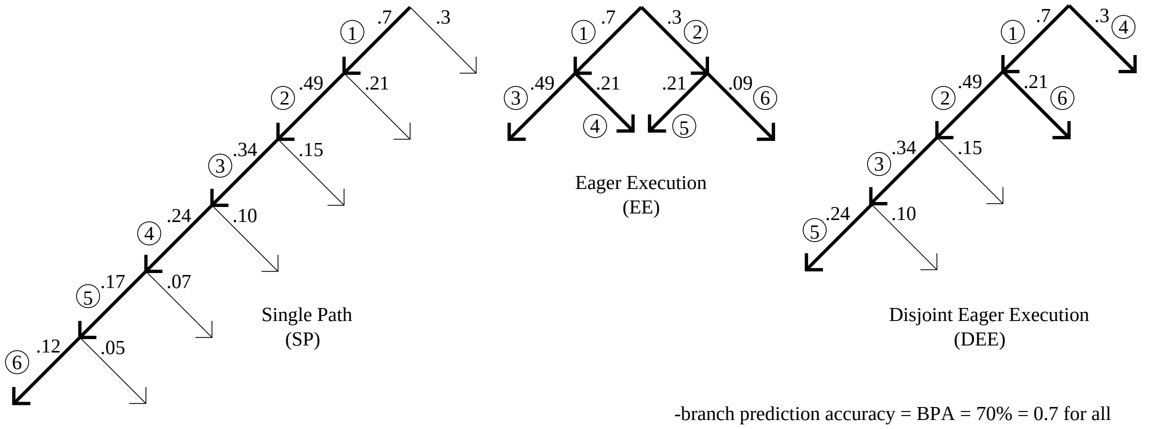
BERT's Conditionally Executing Past an Unresolved Branch:

We first describe different forms of speculative execution, and then discuss various branch prediction techniques that can be used by these forms. Speculative execution can be realized in hardware or software. (It can also be used in other kinds of computers, e.g., multiprocessors, data flow machines, etc.) What is commonly referred to today as "speculative execution" is usually single-path speculative execution.

Forms of Speculative Execution: Until recently, only two forms of speculative execution were known: the *Single-Path* (SP) type and *Eager Execution* (EE). A third, optimal form has recently been devised by Uht;[1] it is called *Disjoint Eager Execution* (DEE). Schematic examples of these three forms of speculative execution are shown in Figure 1; the paths shown indicate code corresponding to pending branches.

In single-path execution, when the CPU encounters a branch, the sign of the branch is predicted, and execution proceeds down the predicted path. Since the branch is unresolved, all writes to architectural registers or memory, and all I/O operations, must be made conditionally, and only finalized or committed if all previously speculated branches are predicted correctly. If there is a misprediction before a conditional operation, then that operation must not be committed, but *squashed*. The hardware cost of SP is low, being linear in the number of branches pending. However, the usefulness of increasing predictions becomes negligible quite rapidly. SP is the speculative execution form of choice for current microprocessors.

With the eager execution model, execution proceeds down both paths of a branch; no prediction is made. Similar to SP, when a branch resolves, all of the state on the not-to-be-followed paths is discarded. EE



-each line segment with arrow represents a branch path
-left-pointing paths are Predicted paths
-right-pointing paths are Not-Predicted paths
-an uncircled number on a path is the overall or Cumulative Probability of the path being executed (cp)

-branch prediction accuracy = BPA = 70% = 0.7 for all branches, for the sake of illustration
-bold lines indicate loci of execution (code in the window)
-circled path numbers indicate order of resource assignment
-overall resources fixed at 6 branch paths
-all of the branches are pending

Figure 1: *Comparison of the three speculative execution strategies.* In each strategy, resources are assigned only to the bold paths. Note that the cumulative probabilities of the DEE bold paths are greater than the corresponding ones of both single path execution and eager execution: in DEE, the most likely code to be executed is given the resources.

with unlimited resources gives the best performance, but is unrealizable. It is the same as *oracle* execution (BPA=100%): branch effects do not occur. With constrained resources, EE does not perform very well.[2] Maintaining the paths' state is hard. Also, EE's hardware cost rises exponentially with the levels of pending branches; thus, it is not often used. When it is, it is in a limited fashion for, say, instruction fetch and decode, but not execute; examples are the Sun SuperSPARC and the IBM 360/91.

The disjoint eager execution model performs better than both SP and EE given the same branch path resources. DEE is optimal, in that for fixed, limited resources, it provides the best possible performance of any speculative execution method. The key idea of DEE is to assign resources to those branch paths most likely to execute over all unexecuted branches. That is, the overall likelihood or *cumulative probability* of execution (cp) for each branch path is used to make the resource assignment. In DEE, all branches are predicted, while some of these are also eagerly executed. The cost of DEE increases as a fraction of the square of the length (l) of the Main-Line path (see Figure 5), much less than the cost of EE, and close to that of SP.

