An Introspective Approach to Speculative Execution

Nemanja Isailovic
Computer Science Division
University of California, Berkeley
nemanja@cs.berkeley.edu

Abstract

In the introspective computing model, on-chip resources are divided into those used for *computing* and those used for *introspection*. The introspective processor or processors may perform sophisticated online observation and analysis of the computation and use the extracted information to improve performance, reliability, or other system properties. This paper considers the possibility of online construction of graphical models representing program behavior and the use of such models to perform better branch prediction. The particular graphical model that we employ is that of *decision trees*. We explore the space of decision tree models, consider the feasibility of implementation of the introspective processor, then present the performance of this model on the SPEC2000 benchmark suite.

1 Introduction

Historically, shrinking transistor sizes have allowed microprocessor architects to extract ever more instruction-level parallelism at a cost of ever increasing complexity. Each successive generation of high-performance chips has devoted a greater fraction of their transistors to control-related functions rather than actual execution units. Further, as the number of transistors in a microprocessor grows, design and verification time grow superlinearly with complexity. Seeking alternative ways to use transistors, computer architects have turned to different computation models such as chip multiprocessors (CMP) [12, 2, 21] and reconfigurable hardware [16].

Given the plethora of resources available to a modern computer architect, we might consider an alternative model, called *introspective computing*. In the introspective computing model, on-chip resources are divided into those used for *computing* and those used for *introspection*. The introspective processor or processors may perform sophisticated on-line observation and analysis of the computation and use the extracted information to improve performance, reliability, or other system properties. The basic introspective computing model is shown in Figure 1. Here, in addition to normal computation,

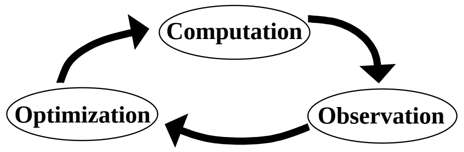

Figure 1: The Cycle of Introspection

we show some processor resources devoted to monitoring and others to adaptation. The essential element of this model is that the feedback is continuous and in real time.

Much of the hardware of an introspective system resembles a chip multiprocessor, possibly with FIFO communication channels rather than shared memory. Thus, we can design such a system by replicating a simple processor unit many times. The essential difference between introspective computing and multiprocessing lies in the use of resources. An introspective system does not attempt to utilize every available CPU cycle for performing the primary computation task. Instead, it devotes some of these resources to performing "meta-tasks" such as analyzing and adapting the behavior of running programs. The thesis is that this might ultimately lead to better overall behavior of the application than devoting all resources to (say) producing a wider superscalar. Existing examples of this model might include continuous dynamic (re-)compilation [3], fault analysis (*e.g.* DIVA [4]) and complex power adaptation [18].

The possibilities for introspective computing are numerous. In this particular paper, we explore the use of graphical models to perform a traditional microprocessor task: branch prediction. The existence of an introspective processor allows for the creation and manipulation of more complicated graphical models than would be practical (or desirable) with hardware alone. These graphical models encapsulate the most vital aspects of a given program execution with regard to the desired end: effective prediction of branch outcomes. These models are built from observations of the state of the running program and used to adjust the behavior of the program in the same run.

The particular graphical model that we have determined strikes the best balance between prediction effectiveness and feasibility of implementation is that of a *decision tree*. A decision tree is a tree whose interior nodes represent decision points (or questions about the past) and whose leaves represent predictions of the future. We show that it is possible to strike a compromise such that graphical models can be built and used in real time, while still being highly efficient predictors. In the rest of the paper, we explore the space of decision tree models and consider the hardware complexity of the introspective processor. We present results from the SPEC 2000 benchmark suite that show speedups as high as 22% for a four-way superscalar processor.

The remainder of the paper is divided as follows. Section 2 discusses general trends on computer design. Section 3 discusses previous work in this field as well as motivations for our own work. Section 4 contains an overview of general decision trees, the decision trees used in our algorithms, and the overall prediction mechanism employed. Section 5 describes the decision tree algorithms that we use. Section 6 discusses a possible implementation of the algorithm and the trade-offs that need to be considered in order to run this in real time. Section 7 provides experimental results. Section 8 discusses future lines of research, and Section 9 concludes.

2 General Trends

The argument is made in this paper that the extra transistors which will inevitably be available to future chip designers may be put to better use in introspection rather than in the standard forums (wider issue, more functional units, larger caches, etc.). However, in order to justify this argument, we must consider the actual performance improvement that has generally been achieved through these standard means as transistor count on a chip increases.

Obviously, actual performance improvement cannot be linked linearly (or in any other such simple manner) with transistor count, since performance depends on many factors of the design as well as the targeted uses of the chip. However, by studying average reported performance of various machines over a few years, we may be able to draw some general conclusions concerning performance improvement over time and increased transistor count.

