Research Feature

# Branch Effect Reduction Techniques

Branch effects are the biggest obstacle to gaining significant speedups when running general-purpose code on instruction-level parallel machines. This survey compares current branch effect reduction techniques, offering hope for greater gains.

*Augustus K. Uht*
University of Rhode Island

*Vijay Sindagi*
Texas Instruments

*Sajee Somanathan*
ADE Corp.

There is an insatiable demand for computers of ever-increasing performance. Old applications are applied to more complex data and new applications demand improved capabilities. Developers must exploit parallelism for all types of programs to realize gains. Multiprocessor, multithreaded, vector, and dataflow computers achieve speedups up to the 1,000's for programs with large amounts of data parallelism or independent control flow. For general-purpose code, however—which comprises most executed code—parallel execution has been only two or three times faster than sequential.

General-purpose code has many conditional branches, irregular control flow, and much less data parallelism. These code characteristics and their detrimental consequences, in the form of *branch effects*, have severely limited the parallelism that can be exploited. Branch effects result from the uncertainties in the way branches execute.

In this article, we survey techniques to reduce branch effects and describe their relative merits, including examples from commercial machines. We believe this survey is timely because research is bearing much fruit: Speedups of 10 or more are being demonstrated in research simulations and may be realized in hardware within a few years. The hardware required for large-scale exploitation is great, but the density of transistors per chip is increasing exponentially, with estimates of 50 to 100 million transistors per chip by 2000.

## PERFORMANCE FACTORS

Architectural enhancements alone account for half the increase in processor performance over the years—a percentage that is expected to stay the same, if not grow. However, within the past five years, single-instruction-issue pipelined processors have topped out their performance, executing slightly less than one instruction per cycle. If designers are to continue increasing processor performance, they must turn to methods that exploit *instruction-level parallelism* within each program. Superscalar processors like the Intel Pentium and Motorola 68060 have been doing exactly this. However, performance has stalled at speedups of two to three instructions per cycle, on average. This stagnation is due to branch effects.

### Branch effects

To illustrate how branch effects can block the exploitation of instruction-level parallelism, consider the typical program, which has two kinds of instructions: assignments `(A=B+C)` and branches. Branches are 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, such as `A=B+C` and `D=E+F`, are independent and thus may be executed in parallel. The performance improvement or speedup due to this parallelism is the time to execute a program sequentially divided by the time to execute the program in parallel. In a program composed of the two instructions just given, the speedup is 2 (2/1).

Branches give rise to control dependencies, a type of branch effect. Classically, if some condition is true, control transfers to the instruction at the branch's target address. The branch is then "taken," and its sign becomes *T* or 1. If the condition is false, execution continues with the instruction immediately after the branch, in which case the sign is *N* ("not taken") or 0. The computer cannot execute the code after a branch until it executes the branch and updates the program counter. With this restriction, parallelism can be exploited only from the instructions occurring up to the next branch. Because a branch path (code between executed branches) is typically three to nine instructions, and because data dependencies also restrict parallelism, the speedup is only about 1.6.[1,2] The sidebar "How

0018-9162/97/$10.00 © 1997 IEEE

## How Dependencies Limit Instruction-Level Parallelism

Two instructions must be executed sequentially if there are dependencies between them. A *resource* dependency arises if there are insufficient resources, such as adders, to execute all possible pending instructions simultaneously. *Semantic* dependencies require instructions to execute sequentially to ensure correct program results. Within this class are data and control (or procedural) dependencies. Both consist of a set of classical dependency types that restrict the available instruction-level parallelism. By determining a minimal set of these dependencies—a set that contains only true dependencies—more parallelism can be made available.

Table A shows classical data dependencies. 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 data dependencies is composed of flow or true data dependencies only. The other two types of data dependencies can be eliminated with renaming. In renaming, multiple copies of instruction sinks, such as *A*, are created. We assume that renaming is used throughout this article.

**Table A. Classic data dependencies.**

| Dependency name | Alternate (hazard) name | Example |
|---|---|---|
| Flow or True | (read after write) | 1. A = b + c<br>2. z = A ∗ y |
| Anti- | (write after read) | 1. z = A + c<br>2. A = y ∗ x |
| Output | (write after write) | 1. A = b + c<br>2. A = z ∗ y |

Recent research is exploring the possibility of reducing the effects of even true data dependencies using data prediction and speculation. Results are still inconclusive, however.

Classically, all instructions after a branch must wait for the branch to execute before they can execute. In the following example, instructions 2 through 7 are control dependent on instruction 1, a branch.

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

With minimal control dependencies, the execution of instructions 3 through 7 does not depend on whether instruction 1 is taken. Because instructions 3 through 7, including the branch at 5, can execute concurrently with instruction 1, more parallelism is realized.

Dependencies Limit Instruction-Level Parallelism" describes both data and control dependencies.

On the face of it, then, designers are stuck—they cannot create processors that execute more than one or two machine instructions per cycle. However, if branch effects could be completely eliminated, performance could improve 25 to 158 times over that with sequential execution.[1,3]

### Branch effect reduction

Branch effect reduction techniques, or *BERTs*, attempt to free instruction-level parallelism using the mechanisms listed in Table 1. The table lists the techniques we describe here.

As the table shows, a technique can use more than one mechanism. Most work has gone into speculative execution techniques, and they are consequently more common in commercial machines.

**Speculative execution.** This mechanism conditionally executes code after a branch, even if the code is dependent on the branch. Hence, execution is *speculative* because code is executed before the processor knows it should be executed.

Branch predictors, which attempt to predict the branch sign, are key to most forms of speculative execution. The path predicted to be followed is the *predicted path*; the path predicted not to be followed is the *not-predicted path*. The predicted path can be of either branch sign (not-taken or taken). A technique commonly predicts the branch path after the code being executed enters the processor's execution window but before the branch has resolved (before the sign is actually known).