General Branch Prediction: Since branch prediction is key to most speculative execution techniques, and hence to ILP extraction and CPU performance, we now discuss the known prediction methods in some depth. We first discuss introductory material on both the *Branch Target Buffer*, commonly used in hardware branch handling and prediction, and on *Branch Predictor Accuracy* (BPA).

The Branch Target Buffer is a form of cache, holding the target addresses of recently executed branches. In typical operation, when a branch is first executed taken, its target address must be computed before the branch can actually be taken; this initial computation slows down the branch's execution. However, as a side effect of this initial execution the branch's target address is saved in the buffer; therefore, on subsequent executions of the branch the target address is immediately available from the buffer, eliminating the time penalty. This buffer is commonly used, regardless of other BERT's used. The buffer can also hold various branch prediction information.

The amount of ILP realized by a BERT is very sensitive to its predictor's BPA. A small absolute increase in BPA can result in a large increase in the ILP exploited. ILP increases with increasing distance between mispredictions, d_{mp}. This distance is proportional to the inverse of the prediction *in*accuracy: $d_{mp} = BPL/(1 - BPA)$, where BPL is the Branch Path Length. For example, improving BPA from just 85% to 90% increases d_{mp} by 50%.

Static Prediction: This BERT operates by making predictions that are hardwired. Commonly, such predictors assume that branches are executed all not taken, e.g., the Intel i486; or all taken, e.g., the Sun SuperSparc. They cost practically nothing, but their accuracies are not large, being about 40-60%.

More involved, but still cheap, methods utilize typical characteristics of branches, such as the branch *direction*: forwards or backwards. BTFN predicts that all Backward branches are Taken (typically true 90% of the time, as they often form loops), and all Forward branches are Not taken (typically true only 50% of the time). The predictor takes advantage of backward branches' predilection to be taken, improving the BPA to 65%; BTFN is used in the HP PA-7x00 processors.

Semi-Static Prediction: Again, predictions are constant over the execution of a program, but they vary from static branch to static branch, and are made by the compiler and included in the actual machine instructions. Thus, if this method is added to an existing CPU, its instruction set must be modified.

The compiler makes its predictions using program *profile* statistics, obtained by compiling the program once, and then running it on test data while counting the times a branch is taken vs. times not taken. Lastly, the program is recompiled, using the statistics to set the object code's branches' prediction bits accordingly. This has limitations, as the statistics and hence predictions can vary from the test data to the actual data. Allowing predictions to vary from branch to branch primarily improves the prediction accuracies of forward branches; a typical forward branch executes predominantly with one sign. Therefore the BPA improves, being about 75%. This method is used in many of the PowerPC processors.

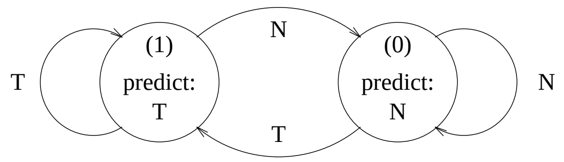


Figure 2: *One-bit branch predictor.* A state transition occurs when a branch resolves. The transition is determined by the resultant sign of the resolving branch. The output of the predictor is indicated within each state: "predict: T(aken)" or "predict: N(ot taken)". Ideally, there is one automata per static branch.

Dynamic Prediction: Now the predictions can vary over the execution of the program, adapting to the input data. A branch may execute consistently one way in one part of the program's execution, and the other way in another part of the execution, and a dynamic method will be able to adapt to the change and continue to make accurate predictions; a semi-static predictor in a similar situation would give wrong predictions much of the time. No profiling is needed; dynamic prediction can be accomplished entirely in hardware.

A simple dynamic branch prediction algorithm predicts that a branch will execute next the same way. This is shown by the 2-state automata in Figure 2. The automata goes to the (1) or (0) states if a branch is actually taken or not taken, respectively. The automata can be realized implicitly with a branch target buffer. The automata is in state (1) if the buffer contains an entry for the branch: the branch was taken when last executed. Thus, the branch is to be predicted taken when next encountered; after the next instance of the branch is fetched, instruction fetching continues with the instruction at the target address. The automata is in state (0) if there is no entry in the buffer: the branch is predicted not taken, and fetching continues with the instruction statically after the branch. The automata can also be realized explicitly with 1 bit of storage, and thus is referred to as a "1-bit" predictor; it is fairly accurate, having a BPA of 77-79%. The DEC Alpha 21064 processor uses this predictor, holding the state for up to 2K automata.