Figure 2 shows the reported base results of several popular workstation processors on the SPEC 2000 Integer benchmark suite as a function of the chip's transistor count. The best-fit line through these points indicates that, in the last few years, a doubling in the number of transistors on a chip has resulted in approximately a 23.9% overall performance improvement.

Figure 3 shows similar results for earlier workstation processors on the SPEC 1995 Integer benchmark suite. In this case, the best-fit line through these points shows that a doubling in the number of transistors results in approximately a 27.9%

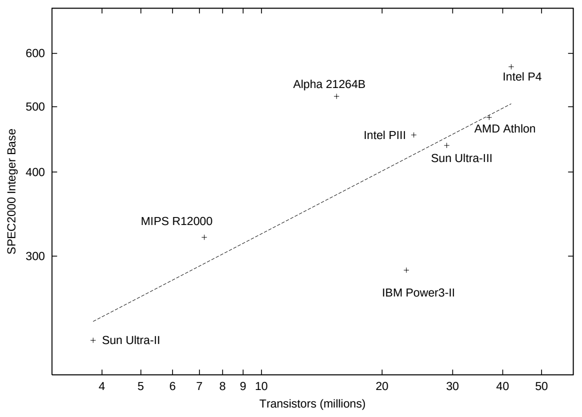

Figure 2: Various workstation processors that were released in the late 1990s and early 2000s. Each point represents one processor and shows the SPEC 2000 Integer Base performance metric and the transistor count for the machine.

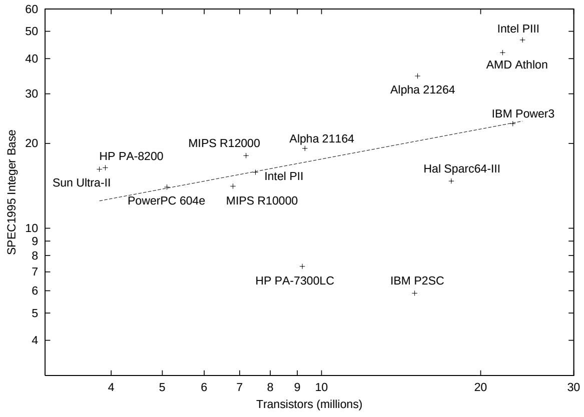

Figure 3: Various workstation processors that were released in the late 1990s. Each point represents one processor and shows the SPEC 1995 Integer Base performance metric and the transistor count for the machine.

performance improvement. So, using conventional techniques and technology and clock improvements over time, we can expect a 20-30% performance improvement for a doubling in the number of transistors.

3 Related Work

An enormous number of papers have been published in the last fifteen years regarding branch prediction techniques and analyses, using a variety of techniques both novel and recycled from such other fields as artificial intelligence. More recently, research has been conducted into the possibilities of introspection to make various run-time modifications in order to improve performance, power, etc.

Table-Based Predictors In 1992, Yeh and Patt published their paper on two-level adaptive branch prediction [22]. In this mechanism, predictions are made using 2-bit saturating counters. The counters are stored in a table which is indexed by a portion of the branch address as well as global and per-branch history. The 2-bit counter is updated based on whether the prediction matches the actual outcome of the branch.

Since only a portion of the branch address is used in indexing the table (in order to reduce table size), aliasing can be a significant problem [19]. Several schemes have been proposed which maintain the successful approach of the Yeh and Patt algorithm (tables of 2-bit saturating counters) while attempting to reduce aliasing [17] [15] [8] [20]. From among these, we have chosen the YAGS predictor [8] as a baseline against which to compare our own mechanisms in this paper, since YAGS is representative of a top-of-the-line predictor.

Branch Prediction Using AI Algorithms A few attempts have been made at using algorithms from the field of artificial intelligence (AI) in building better branch predictors. Calder et al. [6] devised a method of performing static branch prediction using neural networks. Their mechanism extracts program information at compile-time and feeds it into a trained neural network. They achieve 80% prediction accuracy overall, which is good for a static prediction mechanism, but significantly worse than can be achieved using modern dynamic prediction techniques.

Jimenez and Lin [14] explore the possibility of using the simplest of neural networks, perceptrons, to do effective dynamic branch prediction efficiently in hardware. Due to its simplicity, their mechanism is capable of considering longer branch histories than is possible using the table-based predictors mentioned above, for a given amount of hardware. Their predictor gets the best gains over bi-mode and gshare on benchmarks which have largely linearly separable branch behavior (that is, branch behavior which can be described by a perceptron).