Most speculative execution methods are *single-path* because they execute down one path from a branch. When the processor encounters a branch, the technique predicts the branch sign, and execution proceeds down the predicted path. However, because the branch is unresolved, the processor performs all writes to registers or memory and all I/O operations conditionally, finalizing them only when it is certain that all previously speculated branches have been predicted correctly. If there is a misprediction before a conditional operation, that operation is discarded. Hence, the greater the distance between mispredictions, the more parallelism can be extracted.

The accuracy of a technique's prediction is expressed as its *branch prediction accuracy*, the average fraction of correct predictions. The amount of instruction-level parallelism a reduction technique can realize is extremely sensitive to its branch prediction accuracy. For example, improving branch prediction accuracy from just 85 percent to 90 percent increases the distance between mispredictions by 50 percent, as given by

$$\text{distance} \propto 1/(1 - \text{accuracy})$$

Another important concept is the *branch target buffer*—a form of cache commonly used to handle branches through hardware. Typically, before a processor can execute a branch as taken, it must compute the branch's target address. This computation slows down

**Table 1. BERT mechanisms and implementations.**

| Technique | Mechanisms used: Speculative execution | Mechanisms used: Branch range reduction | Mechanisms used: Block size increase | Commercial implementation examples |
|---|---|---|---|---|
| **Eager execution** | ✔ | | | IBM 360/91, Sun SuperSparc |
| **Disjoint eager execution** | | | | |
| alone | ✔ | | | — |
| with minimal control dependencies (MCD) | ✔ | ✔ | | — |
| **Single path** | | | | |
| No branch prediction | | | | Intel 8086 |
| Static | | | | |
| Always not taken | ✔ | | | Intel i486 |
| Always taken | ✔ | | | Sun SuperSparc |
| Backward Taken; Forward Not Taken (BTFN) | ✔ | | | HP PA-7x00 |
| Semistatic (profiling) | ✔ | | | Early PowerPCs |
| Dynamic | | | | |
| 1-bit | ✔ | | | DEC Alpha 21064, AMD-K5 |
| 2-bit | ✔ | | | NexGen 586, PowerPC 604, Cyrix 6x86, Cyrix M2, Mips R10000 |
| Two-level adaptive | ✔ | | | Intel Pentium Pro, AMD-K6 |
| Selector | ✔ | | | DEC Alpha 21264 |
| Hybrid | ✔ | | | — |
| Multiscalar | ✔ | ✔ | | — |
| **Other BERTs** | | | | |
| Minimal control dependencies | | ✔ | | — |
| Predication alone | | ✔ | ✔ | Denelcor HEP |
| Predication with software | ✔ | ✔ | ✔ | Cydrome Cydra 5, Intel Pentium Pro |
| VLIW | ✔ | ✔ | ✔ | Multiflow Trace, Cydrome Cydra 5, Intel/HP Merced (?) |

the branch's execution, but the target address is saved in the branch target buffer. When the branch is executed again, the availability of the target address eliminates the time penalty that would occur otherwise. The buffer can also hold miscellaneous branch prediction information, such as the predictor's state.

**Branch range reduction.** This mechanism has two approaches. One is to use the set of minimal control dependencies. As the sidebar "How Dependencies Limit Instruction-Level Parallelism" describes, the classical model of control dependencies that all commercial and most research processors use treats all dependencies as true instead of recognizing the minimal set that are actually true. This is relatively inexpensive but misses significant potential performance gains.[3]

Another form of this mechanism is *predication*, in which some assignment statements are executed only if another input to the statement, a *predicate*, is true.[4]

**Block size increase.** This mechanism increases the distance between branches, thus increasing the size of the average basic block and increasing the amount of code available for parallelism. Techniques include compiler-based methods, such as code percolation or motion, or trace scheduling.[5]

## SPECULATIVE EXECUTION

Speculative execution can be realized in hardware or software and can be used among processors as well as within them. Although speculative execution most often refers to single-path, *eager execution* and the more recent *disjoint eager execution* (DEE) are also possible.[6] Figure 1 illustrates their differences.

Typically one or two processing elements are needed to execute the code in a branch path as concurrently as possible. In the single-path strategy, these resources are assigned linearly according to the number of branches pending. This strategy lowers hardware cost, but the usefulness of increasing predictions becomes negligible quite rapidly. The overall likelihood or *cumulative probability* of execution of the last branch path (at the tail of the tree) goes to zero, making the added resources useless.

With the eager execution model, execution proceeds down both paths of a branch, and no prediction is made. When a branch resolves, all operations on the not-taken paths are discarded. Consequently, eager execution with unlimited resources (oracle execution), would give the best performance, but it is hardly practical. With constrained resources, the eager execution strategy does not perform very well.[1] Also, hardware cost rises exponentially with each level of branches, and it is hard to keep track of different sets of operations. For these reasons, the eager execution strategy is seldom used, except for limited applications, such as instruction fetch and decode in the Sun SuperSparc and IBM 360/91.