A better performing, though costlier, predictor is the 2-bit saturating up/down counter of Smith;[8] see Figure 3. In real machines it exhibits accuracies of 78 to 89%. Each 2-bit automata's state is stored in a branch target buffer. A branch is predicted by reading the buffer and using the state of the automata. In operation, branches that are more often taken are predicted taken, and likewise for not taken branches; i.e., a form of "averaging" occurs. Thus, this predictor is less adversely impacted by occasional changes in branch sign than the one-bit predictor. For example, in the stream of branch executions: ...NNNTNNN..., the 1-bit predictor gives two mispredictions, while the 2-bit predictor has only one. However, there are simple situations that can cause this predictor to predict wrong 100% of the time! (For example, starting from state 01, the alternating sequence TNTNTN... causes this phenomenon.) This predictor is the most popular one used in current high-end microprocessors, such as the NexGen 586 (2K automata) and the Intel

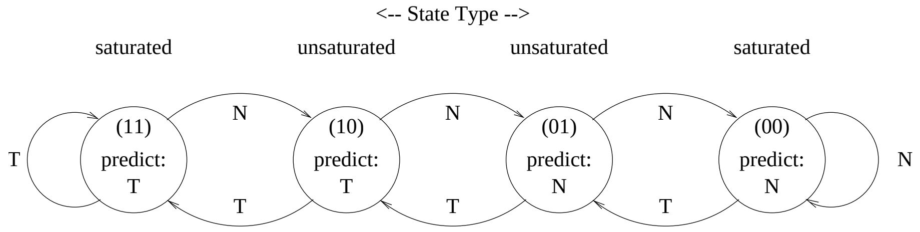


Figure 3: *Two-bit branch predictor.* As with the one-bit predictor, a resolving branch causes a state transition. The most significant bit of the state indicates the prediction of the sign of the next instance of the same static branch.

Pentium (256 automata). 3- and greater-bit automata are also possible, but can actually decrease the BPA.[8]

In 1991, Yeh and Patt at the University of Michigan, and independently Pan *et al* of IBM and the University of Texas in 1992, devised a predictor which is significantly more accurate; it is called a "2-level adaptive" or "branch correlation" predictor. Predictions are based on the extended *history* of branch signs, associating separate 2-bit counters with different histories. Thus, this predictor is able to detect many more varied branch execution sequences and treat them individually, enhancing the overall BPA.

As the first name implies, branch prediction is now a two-step operation, with two corresponding structures. The *Branch History Register* holds the branch execution history; see the upper-lefthand corner of Figure 4. Every time a dynamic instance of any branch resolves, its sign is shifted into the register. The register helps in the prediction process by capturing much longer and more varied patterns of branch executions than a 2-bit predictor alone. In the second structure, there is a 2-bit counter automata for each possible pattern of the branch history register. Hence, with a 4-bit register, there are $2^4 = 16$ possible 2-bit automata, grouped into a *Branch Pattern Table.* This is accessed using the contents of the branch history register as the table's address; see the lower-righthand corner of Figure 4. The state of the indexed automata in the table indicates the prediction, as in Figure 3. The indexed automata is also updated similarly.

Multiple branch history registers and branch pattern tables are possible, for example one register and table per static branch. Altogether, there are nine possible configurations of the 2-level adaptive method, exhibiting various cost/accuracy tradeoffs.[9] The BPA with one 8-bit history register and 16 pattern tables, for example, has been found to be 94.2% for the SPECint89 benchmarks.

The adaptive method is more accurate than the 2-bit counter predictor since it bases its predictions on specific branch histories, not a general averaging. For example, again consider the branch sign sequence: ...TNTNTN...; as discussed, the 2-bit predictor can mispredict 100% of the time. However, the adaptive predictor recognizes the specific branch execution history, and makes <u>correct</u> predictions 100% of the time.

The 2-level adaptive predictor is used in the NexGen Nx686. A single 9-bit history register is concatenated