Fern et al. [10] study the general approach of dynamically selecting features from among system state (and state history) which can be used to generate graphical models useful for prediction. They describe a dynamic decision tree algorithm and present experimental results which show that their design is close to PAp and GAp predictors in prediction success rate. Though our paper likewise describes a decision tree algorithm, we use a significantly different learning algorithm based on a different fundamental approach to the problem. We also show significant performance improvement over top-of-the-line branch predictors.

Analyses of Branch Prediction A more high level approach to this research problem has led to a series of theoretical papers containing analyses of various sorts. Young and Smith [24] suggest a method of statically analyzing branch correlations in order to transform code to take advantage of static prediction schemes. They expand upon this in a later paper [23] by analyzing the effectiveness of various prediction schemes based on correlations between branches.

An analytical study of two-level predictors was published by Evers, Patel, Chappell and Patt in 1998 [9]. The results of this paper show that branches have a strong tendency to correlate highly with only a few other branches (two or three at most). This conclusion was a large factor in our decision to use rather small decision trees to encapsulate a good deal of information about correlations.

Also in 1998, a paper was published [11] exploring the possibility of estimating confidence levels for predictions in order to only use those predictions with high confidence, thus avoiding unnecessary rollback after faulty predictions. More recently, Jimenez, Keckler and Lin published a paper [13] studying the effect that various realistic delays have on the design and optimization of prediction mechanisms. These two papers emphasize the fact that prediction accuracy is not the whole picture when it comes to branch prediction. Actual performance improvement on real program runs are the final test of prediction mechanisms. For this reason, our final performance numbers in this paper are reported in terms of performance (cycle count) rather than branch misprediction rate.

Introspection Recently, there has been work done in the area of introspection, though often under other names. In 1999, Todd Austin published a paper on DIVA [4], a system which uses dynamic verification to verify the correctness of a running program. Additionally, current research is going on at IBM under the name of Autonomic Computing [1], which focuses on creating “a systemic view of computing modeled after a self-regulating biological system.” They are essentially researching the possibility of a self-modifying and self-improving system, which is the basic idea behind the introspective processor discussed in this paper.

4 Decision Trees for Branch Prediction

In general, branch prediction involves the use of probabilistic models to predict the outcome of branch instructions. The process of predicting a branch is straightforward: first, select an appropriate model for the branch and second, apply historical information to this model. What makes branch prediction complex is the selection and training of the models.

Decision trees have been used in the literature for a variety of tasks. In this section, we discuss the use of decision trees to predict branches. We assume that each branch has its own set of decision trees. The result is a branch-specific probabilistic

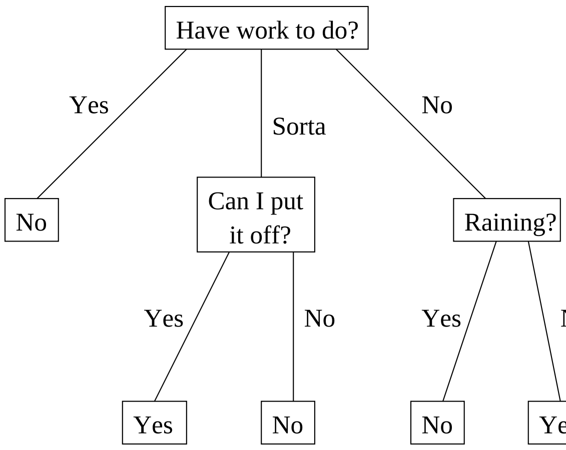

Figure 4: This sample decision tree helps one determine whether to go hiking today. To use it, start at the root, answer the question, follow the appropriate leg, and repeat this until a leaf node is reached. The leaf node returns the decision made by the tree given the current state of the world.

model. We describe these models, their physical representation and their impact on the computational processor. We will leave the details of producing "good" decision trees to later sections of the paper.

4.1 General Decision Trees

Decision trees are simple, yet often highly effective decision-making tools used commonly in work concerning artificial intelligence and other related fields. A decision tree uses data concerning the (present and past) state of the system under consideration to make a binary decision.

Consider the sample decision tree shown in Figure 4. This decision tree helps one decide whether to go hiking today, with the sole output being either a Yes (go hiking) or a No (don't go hiking). The tree takes as input portions of the state under consideration (in this case, the physical world).

Each non-leaf node in the tree corresponds to some determinate component of the state of the system. Each leg of a non-leaf node corresponds to a value of the component specified by the node. Note that some of the nodes have two legs to their children, while some have more. A general decision tree node can have any number of legs.

The actual use of the decision tree is rather intuitive. One begins at the root node, answers the question posed, and then follows the appropriate edge to the next question. One continues like this until a leaf node is reached. Each leaf node corresponds to a possible value for the output of the decision tree (Yes, No, 1, 0, Taken, Not Taken). For a given state of the system, some leaf node will ultimately be reached, and that value is the decision made.