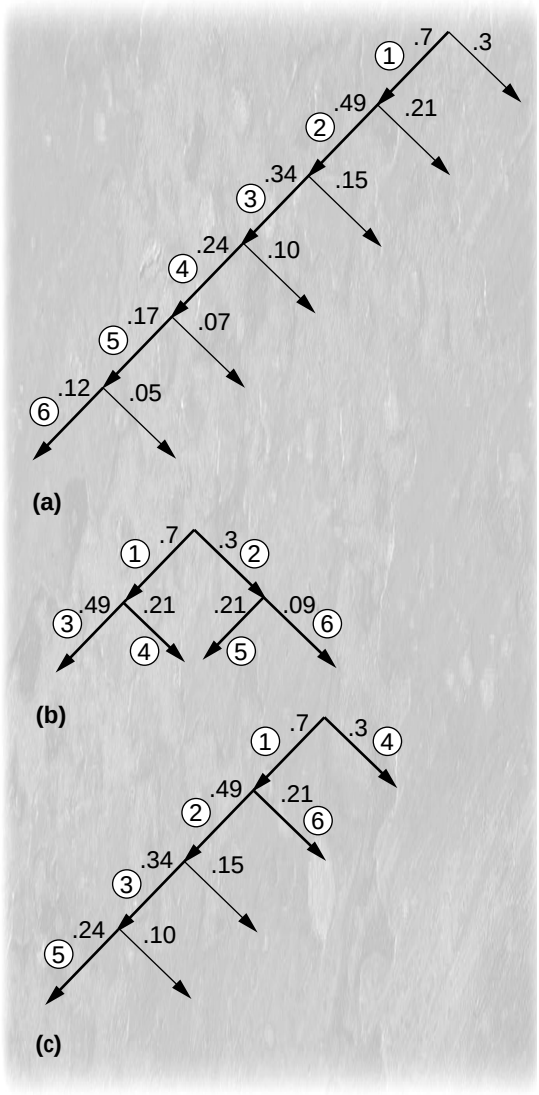


***Figure 1. Speculative execution strategies: (a) single path, (b) eager execution, (c) disjoint eager execution. Each line segment with an arrow represents a branch path. Resources are fixed at six branch paths. Bold lines indicate the code in the execution window; resources are assigned only to bold lines (paths). All branches are pending (unresolved). Left-pointing lines are predicted paths. Right-pointing lines are not-predicted paths. Circled numbers indicate the order of the resource assignment. Uncircled numbers indicate the cumulative probability that the path will be executed. For illustration, branch prediction accuracy is 70 percent for all branches. The disjoint eager execution strategy allocates resources to more likely paths than the other strategies.***

The disjoint eager execution strategy performs better than the other two strategies when resources are limited. The idea is to assign resources to branch paths whose results are most likely to be used; that is, branch paths with the highest cumulative probabilities of execution. Thus, all branches are predicted, and some are eagerly executed. The hardware cost is close to that of single-path, but performance is much better. As the sidebar "Disjoint Eager Execution: A Simulation Experiment" describes, speedups of 32 are possible. Many instantiations of this strategy provide variations in cost-accuracy trade-offs; we describe one implementation in the sidebar.

Most speculative execution uses some form of branch predictor. The latest ones are very accurate but improve branch prediction accuracy by less than a percent—an indication that branch prediction accuracy may be topping out. We describe the most common predictors here.

## Static predictors

Static predictors operate by making hardwired predictions, typically that branches are executed as either all not taken (Intel i486) or all taken (Sun SuperSparc). These techniques cost practically nothing but have an accuracy of only 40 to 60 percent. More involved but still inexpensive methods also look at branch direction. BTFN (backward taken, forward not taken), for example, predicts that all backward branches are taken and all forward branches are not taken. Because backward branches are taken typically 90 percent of the time, BTFN improves branch prediction accuracy to 65 percent. The HP PA-7x00 processors use this strategy.

*Semistatic predictors* form a large class of static predictors. Again, predictions are constant over the program's execution. However, unlike other static predictors, semistatic predictors vary across static branches. And because the compiler makes these predictions, they are included in the machine instructions, which means that if designers port this method to an existing processor they must modify the processor's instruction set.

The compiler makes predictions using *program profile statistics*, which it obtains by compiling the program once and then running it on test data while counting the times a branch is taken versus the times it is not taken. The program is recompiled, using the statistics to set the prediction bits in the object code's branches accordingly.

These predictors are limited because the statistics, and hence predictions, can vary from the test data to the actual data. Allowing predictions to vary from branch to branch improves the prediction accuracies of forward branches primarily; a typical forward branch executes predominantly with one sign. Therefore the branch prediction accuracy improves to, on average, 75 percent. Many PowerPC processors use semistatic prediction.

## Dynamic predictors

In dynamic prediction, predictions adapt to the input data. A branch may execute consistently one way in one part of the execution and the other way in another part. A dynamic predictor can adapt to the change and continue to make accurate predictions; a semistatic 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.

Dynamic predictors are typically 1-bit or 2-bit, so named because of the storage needed to implement them. The two-level adaptive predictor, a more recent type, greatly increases the branch prediction accuracy of the 2-bit predictor. The selector predictor allows multiple predictors to be used together.

**1-bit predictors.** Figure 2a shows how a 1-bit prediction algorithm uses state to predict that a branch will execute next the same way. Nominally, there is a separate automaton (state machine) for each static branch.