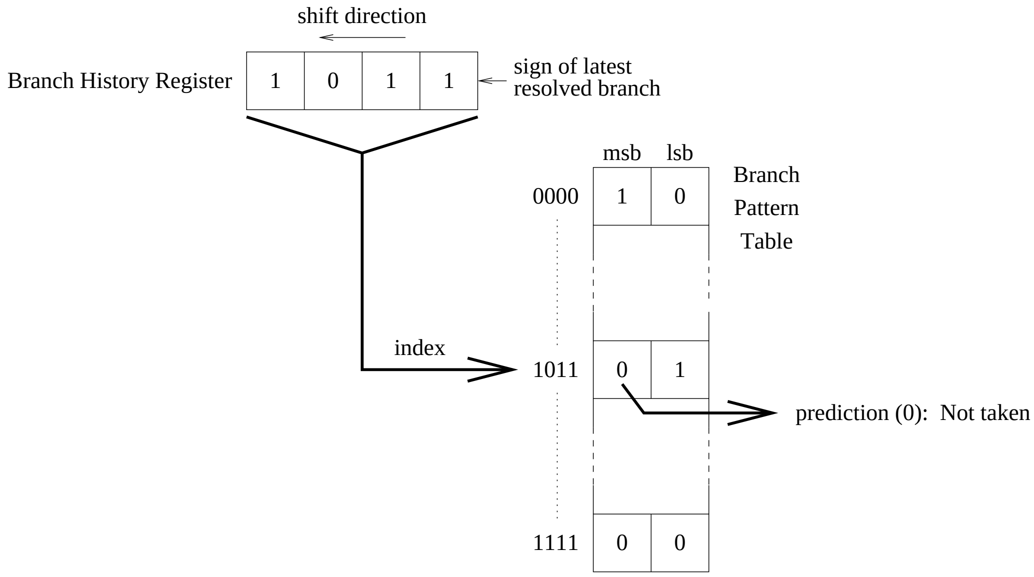


Figure 4: *2-level adaptive branch predictor.* Predictions now take two steps. Each entry (row) of the Branch Pattern Table is the equivalent of the 2-bit counter of Figure 3. The History Register holds the signs of past branch executions. This recent history is used to index to a particular automata in the Pattern Table. Thus, different automata are used for different patterns of branch executions; the predictor thereby adapts to different sequences of branches' dynamic executions, improving the prediction accuracy. With a single History Register the predictor combines information from multiple branches, allowing the correlation among different static branches to be exploited. (The index can also be formed by concatenating or hashing part of the program counter, which identifies the static branch, with the Register, slightly improving the BPA.)

with 4-bits of the program counter to index a pattern table of 8K 2-bit counters. An undisclosed version of the 2-level method is also used in the Intel Pentium Pro (P6).

Hybrid Prediction:[10] This method combines the static profiling of semi-static prediction with dynamic branch correlation. As with semi-static prediction, each static branch contains a prediction bit. The profiling phase, however, is much more involved. The different instruction sequences occurring during program execution up to each static branch (Y) are recorded, indicating how dynamic branches prior to Y executed. The object code is then modified to allow different static predictions of Y to be used at run-time. This is done by copying Y and its branch path for each unique sequence through Y, with each Y-copy's prediction bit set according to the likely sign of Y for the corresponding sequence.

Thus, Y is effectively predicted at run-time according to the program's actual control-flow, but not directly on Y's dynamic execution history. Since Y's predictions can vary dynamically, and adapt to some of the program dynamics, hybrid prediction performs better than semi-static prediction, having a BPA of about 88%. However, it does not adapt as well as hardware-based 2-level adaptive methods, since the latter also take into account Y's history. Another disadvantage of the hybrid prediction method is that object code size may increase significantly, due to the code replication. However, no predictor hardware is needed.

Advanced Dynamic Predictors: A variation of the 2-level adaptive predictor combines the scheme with profiling information on the degree of predictability of the branches to better allocate predictor resources. Another method employs two different predictors at once; a separate "predictor-predictor", e.g., a separate set of 2-bit saturating counters, is used to select the best of the two main predictors to predict a given branch. These methods are more sophisticated but only improve the BPA by an absolute increment of about 1%, if both are used together. We may be nearing another "topped-out" point, this time of predictor accuracy.

Another Prediction Dimension: A key implementation parameter of branch prediction systems is the number of predictions that may be made simultaneously. Multiple predictions per cycle are necessary to realize high ILP speedups, due to the high frequency of dynamic branches in application code. Most machines today only allow one branch to be predicted at a time; the Levo model[1] is an exception, allowing up to 32 predictions per cycle. P. G. Emma *et al* of IBM Yorktown Heights have also devised hardware allowing many predictions per cycle.

Branch Predictor Summary: Static methods are not that accurate, but use little hardware. At the other extreme, the 2-level adaptive methods are very accurate, but are costly. As hardware densities improve, and more accurate predictors are required, adaptive methods will become more attractive.

High-Level Branch Prediction: *Multiscalar* machines are being studied at the University of Wisconsin. In this model the compiler divides a program into smaller blocks of computation called *tasks*. The dynamic task sequence is predicted by the compiler; therefore, SP speculative execution is performed at the task level. A key feature is that task sequence prediction is performed independently of branch predictions within the tasks. The task graph is included in the object code. Each task is executed by a typical (ILP or superscalar) processing engine. The engines in a multiscalar machine attempt to execute tasks concurrently, improving performance. With this model, the instruction set needs to be modified, and recompilation is necessary. This is a new research area, and thus the ILP realized may improve significantly with further research. Current unsolved multiscalar problems include: keeping all of the engines busy all of the time (load balancing); and making accurate task sequence predictions.

BERT's Decreasing the Effect or Range of Branches:

Minimal Control Dependencies (MCD): This BERT allows the execution of both assignment and branch instructions absolutely, not speculatively, in parallel with earlier branches (see the "dependencies" sidebar). To our knowledge MCD has only been hardware-implemented in the CONDEL research machines' simulators.[5] By itself, MCD gives modest gains, but when combined with SP or DEE, large gains result, since MCD decreases the impact of mispredictions compared with the use of classical control dependencies. A misprediction with the classical model squashes <u>all</u> instructions that follow the mispredicted branch, whereas with MCD only the many fewer MCD-dependent instructions are invalidated.