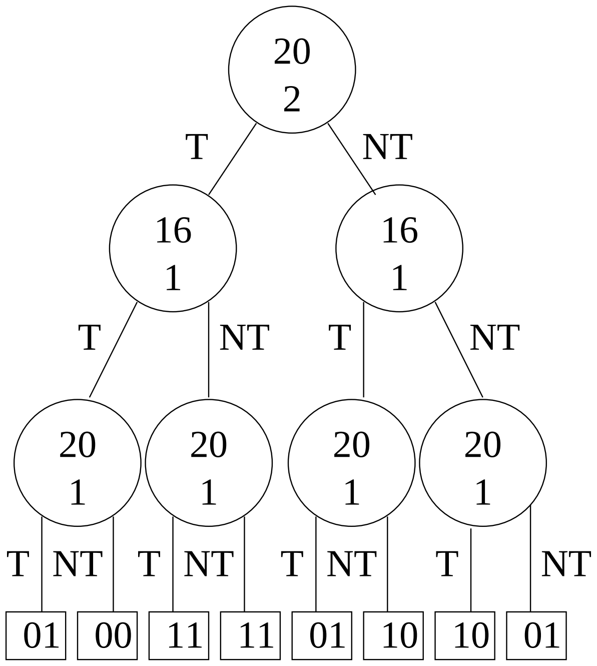

Figure 5: This is a sample Type 1 decision tree. Each non-leaf node contains an AOP (branch address and ordinal) and has two legs going out to its children, one for Taken, one for Not Taken. Each level consists of identical nodes. The leaf nodes are two-bit saturating counters.

4.2 Decision Trees and the Execution History

Figure 5 shows a simple decision tree for branch prediction. In the next section, we will call this a "Type 1" tree. Each internal node in the tree selects a particular branch in the program execution history, while each arc represent a taken or not-taken result of that branch. Branches in the execution history are identified with *Address-Ordinal Pairs* (AOPs), which we will discuss in a moment. The leaves of the decision tree consist of 2-bit saturating counters which are interpreted in the usual way: if the MSB is zero, the prediction is "false"; if the MSB is one, the prediction is "true".

Decision trees only make sense relative to a particular execution history. All of the algorithms discussed in this paper use the contents of the Augmented Global Branch History Register (AGBHR). A Global Branch History Register (GBHR) is an x-bit hardware shift register which stores the outcome (Taken or Not Taken) of the last x branches executed in the current program run, where x is usually on the order of 16.

The AGBHR is augmented in that, in addition to keeping track

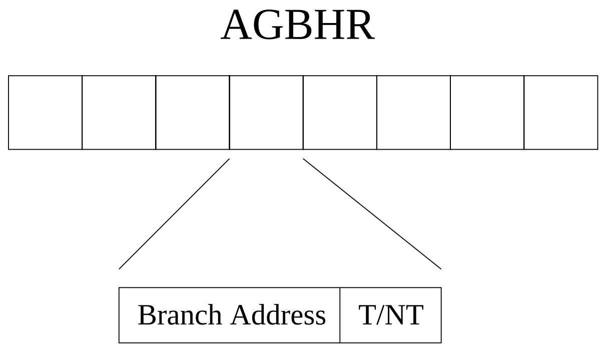

Figure 6: An 8-element AGBHR. Each element consists of a 32-bit address and a 1-bit outcome (Taken or Not Taken).

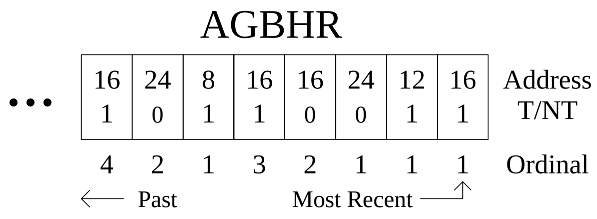

Figure 7: An 8-element AGBHR labeled with the appropriate ordinal for each element. The rightmost element is the most recent one.

of the outcome for the last x branches, it also keeps track of the instruction address of the branch that was executed. Thus, the AGBHR is an x-element hardware shift register in which each element consists of a branch address and a single bit representing Taken or Not Taken (see Figure 6).

A given branch address may appear more than once simultaneously in the AGBHR since that branch may have been executed more than once in the last x branch executions. When considering the elements of the AGBHR, we identify each one by the branch address specified and by an ordinal. For a given branch address, the most recent instance of that address in the AGBHR is assigned ordinal 1, the second most recent is assigned ordinal 2, etc., as shown in Figure 7. Thus, we care less about how recently the branch instance occurred overall and more about how recently it occurred relative to other instances of the same branch. An address/ordinal pair (AOP) completely specifies a particular element in the AGBHR.