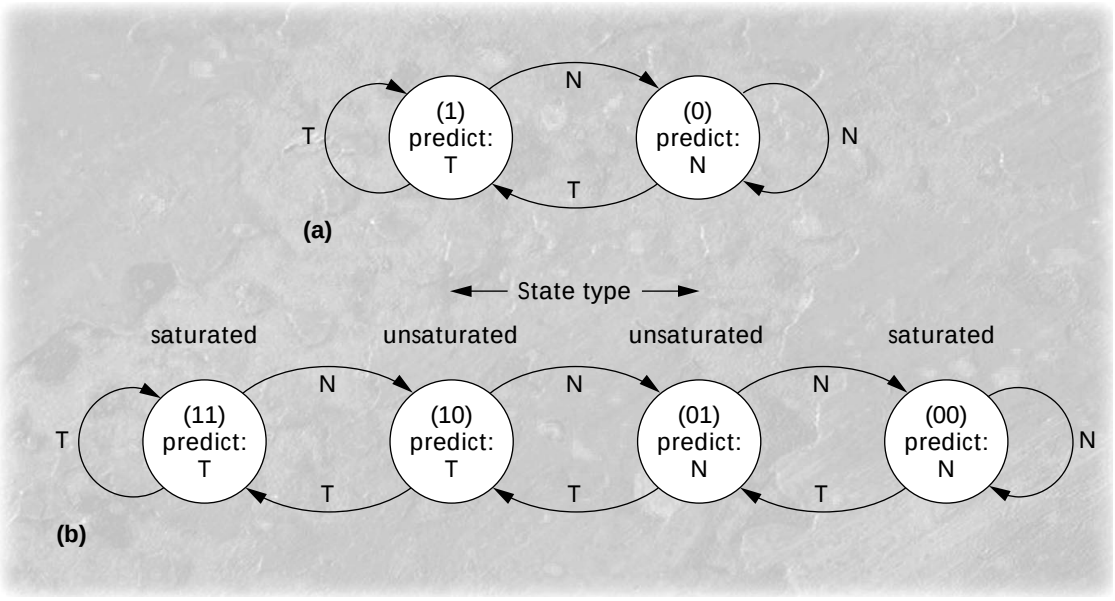


*Figure 2. Simple dynamic branch predictors, which predict if a branch is taken or not taken by looking at the most significant bit of the predictor's state. This bit gives the sign of the branch: 1 is "predict T(aken)"; 0 is "predict N(ot taken)." A state transition occurs when a branch resolves, and is determined by that branch's sign. (a) One-bit branch predictor and (b) 2-bit predictor.*

The state of the automaton becomes 1 if a branch is actually taken and 0 if it is not. The new state indicates the prediction for the next instance of the branch.

The automaton can be realized implicitly with a branch target buffer. If the buffer contains an entry for the branch, the branch was taken when last executed, and the dynamic prediction algorithm predicts that the same branch will be taken when next encountered. If there is no entry in the buffer, the branch was not-taken when last executed, and the algorithm predicts it will be not taken again.

One-bit predictors have a branch prediction accuracy of 77 to 79 percent. The DEC Alpha 21064 processor uses this predictor, holding the state for up to 2K automata.

**2-bit predictors.** Figure 2b shows the 2-bit saturating up/down counter developed by James Smith.[7] Performance is better (78 to 89 percent accuracies on real machines), but the cost is higher.

Each 2-bit automaton's state is stored in a branch target buffer. A branch is predicted by reading the buffer and using the state of the automata. Branches that are more often taken are predicted taken; likewise for not-taken branches. In this way, the predictions are based on averaging.

The 2-bit predictor is less affected by occasional changes in branch sign than the 1-bit predictor. In the branch execution stream N-N-N-T-N-N-N, the 1-bit predictor gives two mispredictions; the 2-bit predictor, only one. However, the 2-bit predictor can potentially be wrong 100 percent of the time (if starting from state 01, every branch in T-N-T-N-T-N... would be mispredicted).

Recent microprocessors, such as the NexGen 586 (2K automata) and the Intel Pentium (256 automata) use this predictor.

**Two-level adaptive predictor**. Researchers at the University of Michigan[8] and later IBM and the University of Texas[9] devised the two-level adaptive, or branch correlation, predictor, which is significantly more accurate (typically 93 percent accuracy)

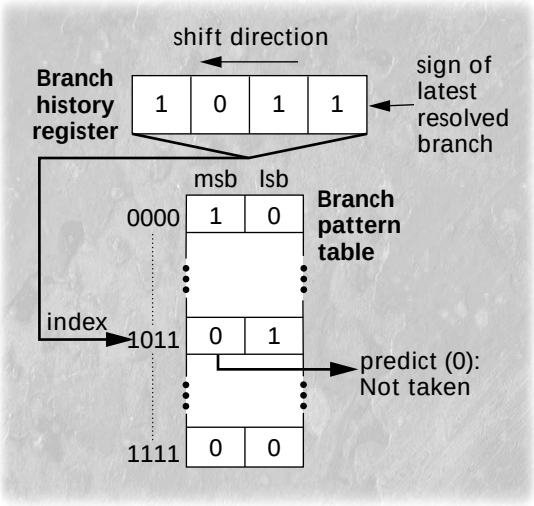


*Figure 3. Two-level adaptive branch predictor. Each row of the branch pattern table is the equivalent of the 2-bit counter in Figure 2b. The branch history register holds the signs of past branch executions. The predictor uses this recent history to index to a particular automaton in the branch pattern table.*

than the 1- or 2-bit predictors because the predictor bases its predictions on specific branch histories, not on a general averaging.

As Figure 3 shows, prediction involves two structures. The *branch history register* holds the branch execution history. Each time a dynamic instance of any branch resolves, its sign is shifted into the register. The register helps prediction by capturing much longer and more varied patterns of branch executions, relative to a 2-bit predictor. The *branch pattern table* contains a 2-bit counter automaton for each possible pattern of the branch history register. Typically, a processor uses one register and one table for all branches.

The automata are accessed using the contents of the branch history register as the table's address ("index" in the figure). As with the 1- and 2-bit predictors, the state of the indexed automaton indicates the prediction.

Using a single branch history register, the predictor combines information from multiple branches, allowing the correlation among different static branches

## Disjoint Eager Execution: A Simulation Experiment

We simulated disjoint eager execution (DEE) and DEE with minimal control dependencies (DEE-MCD) models using a heuristic devised by Augustus Uht.[1] We also simulated single-path speculative execution, eager execution, single-path speculative execution with minimal control dependencies, and an oracle. We used a modified version of Monica Lam and Robert Wilson's simulator.[2]

The simulator operates on MIPS R3000 machine code but assumes every instruction executes in one cycle. Using the Smith 2-bit dynamic branch predictor, we simulated five of the six SPECint92 benchmarks,[1] omitting the more predictable `sc` benchmark. We traced each benchmark for up to 100 million machine instructions. All models used the number of branch path resources allowed as the independent variable. This number and the geometric mean of the SPECint92 benchmarks' branch prediction accuracies—90.53 percent—determined the shape of the simulated static trees.