Predication:[6] This BERT is currently receiving much attention. As shown below, the Boolean results of condition testing are put into predicate registers, and branches are eliminated. Instructions which were dependent on a branch now instead have a one-bit predicate register as input. An instruction executes only if its predicate is true. The advantage of predication is that it eliminates some branches and their associated control dependencies, nominally improving performance.

```
        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.
```

Predication is equivalent to a limited form of minimal control dependencies, but impacts the instruction set of a computer, adds a port to the CPU register file, and complicates instruction execution. Alone, it does not realize much ILP; sophisticated compilers are also needed to obtain significant ILP. Although some branches are eliminated with predication, increasing the distance between mispredictions, the actual overall dependencies in the code are not reduced below those occurring with minimal control dependencies, since the control dependencies removed by predication become data dependencies. Thus, the BPA is artificially raised and cannot be compared to plain branch predictors'. Predication may prove more efficacious with further research.

BERT's to Increase Basic Block Size:

The best known such BERT is compiler-based *trace-scheduling*, used in most VLIW (Very Long Instruction Word) computers, such as the Multiflow Trace.[7] Trace-scheduling unrolls loops, increasing the basic block size. Profiling is used to estimate the most likely path or *trace* through the code. Machine operations in this trace which can be executed in parallel are grouped into "Very Long..." machine instructions. Typically, each instruction includes both assignment operations and a multi-way branch. Since the operations within a VLIW instruction are independent, the hardware need have no scheduling or dependency-checking hardware. However, recompilation is necessary whenever the processor is replaced. Since recompilation is not always possible, VLIW machines currently have a limited appeal. Also, VLIW machines do not typically take advantage of the dynamics of the code well. Trace-scheduling often greatly increases the size of a program's object code. Further, it is not clear that VLIW machines can use MCD to execute multiple multiway branches in parallel; it is a difficult unsolved problem. However, there are significant efforts underway by VLIW researchers to solve some of these problems. In fact, there is an alliance between Intel and HP to create a new VLIW-based microprocessor, code-named "Tahoe" or "P8", having both x86 and HP-Precision Architecture compatibility.

The minimal control dependency model is also used in some software-based BERT's, such as *software-pipelining*, a VLIW variant. This BERT has standard VLIW advantages and disadvantages. In limited cases, it can achieve the equivalent of eager execution performance, with less hardware.

Other BERT's such as *boosting* or *spreading* use software. These employ some VLIW methods, such as *code motion*, in which instructions are moved at compile-time to enhance the net ILP realized in the target machine. The ILP speedup realized with VLIW or VLIW-aided methods is typically less than 3.

Caveat: When changing processors, it is often costly or impractical to update the machine code, as re-compilation is not possible. Therefore users tend to stay with the same *instruction set architecture*, e.g., IBM 360 or Intel x86. This limits the likely usage of those BERT's requiring changes to existing instruction sets; however, this constraint may not always apply (e.g., to embedded machines).

	Characteristics[a]									
	Performance			Hardware			Software			
BERT	ILP speedup realized[b,e]	BPA (research)[b] (%)	BPA (real)[b] (%)	real machine hardware[b,d]	instruction set modified?	hardware cost[c]	code size increase?[c]	profiling necessary?	recompilation necessary?[f]	software complexity[c]
Single-Path with the specified branch predictor, given in the order appearing in the text:										
none	1.6	-	-	none	no	0	no	no	no	0
always Not Taken	NA	NA	40	little	no	1	no	no	no	0
always Taken	NA	61-75	60	little	no	1	no	no	no	0
BTFN	NA	NA	65	little	no	2	no	no	no	0
semi-static (profiling)	1.5	83.4	75	1 OP-Code bit	yes	3	no	yes	no	2
dynamic (1-bit)	NA	90	77-79	1K-2K entries	no	4	no	no	no	0
dynamic (2-bit)	2.9	90-92.2	78-89	256-4K entries	no	5	no	no	no	0
dynamic (adaptive)	3.6	90-96	90-94	$\geq$2K bits	no	6	no	no	no	0
hybrid	NA	88	-	1 OP-Code bit	yes	4	yes; 2	yes	no	3
Other BERT's, using various branch predictors, in increasing ILP speedup order:										
predication (alone)	1.6	-	-	sev. OP-Code bits	yes	7	no	no	no	0
MCD (hard.-based)[g]	2.1	-	-	-	no	7	yes; 1	no	no	1
multiscalar	2.6	-	-	sev. OP-Code bits	yes	8	yes; 1	no	yes	3
predication (w/soft.)	3.0	-	-	sev. OP-Code bits	yes	7	yes; 1	yes	no	4
VLIW-based	3.6	-	-	none	yes	0	yes; 3	yes	yes	4
Eager Execution (EE)	7.6	-	-	1 branch bypassed	no	9	no	no	no	0
Disjoint EE (DEE)[h]	8.5	-	-	-	no	7	no	no	no	0
DEE-MCD[g,h]	31.9	-	-	-	no	8	yes; 1	no	no	1

[a] Space and style constraints prohibit citing all of the sources of the BERT data in this Table. However, these references may be found in Technical Report No. 0496-0001 of the Dept. of Electrical and Computer Engineering, the University of Rhode Island; this is available on the World Wide Web at URL: `http://www.ele.uri.edu/faculty/uht/papers.html`. "NA" is: Not Available. "-" is: not relevant, or has not been used in a real machine.