4.3 Variations on a Theme

In our algorithms, each unique branch has a set of 2-bit saturating counters associated with it which are used to predict future outcomes. Graphical models, specifically trees, are a way of structuring these counters. The problem is finding a structure which allows for high learning potential while maintaining simplicity of design to allow for efficient hardware. As a result, we've tried a few different decision tree structures.

Type 1 Trees In a Type 1 decision tree (Figure 5), each level consists of nodes corresponding to the same AOP. In such a tree, any path from root to leaf passes through exactly the same AOPs. In effect, this correlates the branch being predicted with a small set of AOPs. Associated with each collective outcome of these AOPs is a separate 2-bit saturating counter.

This approach is reminiscent of the conclusions drawn by Evers et al in [9]. In that paper, the authors make the claim that many branches correlate highly with only very few other branches. If those few correlated branches could be found individually for each unique branch, then we wouldn't need the huge tables of gshare and other such predictors. With the Type 1 decision tree, we are in fact doing exactly that. We are correlating each unique branch with only x other branches, and thus we need 2^x 2-bit counters.

However, while Evers et al discuss this important conclusion about correlation, they do not provide an effective online algorithm for learning these correlations. Such an algorithm will be discussed later in this paper.

Type 2 Trees A degree of freedom can be added to the learning process, resulting in what we refer to as a Type 2 tree (see Figure 8). Each non-leaf node is again associated with an AOP, and each leaf node is a 2-bit saturating counter. The difference is that each level in a Type 2 decision tree need not consist of nodes containing the same AOP. Each node in a Type 2 tree is learned independently, thus allowing for the possibility that a "Taken" correlation with root lead to some further set of correlations, while a "Not Taken" at the root may lead to an entirely different set of correlations. This structure preserves the notion that each unique branch correlates with only a few other branches, but it allows greater freedom in types of correlations allowed.

Type 3 Trees Nonetheless, there is yet another degree of freedom which can be allowed in tracking correlations. A Type 2 decision tree assumes that an AOP correlates both with the Taken and Not Taken outcomes. If an AOP is associated with a node, then it must have both a Taken and a Not Taken leg. But we may wish to learn even these independently. Figure 9 shows a Type 3 decision tree structure. In this structure, each leg in the tree (rather than each node) is associated with an AOP. Each leg is learned independently, thus allowing Taken and Not Taken outcomes to be separately learned.

Notice that the different legs of a single node are no longer mutually exclusive. Since they are completely independent of each other, it is entirely possible for two or more of them to be simultaneously true (for a tree traversal with a given AGBHR).

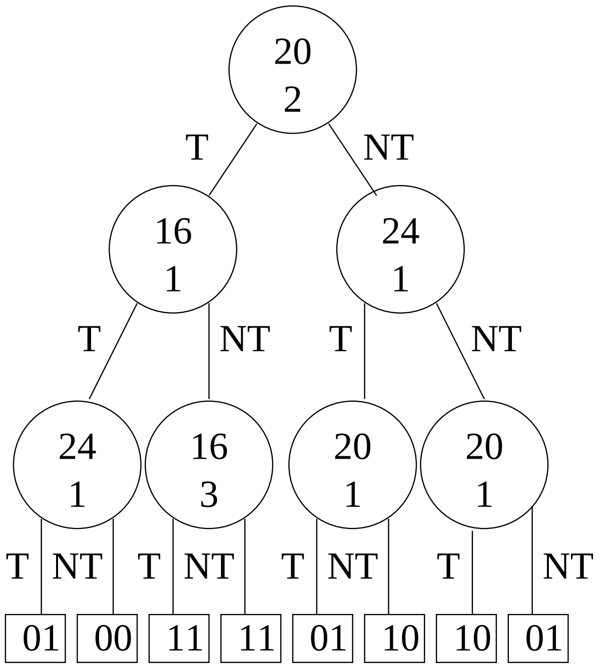

Figure 8: This is a sample Type 2 decision tree. It is similar to the Type 1 decision tree, except that each node is independently learned. This means that each level may consist of nodes with different AOPs, unlike in the Type 1 tree.

In this case, the simple solution is to take the leftmost leg that is currently true (that is, the first one that was created). It is also possible for none of the legs to be true for a given node. This is discussed in the Algorithm section. Additionally, in order to keep the trees bounded, we put some limit on the number of legs that each node can have, as well as a limit on the maximum height of each tree.

The nodes in this tree have a slightly ambiguous meaning, but one could imagine the question at each node being, “Which of the AOP/outcome pairs among the possible legs can be found in the current AGBHR?” To traverse the tree, we start at the root and keep following legs that are in the AGBHR until we reach a leaf, which gives us a prediction.