### Execution model

As we describe in the main article, DEE works by assigning resources only to the most likely paths to be executed. The heuristic uses a logical static tree of resources,[1] shown in Figure A, to avoid determining the most likely paths at runtime. The tree's shape is determined when a uniprocessor is designed. The assumption is that the predictor exhibits the same branch prediction accuracy on every branch. With a constant accuracy, the tree has the same basic shape—a relatively long mainline region with length $l$ and relatively few side paths. For a constant branch prediction accuracy, the shape of the DEE region is the lower right half of a square with side dimensions of $h_{DEE} = w_{DEE}$.

From geometric analysis the relationship between

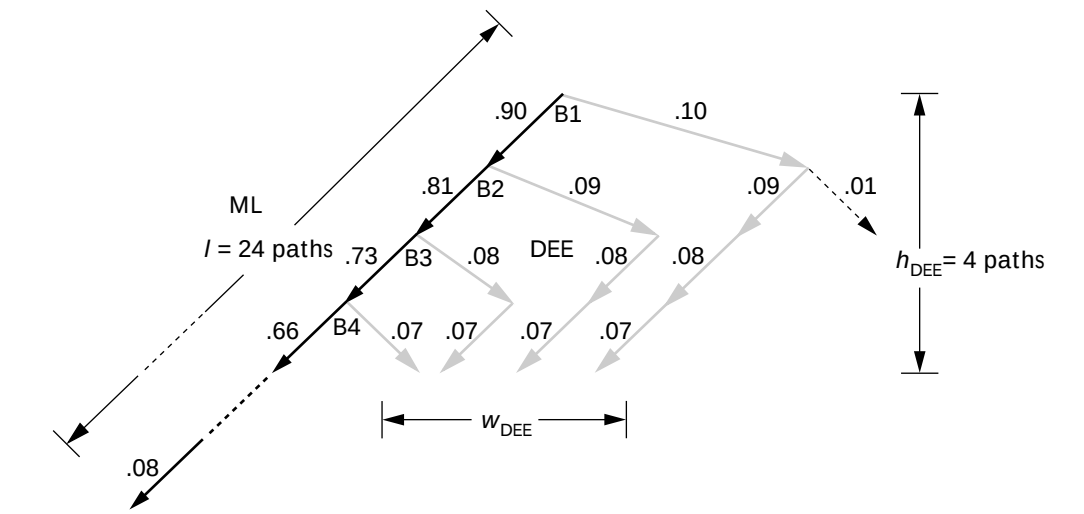


*Figure A. Typical static assignment tree in disjoint eager execution. ML is the mainline path or region. The number on each path is the overall cumulative probability of the path's execution. The execution of ML can be realized via single-path speculative execution. Each composite DEE path to the side (gray paths; each one to four branch paths long, in this example) can be treated with single-path execution within itself. All the composite DEE paths form the DEE region. Branch prediction accuracy is 90 percent. The total number of branch paths is 34.*

to be exploited. Implementers can also form the index by concatenating or hashing part of the program counter that identifies the static branch with the branch history register. This slightly improves prediction accuracy.

For the sequence T-N-T-N-T-N..., which the 2-bit predictor may mispredict 100 percent of the time, the adaptive predictor asymptotically correctly predicts branches 100 percent of the time.

Multiple branch history registers and branch pattern tables are possible. The two-level adaptive method has nine possible configurations, but accuracy typically comes with a high hardware cost. Fortunately, as hardware densities improve, the relative cost will become much lower.

The AMD-K6 uses the two-level adaptive predictor. A single 9-bit history register is concatenated with 4 bits of the program counter to index a pattern table of 8K 2-bit counters. An undisclosed version of the two-level method is also used in the Intel Pentium Pro (P6).

A variation of the two-level adaptive predictor adds profiling information about the degree of a branch's predictability. This approach better allocates predictor resources.

**Selector predictors.** The selector predictor[10] uses multiple predictors simultaneously: typically two main predictors (although implementers can modify the selector predictor to choose from more than two) and a selection predictor that chooses the better of the two main predictors to predict a given branch. The three predictors can be any combination of those already described. The selector pre-

branch prediction accuracy (BPA), $h_{DEE}$, and $E_T$ (total number of branch paths in the tree) is

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

Using the quadratic formula, we solve this equation for $h_{DEE}$. The mainline length $l$ is given by

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

Therefore, to determine the tree's shape at design time, the designer first determines the likely value of the branch prediction accuracy of the branch predictor to be used 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 $E_T$.

To get the the tree's dimensions, the designer simply plugs the BPA and $E_T$ values into the above equations. Of course, detailed study would then fine-tune these dimensions.

At runtime the actual control flow maps onto the tree, and the tree 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. Implementers can use this execution model either within or among uniprocessors.

### Results

Figure B shows the results of the simulations. For 100 branch paths, the DEE-MCD model is 31.90 times (3,090 percent) faster than the sequential machine. The jump in performance from 16 to 32 paths is a result of DEE being the same as single-path speculative execution for 16 paths and below.

These results indicate that DEE is effective in models with fewer resources. The speedup is three times better than that achievable with an unlimited resource version of single-path speculative execution with a selector predictor.[3] They also show that the DEE-MCD model exploits about 59 percent of the instruction-level parallelism that can be obtained from these benchmarks (which we determined by comparing it with the oracle simulations).