[b] These data are from various sources, most included in the references. Many are from Microprocessor Report newsletters.[11] The BPA (research) figures are from studies looking at predictors by themselves, whereas the BPA (real) numbers include hardware and code limitations; most are based on real computers. When possible, the BPA and ILP data assume the SPECint92 benchmark suite or a portion thereof.

[c] The numbers are **order** rankings, worse numbers are higher; they do not indicate anything in absolute terms, nor can numbers from different columns be compared by their values in "better" or "worse" terms. The numbers are the authors' subjective opinions only.

[d] OP-Code bit(s): The BERT requires one or more ("sev.") additional bits in either branch or assignment instructions. An "entry" is the state (memory bits) for one automata.

[e] The numbers are speedup factors (i.e., in 100's of percent). These data are from research machine simulations. Each datum was computed as the harmonic mean of the individual benchmarks' speedups. Only the Single-Path "none" data assume unrestricted resources. Note that a BERT's actual operation and its simulated operation differ in varying degrees from BERT to BERT, and from study to study.

[f] ...upon machine upgrade or downgrade.

[g] These BERT's work best with some binary-to-binary code translation, via a *filter* program run once for an application program. The main action performed is to do limited loop unrolling. The filter is not required for correct code execution.

[h] The data given assume hardware-based implementations; 2-bit dynamic branch prediction is used. Software-based implementations are also possible.

Table 2: *Comparison of the BERT's.* The BERT's are listed in the leftmost column, with their absolute and relative characteristics given in the columns to the right. The characteristics are grouped into categories of performance realized, hardware attributes and software attributes.

Disjoint Eager Execution - Practicalities, Simulations and Realization:

A Practical DEE: Recall that DEE works by assigning resources only to the most likely paths to be executed. Determining these at run-time, with branches having different and time-varying BPA's, is very hardware-intensive and difficult; it is not even theoretically possible to do this "exactly".[1]

However, a heuristic proposed by Uht[1] works well. A logical "static tree" of resources is created, as shown in Figure 5. Its shape is determined when a uniprocessor is designed, by assuming that the predictor exhibits the same BPA on every branch. With a constant BPA, the tree has the basic shape shown in Figure 5, consisting of a relatively long "Main-Line" region with length l, and a relatively small number of side paths composing the DEE region. It can be shown that for a constant BPA the shape of the DEE region is of the lower right half of a square having side dimensions of $h_{DEE} = w_{DEE}$.

From geometric analysis the relationship between BPA, h_{DEE} and E_T (the total number of branch paths in the tree) is seen to be:

$$E_T = \log_{BPA}(1 - BPA) + \frac{1}{2}h_{DEE}^2 + \frac{3}{2}h_{DEE} - 1$$

Using the quadratic formula, this equation can be solved for h_{DEE}. The Main-Line length l is given by:

$$l = h_{DEE} + \log_{BPA}(1 - BPA) - 1$$

Therefore, to determine the shape of the static tree at machine design time, the designer first determines the likely value of the BPA of the branch predictor to be used; this is readily done by analyzing execution traces of representative target application and operating system codes. The designer also determines the total resources (cost) allowable for the machine, giving E_T. These values of BPA and E_T are then plugged into the above equations, giving the dimensions of the static tree. Of course, detailed study would then be used to fine tune the tree dimensions.

In operation, at run-time the actual control-flow maps onto the tree, and the tree in effect moves down the dynamic trace of execution as branches resolve at the tree's root. Code executes only when in the tree; the tree is the CPU's window.