4.4 Fast Decision Tree Traversal

Each of the above decision trees can be represented by a combined vector of AOPs (for the nodes or arcs) and 2-bit counters. When a branch is encountered, its decision tree (vector of bits) must be fetched from somewhere. Assume that we use a fast cache of decision trees. Given the description of the decision tree, we can combine it with the current execution history (AGBHR) to perform a prediction.

Conceptually, traversing a decision tree involves starting at the

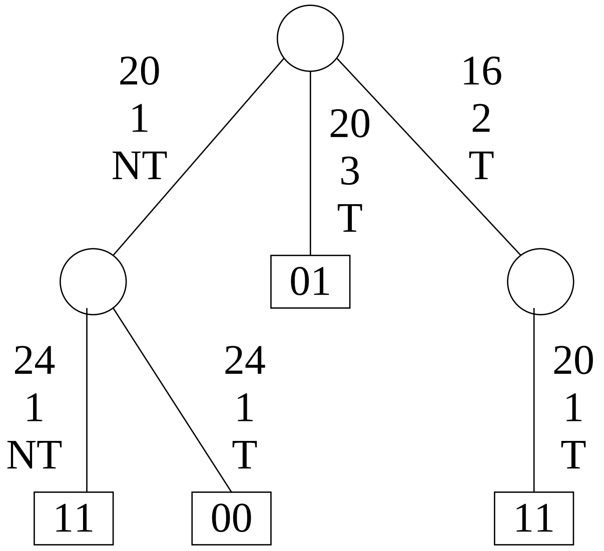

Figure 9: This is a sample Type 3 decision tree. Each leg (rather than each node) is associated with an AOP. In traversing this tree, we simply follow the leg that appears in the current AGBHR until we reach a 2-bit counter.

root, matching and continuing to a child, and repeating until a leaf is reached. However, doing these comparisons sequentially would be time consuming. Instead, all comparisons can be performed in parallel as in Figure 10 (for a Type 3 tree). The description of each arc includes an AOP/result combination which we match against each entry of the AGBHR. Once the comparisons are made, we can identify paths through the tree that lead to leaves, which are two-bit saturating counters. The final prediction is made using the predictions of the valid path(s), as described further in Section 6. The outcome of the branch then modifies (increments or decrements) the counter(s) used. Note that Figure 10 is only representational; an optimized design would likely use dynamic matching logic (treating the AGBHR like a CAM).

5 Algorithm

In this section, we discuss an online learning algorithm to produce good decision trees. Section 6 will show how the introspective processor implements this algorithm.

5.1 Constructing Decision Trees

The tree construction algorithm is similar for all three types of decision trees. Assume that each unique branch has a decision tree associated with it. The decision tree starts out null, and nodes are added as correlations are found between the current

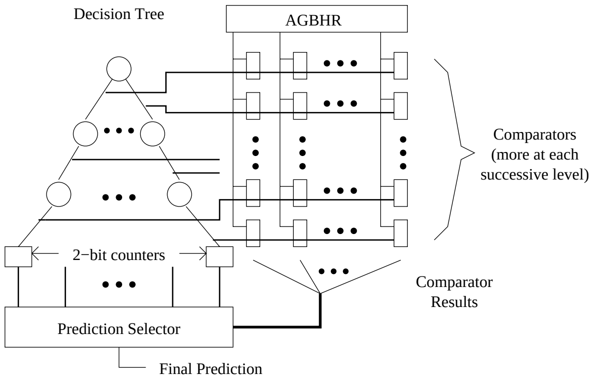

Figure 10: Decision Tree Traversal: Comparators simultaneously compare each leg (or node) against the AGBHR. The results are used in conjunction with the 2-bit saturating counters to make a prediction, as described in detail in Section 6.

branch and the execution history. For instance, suppose that a Taken outcome for a given AOP always coincides with a Taken outcome for the branch in question. If this happens frequently enough (i.e. there is a correlation), we reflect that by adding a node or link to the decision tree.

Type 1 trees are filled in one level at a time (as opposed to one node at a time). When a correlation is discovered, each node in the highest empty level of the decision tree is set to this AOP. That is, if the decision tree is empty, then the root node is set to this AOP; if there is only a root, then both of its children are set to this AOP; etc.

With Type 2 and 3 decision trees, each tracked correlation is additionally associated with a particular empty location in a particular tree (a node for Type 2 trees, a leg for Type 3). When a CFC saturates, we know exactly where to add it. For all three types, the maximum height of the tree is set (in order to facilitate hardware design), so any further learning is discarded. Additionally, for Type 3 trees, the maximum width (number of legs per node) is also bounded. We explore the sensitivity with respect to tree size in Section 7.