We plan to use the DEE-MCD model in Levo, a prototype machine we are developing at the University of Rhode Island. Tens 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. We anticipate 32 processing elements for Levo, to yield a speedup in instruction-level parallelism of 20 or more. We estimate that Levo could fit on a single chip by about 2000.

### References


1. A. Uht and V. Sindagi, "Disjoint Eager Execution: An Optimal Form of Speculative Execution," *Proc. 28th Int'l Symp. Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 313-325.
2. M. Lam and R. Wilson, "Limits of Control Flow on Parallelism," *Proc. 19th Annual Int'l Symp. Computer Architecture*," ACM Press, New York, 1992, pp. 46-57.
3. D. Wall, "Limits of Instruction-Level Parallelism," Tech. Report 93/6, Digital Western Research Laboratory, Palo Alto, Calif., 1993.


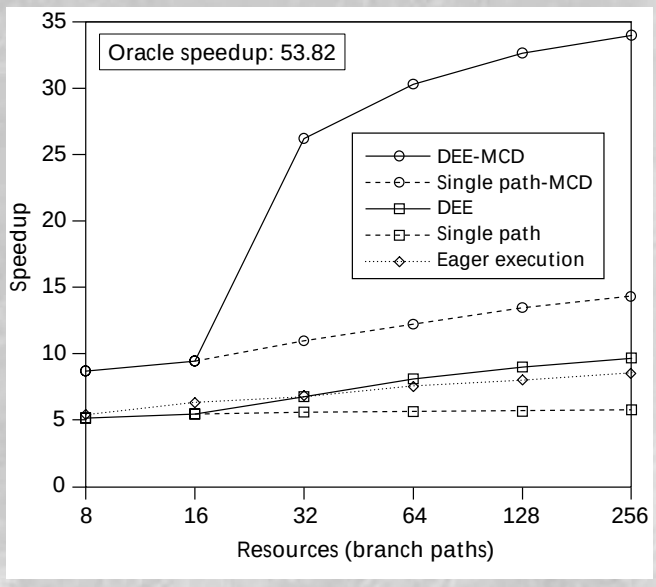


***Figure B. DEE simulation results on five SPECint92 benchmarks (sc omitted). Speedup is with respect to a strictly sequential model, computed as the harmonic mean of the five individual benchmarks' speedups. The best performance occurs when DEE is combined with the minimal control dependencies model. Results are about 10 times better than those previously demonstrated, either in simulations or on real machines.***

dictor is successful because a particular branch is often predicted more accurately by a certain predictor.

This method was adopted in the DEC Alpha 21264 in Fall, 1996. The total state storage for this implementation is greater than 28K bits.

### Hybrid predictor

The hybrid predictor[11] combines the static profiling of semistatic prediction with the dynamic branch correlation of the two-level adaptive method. Each static branch again contains a prediction bit, but the profiling phase is much more involved. The predictor records different instruction sequences during execution up to each static branch, *Y*, indicating how the dynamic branches that occurred before *Y* executed. The predictor then modifies the object code to allow different static predictions of *Y* to be used at runtime. It does this by copying *Y* and its branch path for each unique sequence through *Y*, and setting each *Y*-copy's prediction bit according to the likely sign of *Y* for the corresponding sequence.

Thus, *Y* is effectively predicted at runtime according to the program's actual control flow, but *not* directly on the basis of *Y*'s dynamic execution history. Because *Y*'s predictions can vary dynamically and adapt to some of the program dynamics, hybrid prediction has a branch prediction accuracy of about 88 percent versus semistatic prediction's accuracy of about 75 percent. However, it does not adapt as well as the two-level adaptive method, which also takes into account *Y*'s history. Also, object code may increase significantly because of

**Table 2. Classic code versus predicated code.**

| 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<br>// all dependent on 1. | 4. // later instructions: NOT dependent on 1. |

code replication. On the other hand, no predictor hardware is needed.

### Implementation issues

Branch prediction systems are constrained by the number of predictions that may be made simultaneously. To realize greater speedups from exploiting instruction-level parallelism, a predictor must make multiple predictions per cycle—yet most current machines allow only one branch to be predicted at a time. The Levo machine,[6] which we briefly describe in the sidebar "Disjoint Eager Execution: A Simulation Experiment," allows up to 32 predictions per cycle. Philip Emma and his colleagues at IBM Yorktown Heights have also devised hardware that allows many predictions per cycle.

*Confidence predictors* indicate what branch predictions are likely to be correct, which will aid machines using forms of eager execution (these predictors have not yet been used in a real machine). Confidence predictors can use the same basic structures as branch predictors.

Another implementation model is *multiscalar* machines, which are being studied at the University of Wisconsin.[12] In this model the compiler divides a program into smaller computation blocks, or *tasks*. Because the dynamic task sequence is predicted, single-path speculative execution is performed at the task level. *Task predictors* use solutions similar to those for branch prediction, except that the prediction may have more than two possible outcomes, corresponding to multiple exits from a task. Task sequences are predicted independently of branch predictions within the tasks. The task graph is included in the object code. Each task is executed by a typical superscalar processing engine. The engines in a multiscalar machine attempt to execute tasks concurrently.

The drawback is that the instruction set must be modified, requiring recompilation. With additional research, the available instruction-level parallelism may increase.

## BRANCH RANGE REDUCTION

Approaches that use this mechanism include minimal control dependencies and predication.

### Minimal control dependencies

The minimal control dependencies model allows the execution of both assignment and branch instructions absolutely, not speculatively, in parallel with earlier branches. To our knowledge, it has been hardware-implemented only in the Condel simulators.[13] The gains in instruction-level parallelism are modest when this technique is used alone, but increase significantly with single-path speculative execution or disjoint eager execution.