Simulation Environment: The DEE and DEE-MCD (DEE with minimal control dependencies) models were simulated, both using the static tree heuristic. SP, EE, SP-MCD (SP with minimal control dependencies) and Oracle models were also simulated, for comparison. A modified version of Lam and Wilson's `dsim` simulator[4] was used for the experiments. The simulator operates on MIPS R3000 machine code, but

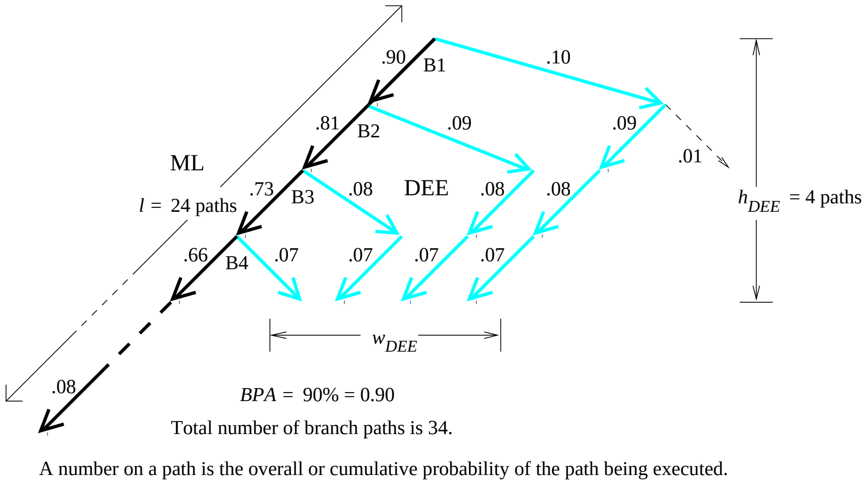


Figure 5: *Typical static DEE assignment tree.* ML is the *Main-Line path* or *region.* The execution of the ML path can be realized via SP execution. Each *composite DEE path* to the side (blue or shaded paths; each 1 to 4 branch paths long, in this example) can be treated with SP execution within itself. All of the composite DEE paths together form the *DEE region.* This model of execution can be employed within or among uniprocessors to realize DEE.

assumes every instruction executes in one cycle. The Smith 2-bit dynamic branch predictor was used. Five of the six SPECint92 benchmarks were simulated[1] (`sc` was not, since it was much more predictable), each one traced for up to 100 million machine instructions. All of the models used the number of branch path resources allowed (E_T) as the independent variable. This number and the geometric mean of the SPECint92 benchmarks' BPA's, 90.53%, determined the shape of the simulated static trees.

Experimental Results: The results of the simulations are shown in Figure 6. In the graph, the X-axis is the number of branch path resources in a simulated ILP model. The Y-axis is the speedup factor of an ILP model with respect to a strictly sequential model, computed as the harmonic mean of the 5 individual benchmarks' speedups. For 100 branch paths (a possible design point for our prototype machine, Levo), the speedup of the DEE-MCD model is a factor of 31.90, or 3,190%, over that of a sequential machine[A]. This is about an order-of-magnitude better than anything appearing in the literature or commercially. It is also about a factor of 4 better than that achievable with an unlimited resource version of SP employing an adaptive predictor.[3] Also, the data indicate that Levo may be able to be built with only 32 path resources and still get most of this speedup. The DEE-MCD model exploits most (about 59%) of the instruction-level parallelism at all obtainable from these benchmarks, as determined by the Oracle simulations.

[A] The jump in performance figures from 16 to 32 paths is a result of DEE being the same as SP for 16 paths and below. This indicates that DEE paths should be used in models having fewer resources, too.

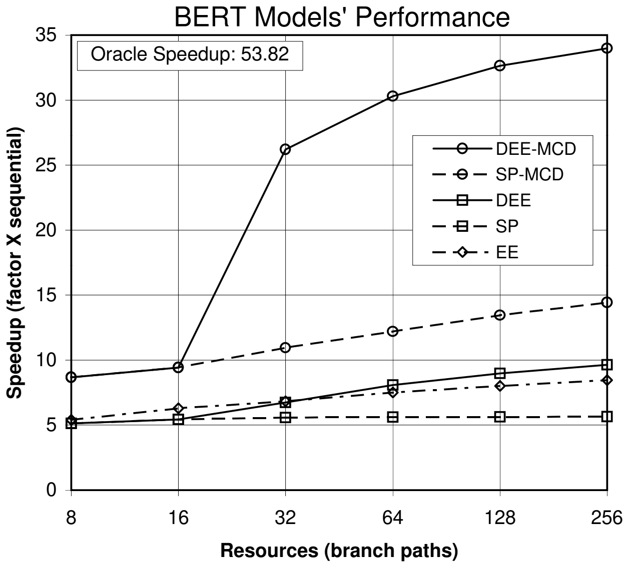


Figure 6: *DEE simulation results on the SPECint92 benchmarks (w/out* `sc`*).* The best performance occurs when DEE is combined with minimal control dependencies. In such cases most mispredictions are nullified, since most are resolved when at the root of the static tree, when many results are already available on the corresponding DEE path. The DEE-MCD results are about an order-of-magnitude better than those previously demonstrated.

Features of Levo, the Prototype Machine: Levo[1] is a prototype computer embodying the DEE-MCD ILP model being developed at the University of Rhode Island. Ten's of branches may be predicted, resolved or executed per cycle, in any way, taken or not taken. The time penalty to recover from mispredictions is 0 or 1 cycle for any type or number of mispredictions occurring in the same cycle.