5.2 Discovering Correlations

The introspective processor runs in parallel with a program execution on the computation processor and creates a set of decision trees to be used for branch prediction during the same run. Thus, the introspective processor must use an online decision tree learning algorithm to create the decision trees.

A true online learning algorithm would use floating point operations. This is infeasible given that we would like the introspective processor to run in parallel with the computation processor (and not fall behind by doing extremely complex and time-consuming operations). In order to approximate these complex computations, the introspective processor uses Correlation Frequency Counters (CFCs).

A CFC is an n-bit saturating counter. For Type 1 decision trees, each CFC is associated with three things:

- the address of the branch that's being predicted;
- the AOP of a possibly correlating branch in the AGBHR; and
- the outcome (Taken/Not Taken) of the same possibly correlating branch in the AGBHR.

In the case of Type 2 and 3 decision trees, we also associate each CFC with one more thing: a node or leg specification. With these two tree types, we are attempting to learn correlations for each node or leg individually. In order to avoid aliasing between CFCs, we need to keep track of which node or leg each CFC is tracking.

Since each tree is of bounded size, we may label each individual node (or leg) of the complete tree with a different integer. Thus, a "Node/Leg ID" is added to each CFC to specify that the CFC is studying the given correlation exclusively for that particular node (or leg). When a branch executes on the computation processor, we take a look at the current AGBHR and find all the possible places we could add a new node (or leg) to the tree (and still be consistent with the contents of the AGBHR). We then create or increment/decrement all the appropriate CFCs for each of the elements of the AGBHR and for each possible node (or leg). When a CFC saturates, we know exactly where it should be placed in the tree.

Each CFC measures the extent of the correlation between the branch currently being predicted and a recently Taken or recently Not Taken branch. Upon each branch execution on the computation processor, the introspective processor takes the branch address and outcome, as well as the complete AGBHR. Each element of the AGBHR consists of an AOP and a corresponding outcome.

If we take the branch address in question and group it together with any individual AOP and corresponding outcome in the AGBHR, we have specified a unique CFC for a Type 1 tree (since all three elements listed above are specified). (For the other types, we must do this for every possible node or leg in the incomplete tree.) Thus, for a given branch in question and corresponding x-element AGBHR, we are concerned with x unique CFCs. For each of these CFCs, if it already exists, then we simply wish to increment it or decrement it. If the branch in question was Taken, we increment by 1. If Not Taken, we decrement by 1. On the other hand, if the CFC does not exist yet, then it is created and initialized to a central value (2^{n-1} if we are using n-bit saturating counters).

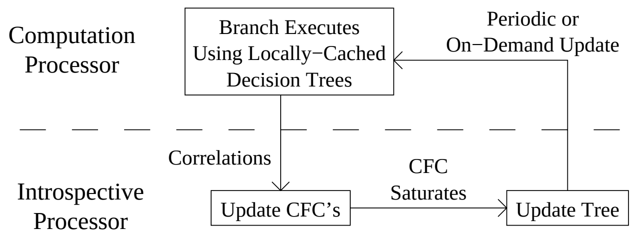

Figure 11: This flowchart shows the basic progression of the algorithms. The computation processor executes branches regularly and passes relevant data to the introspective processor. The introspective processor updates CFCs and adds relevant nodes/legs to decision trees upon CFC saturation. The updated trees are sent to the computation processor (periodically or upon request) to be used for prediction.

Now let us consider what this accomplishes. Suppose that a Taken outcome for a given AOP strongly correlates with a Taken outcome for the branch in question. In that case, the corresponding CFC would be continuously incremented, thus resulting in a saturating value at the high end for the counter. On the other hand, if a Taken outcome for the same AOP strongly correlates with a Not Taken outcome for the branch in question, the corresponding CFC would be continuously decremented, thus resulting in a saturating value at the low end for the counter. The size of the n-bit saturating counter must be chosen to be large enough to avoid accidental saturations but small enough to allow fast learning.

Now we can finally understand exactly what the introspective processor does. When a branch is executed on the computation processor, the introspective processor grabs the address and outcome of the branch in question as well as the complete AGBHR. For each element in the AGBHR, the appropriate CFC is modified as described above (being created if it does not exist yet, being incremented or decremented as appropriate if it does).

5.3 Algorithmic Choices

Figure 11 shows pictorially the basic flow of the algorithms described so far, as well as the interactions between the computation and introspective processors. As mentioned before, various modifications may be made to these algorithms in order to improve effectiveness. Each of these modifications adds performance benefits to the algorithm at the cost of design complexity and additional transistor count (and possibly additional gate delays along the critical path).