Using minimal control dependencies decreases the effect of mispredictions relative to the classical control dependencies model. With the classical model, *all* instructions after the mispredicted branch are discarded; with the minimal control dependencies model, only the truly dependent instructions—many fewer—are discarded.

### Predication

Predication[4] is receiving much attention. As Table 2 shows, a predication algorithm puts the Boolean results of condition testing into predicate registers and eliminates branches. Instructions that were dependent on a branch now have a 1-bit predicate register as input. An instruction executes only if its predicate is true.

Predication eliminates some branches and their associated control dependencies, increasing the distance between mispredictions. But it also affects the instruction set, adds a port to the CPU register file, and complicates instruction execution. Moreover, without sophisticated compilers, predication does not realize much instruction-level parallelism.

There is a nominal increase in branch prediction accuracy, but it is artificial: The overall code dependencies do not decrease below those with minimal control dependencies because the control dependencies removed by predication become data dependencies. Future research may improve predication results.

## BLOCK SIZE INCREASE

The best known technique to increase basic block size is compiler-based *trace scheduling*, which is used in most VLIW (very long instruction word) computers, for example, the Multiflow Trace.[5] Trace scheduling unrolls loops to make the block larger. Profiling is used to estimate the most likely path or "trace" through the code.

Machine operations that can be executed in parallel are grouped into VLIW machine instructions. Typically, each instruction includes both assignment operations and a multiway branch. Operations within a VLIW instruction are independent, so scheduling or dependency-checking hardware is not needed. However, recompilation is necessary whenever the processor is replaced. Because recompilation is not

**Table 3. Comparative performance of branch effect reduction techniques.**

| | Performance | | | Characteristics: Hardware | | | Software | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Technique | ILP speedup realized | BPA, research (percent) | BPA, real (percent) | Real machine hardware | Instruction set modified? | Hardware cost | Code size increase? | Profiling necessary? | Recompilation necessary? | Software complexity |
| **Single path with the specified branch predictor in order of increasing BPA** | | | | | | | | | | |
| 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 |
| Semistatic (profiling) | 1.5 | 83.4 | 75 | 1 op code bit | Yes | 3 | No | Yes | No | 2 |
| Hybrid | NA | 88 | — | 1 op code bit | Yes | 4 | Yes; 2 | Yes | No | 3 |
| 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 | ≥2K bits | No | 6 | No | No | No | 0 |
| Selector | NA | 93-97 | NA | ≥28K bits | No | 7 | No | No | No | 0 |
| **Other reduction techniques using various branch predictors in order of increasing speedup** | | | | | | | | | | |
| Predication (alone) | 1.6 | — | — | Sev. op code bits | Yes | 7 | No | No | No | 0 |
| MCD (hardware) | 2.1 | — | — | — | No | 7 | Yes; 1 | No | No | 1 |
| Multiscalar | 2.6 | — | — | Sev. op code bits | Yes | 8 | Yes; 1 | No | Yes | 3 |
| Predication (with software) | 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 | 7.6 | — | — | 1 branch bypassed | No | 9 | No | No | No | 0 |
| Disjoint EE | 8.5 | — | — | — | No | 7 | No | No | No | 0 |
| DEE w/ MCD | 31.9 | — | — | — | No | 8 | Yes; 1 | No | No | 1 |

* "NA" means not available; a "—" means the characteristic is not relevant or that the technique has not been used in a real machine.

always possible, VLIW machines have a limited appeal. Also, they do not typically exploit code dynamics.

Trace scheduling often greatly increases object code size. Further, it is not clear that VLIW machines can use the minimal control dependencies model to execute multiple multiway branches in parallel—a difficult problem. Research efforts continue. For example, Intel and Hewlett-Packard have joined in a project to create a new microprocessor (Merced or P7) compatible with both the Intel x86 and HP Precision architectures. This microprocessor may be VLIW based.

### Variations

Software pipelining is a variant of trace scheduling and uses the minimal control dependencies model. This variant can sometimes achieve the equivalent of eager execution performance, but with less hardware.

Other techniques, such as boosting or spreading use some VLIW methods. One method is *code motion*, in which instructions are moved at compile-time to enhance the amount of instruction-level parallelism available to the target machine. The speedup realized with VLIW or VLIW-aided methods is typically less than 3.

### Implementation issues

When a processor is changed, updating the machine code is often costly or impractical because recompilation is not possible. For that reason, users tend to stay with the same instruction set architecture. This limits the use of techniques that require changes to existing instruction sets. On the other hand, this constraint may not always apply, such as for embedded machines.

## COMPARATIVE PERFORMANCE

Table 3 summarizes the techniques we have described and gives performance results. Results are based on de facto standard general-purpose benchmarks—the SPECint92 suite. When this data was not available, we used the data closest to those benchmarks.

We give each technique's performance in terms of speedup factors and branch prediction accuracy. The "BPA, real" column consists of numbers that include hardware and code limitations and are based on real machines. The "BPA, research" column consists of figures from studies that look at the techniques in isolation.

We also include the hardware and software characteristics needed to obtain that performance. Only the

The commercial exploitation of instruction-level parallelism has really just begun.

single path with "None" assumes unrestricted resources. Hardware attributes include typical hardware cost ("Real machine hardware"), whether the instruction set is affected, and a ranking of the hardware costs. The "Real machine hardware" column contains the term "entries." An entry is the state (memory bits) for one automaton. "Op code bits" are additional bits required in either branch or assignment instructions.

Software characteristics include whether object code size increases and whether program profiling or recompilation is necessary in a machine upgrade or downgrade. We also rank software complexities.

All ranking is based entirely on our opinion. Ranking is by order only specific to that column, from best (lowest number) to worst (highest number).