The anticipated number of processing elements for Levo is 32, resulting in a target ILP speedup of 20 or more. We estimate that Levo could fit on a single chip by about the year 2000, since the current trend of chip transistor density doubling every 1.5-2.5 years is expected to continue for at least the next 10-15 years.

Summary and Conclusions:

The uncertainty of the outcome of branches has severely restricted the parallelism obtainable from typical general-purpose code. However, many effective branch effect reduction techniques can now be employed, using hardware, software or both. Software-based methods may be used when hardware cost is a prime issue, instruction set compatibility is not required, and the best performance is not desired. Otherwise, hardware methods are preferred. A wide range of branch prediction methods is available, allowing many tradeoffs. Branch prediction accuracies of up to 90-95% are realizable today. Single-path speculative execution is currently a viable option by itself. To go beyond its performance, however, other methods must be used.

A new speculative execution method, disjoint eager execution, shows great promise at the high end of the performance spectrum, potentially leading to realizable instruction-level parallelism of factors in the tens. A range of different instantiations of DEE are possible, giving another tradeoff dimension.

The commercial exploitation of ILP has just begun. New methods are needed to extract more of the potential ILP. However, with known techniques in the research labs, order-of-magnitude ILP speedups are already likely soon.

Acknowledgements and Notes:

This work was supported by the Intel Corporation, the University of Rhode Island Research Office, and the National Science Foundation through grant number: CCR-8910586. We also thank Monica Lam and Robert Wilson of Stanford University for their simulator and their assistance. Many thanks to Qing Yang and the referees for their comments on drafts of this article. Special thanks to Laurette Bradley for both her comments on article drafts, and for her constant support.

Many of the references may be found in an IEEE tutorial.[12] Other relevant readings are in the proceedings of the annual IEEE and ACM International Symposium on Microarchitecture. There is much good ILP literature, too much to be acknowledged here; our apologies to the authors of same.

References

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

[2] 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.

[3] D. Wall, "Limits of Instruction-Level Parallelism," Tech. Rep. 93/6, Digital Western Research Laboratory, Palo Alto, California, November 1993. WRL Research Report; to obtain, send: `Subject: help` to: `WRL-Techreports@pa.dec.com`.

[4] 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.

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

[6] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 Departmental Supercomputer," *COMPUTER*, vol. 22, pp. 12–35, January 1989.

[7] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Transactions on Computers*, vol. C-37, pp. 967–979, Aug. 1988.

[8] 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.

[9] T.-Y. Yeh and Y. N. Patt, "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 257–266, IEEE and ACM, May 1993.

[10] C. Young and M. D. Smith, "Improving the Accuracy of Static Branch Prediction Using Branch Correlation," in *ASPLOS VI: Proceedings of the Sixth Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 232–241, ACM, October 1994.

[11] L. Gwennap, "New Algorithm Improves Branch Prediction," *Microprocessor Report*, vol. 9, pp. 17–21, March 27, 1995.

[12] H. C. Torng and S. Vassiliadis, Eds., *Instruction-Level Parallel Processors.* Los Alamitos, California: IEEE Computer Society Press, 1995.

Biographies:

Augustus K. Uht is an Assistant Professor in the Department of Electrical and Computer Engineering at the University of Rhode Island, and is a member of its High Performance Computing Laboratory. He has conducted research on instruction level parallelism since 1982, concentrating on the enhancement of parallelism via the reduction of branch effects. He currently leads the Levo High ILP Prototype Computer project at URI. His other interests include computer architecture, memory systems, digital system design, and parallel computing.

Uht received the BS and M.Eng.(Elect.) degrees from Cornell University in 1977 and 1978, respectively, and the Ph.D. degree from Carnegie-Mellon University in 1985; all of the degrees were in electrical and computer engineering. He became a licensed Professional Engineer in 1982. Uht has served on the Program Committee of the IEEE/ACM Symposium on Microarchitecture. He worked on mainframe memory systems at IBM, East Fishkill, from 1978 to 1982. Uht was on the faculty of the Department of Computer Science and Engineering at the University of California, San Diego, from 1986 to 1992, when he joined URI. He is a member of the IEEE, the IEEE Computer Society, the ACM, and the NSPE.

Vijay Sindagi is a CAD engineer at Texas Instruments, Houston. His interests include computer architecture, compilers and ASIPs. He received the BS(EE) degree from Bangalore University, India in 1985, and the MS degree in computer engineering from the University of Rhode Island in 1995. Sindagi worked on networking and real-time systems at BEL, Bangalore, from 1986 to 1992. He is a member of the ACM.

Sajee Somanathan is a Software Engineer at the ADE Corporation in Newton, Mass. His current interests include instruction level parallelism, parallel computer architecture, branch prediction and performance evaluation. Somanathan received a B.Tech. degree from the College of Engineering, Trivandrum, India in 1988, and the MS degree in computer engineering from the University of Rhode Island in 1995.