We took data in the performance columns and actual hardware cost from various sources, most of which are references included at the end of this article. Much of the data is from various issues of the *Microprocessor Report* newsletters.[14] We could not cite all our sources, but they are available via http://www.ele.uri.edu/~uht. In the "ILP speedup realized" column, speedup numbers are factors from research machine simulations. Each factor (minus 1) represents a 100x percentage. That is, a 1.6 speedup translates to 60 percent faster. Each was computed as the harmonic mean of the individual benchmarks' speedups. A technique's actual operation and its simulated operation differ in varying degrees from technique to technique and study to study.

In the "Other reduction techniques" section of the table (bottom half), the techniques MCD (hardware-based) and DEE-MCD work best with some binary-to-binary code translation via a filter program run once for an application program. The main action performed is limited loop unrolling. The filter is not required for correct code execution. The data given for DEE-MCD is for a hardware-based implementation with a 2-bit dynamic predictor, although software-based implementation is also possible.

The commercial exploitation of instruction-level parallelism has really just begun. Branch effect reduction techniques can be implemented in hardware, software, or both to free up more parallelism and speed up the execution of general-purpose code. Software-based methods are good when hardware cost is a prime issue, instruction set compatibility is not required, and average performance is acceptable. Otherwise, hardware methods are better. For those who want to read about existing methods in more detail, an IEEE tutorial[15] and the proceedings of the International Symposium on Microarchitecture (IEEE Computer Society Press) are excellent sources.

Although branch prediction accuracies of up to 95 percent can be realized with single-path speculative execution alone, new methods are needed to break this barrier. Simulations of disjoint eager execution with minimal control dependencies have demonstrated an improvement 10 times greater than results with existing methods. In the near future, especially as hardware densities increase, these results should be possible commercially as well. ❖


### Acknowledgments

This work was supported by Intel, the University of Rhode Island Research Office, and the National Science Foundation through grant CCR-8910586. We thank Monica Lam and Robert Wilson of Stanford University for their simulator and their assistance, Qing Yang and the referees for their comments on drafts of this article, and Laurette Bradley for both her comments on earlier drafts and for her constant support. Finally, we thank all those who have published their work on instruction-level parallelism and wish we could have formally acknowledged many more.


### References


1. E. Riseman and C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," *IEEE Trans. Computers*," Dec. 1972, pp. 1,405-1,411.
2. D. Wall, "Limits of Instruction-Level Parallelism," Tech. Report 93/6, Digital Western Research Laboratory, Palo Alto, Calif., 1993.
3. M. Lam and R. Wilson, "Limits of Control Flow on Parallelism," *Proc. 19th Annual Int'l Symp. Computer Architecture*, ACM Press, New York, 1992, pp. 46-57.
4. B. Rau et al., "The Cydra 5 Departmental Supercomputer," *Computer*, Jan. 1989, pp. 12-35.
5. R. Colwell et al., "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Trans. Computers*, Aug. 1988, pp. 967-979.
6. A. Uht and V. Sindagi, "Disjoint Eager Execution: An Optimal Form of Speculative Execution," *Proc. 28th Int'l Symp. Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 313-325.
7. J. Smith, "A Study of Branch Prediction Strategies," *Proc. 8th Annual Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1981, pp. 135-148.
8. T.-Y Yeh and Y. Patt, "Two-Level Adaptive Training Branch Prediction," *Proc. 24th Annual Symp. and Workshop Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 51-61.
9. S.-T. Pan, K. So, and J. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *Proc. 5th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1992, pp. 76-84.

10. S. McFarling, "Combining Branch Predictors," Tech. Report TN-36, Digital Western Research Laboratory, Palo Alto, Calif., 1993.
11. C. Young and M. Smith, "Improving the Accuracy of Static Branch Prediction Using Branch Correlation," *Proc. 6th Annual Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1994, pp. 232-241.
12. G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar Processors," *Proc. 22nd Annual Int'l Symp. Computer Architecture*, ACM Press, New York, 1995, pp. 414-425.
13. A. Uht, "A Theory of Reduced and Minimal Procedural Dependencies," *IEEE Trans. Computers*, June 1991, pp. 681-692.
14. L. Gwennap, "New Algorithm Improves Branch Prediction," *Microprocessor Report*, Mar. 27, 1995, pp. 17-21.
15. *Instruction-Level Parallel Processors*, H. Torng, and S. Vassiliadis, eds., IEEE CS Press, Los Alamitos, Calif., 1995.

*Augustus K. Uht is an assistant professor in electrical and computer engineering at the University of Rhode Island. He is also a member of the university's High-Performance Computing Laboratory, where he leads the Levo High ILP Prototype Computer project. His research interests focus on the enhancement of parallelism through the reduction of branch effects, but also include general computer architecture, memory systems, digital system design, and parallel computing. Uht received a BS and an MEng(Elect) from Cornell University, and a PhD from Carnegie Mellon University—all in electrical and computer engineering. He is a licensed professional engineer and a member of Sigma Xi, IEEE, IEEE Computer Society, ACM, and National Society of Professional Engineers.*

*Vijay Sindagi is a design engineer at Texas Instruments (ASIC), Dallas, where his interests include computer architecture, compilers, and ASIPs. Sindagi received a BS in electrical engineering from Bangalore University, India, and an MS in computer engineering from the University of Rhode Island. He is a member of ACM.*

*Sajee Somanathan is a software engineer at ADE Corp. His current interests include instruction-level parallelism, parallel computer architecture, branch prediction, and performance evaluation. Somanathan received a BTech from the College of Engineering, Trivandrum, India, and an MS in computer engineering from the University of Rhode Island.*

*Contact Uht at Dept. of Electrical and Computer Engr., University of Rhode Island, Kelley Hall, 4 East Alumni Ave., Kingston, RI 02881-0805; uht@ele.uri.edu.*