Default Leg Consider the fact that, during a particular traversal of the tree, the AOP specified by an encountered node may not appear in the current AGBHR. Likewise, with a Type 3 tree, we may encounter a node during a traversal for which none of the legs appear in the AGBHR. One solution to this problem would be to use another branch prediction mechanism (or to not make a prediction at all) in these instances. However, since this may turn out to be a frequent occurrence, we would like a better way of dealing with this.

A second option would be to designate either the "Taken" or the "Not Taken" leg as the default leg to be traversed. This allows us to use the tree, but it results in rather arbitrary interference on the default leg, so the results are unpredictable.

The third and best option is to add a third leg to each node, called the "Not in AGBHR" leg, to be the default in case none of the other options can be taken. Not only does this avoid interference on either of the other two legs, but it allows the decision tree to attempt to glean patterns from the absence of relevant AOPs in the AGBHR, as well as from their presence. This default leg has been found through testing to be vital to the effectiveness of all of the algorithms, so all of the tests in Section 7 use trees with the "Not in AGBHR" leg.

Multiple Trees per Branch Thus far, we have assumed a limit of one tree per unique branch, but this restriction may significantly limit the algorithm's ability to learn a complex branch's behavior. Obviously, many branches in a program don't need a decision tree at all because they occur less than a dozen times over the entire run of the program (which is not enough time to learn any behavior). Many other branches can be accurately predicted by a single tree. However, there are a select few branches in each program which have a much larger number of instances than the others. These branches are unlikely to be accurately represented by a single tree, and they are likely to pass through several phases or patterns throughout the execution, perhaps correlating with different branches at different times.

For this reason, we should allow for the creation of multiple decision trees per branch. However, we also need to be careful not to create decision trees haphazardly and unnecessarily, as this will tend to deteriorate prediction accuracy. As a result, only one decision tree is created at a time (since we expect a branch to be in a single phase or pattern of execution at any given time), with additional trees being created as needed.

The first tree is created normally as before. Once this tree is done, all CFCs corresponding to that branch in question are cleared, and we start anew. If the next CFC (corresponding to this branch) to saturate is either the root or in the first level of the first tree, then we can safely assume that we are still in the same pattern for this branch (since the same AOPs are most strongly correlating). In this case, we clear all the CFCs corresponding to this branch and start anew again.

However, if a different CFC saturates before any of the previ-

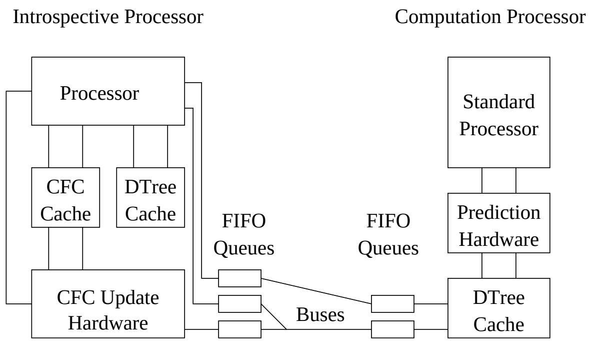

Figure 12: This is a high level view of the system. The computation processor consists of all the standard hardware, as well as a prediction mechanism and a local decision tree cache. The introspective processor consists of the learning hardware, two caches (one for decision trees, one for CFCs) and a processor.

ously most strongly correlated ones, then we have likely entered a new stage of execution, so we create a new tree. This second tree is built using the same techniques as the first one. We continue creating trees like this for as long as is necessary.

6 Implementation

In order to study the feasibility of these algorithms in practice, we must examine the details of a hardware implementation. The hardware of interest consists of the introspective processor, the branch prediction mechanism on the computation processor and the communication mechanism connecting the two (see Figure 12). The introspective and computation processors are preferably embedded on the same chip, thus allowing for fast communication between the two. Each of these components is discussed individually in the following sections.

6.1 Communication

The introspective and computation processors communicate by means of two uni-directional buses. The introspective processor has two incoming FIFO queues corresponding to the two types of messages it can receive. Each time a branch is executed on the computation processor, a message is sent to the introspective processor containing the address and outcome. These messages are all directed into one queue and handled by the CFC-updating hardware, as described below.

The second type of message that may be received by the introspective processor is a request message from the computation processor. If the computation processor misses in the decision tree cache (whether because of a replacement or a cold start miss), it may request that decision tree from the introspective processor. These request messages are given priority and thus a second queue. These are handled as quickly as possible by the introspective software. The appropriate tree is fetched from the introspective processor's cache and is sent to the computation processor.

The computation processor has only a single incoming FIFO queue. Decision trees are periodically sent by the introspective processor to update the computation processor's local cache. This needs to be done because, once the computation processor's local tree cache contains a decision tree for a given branch, the computation processor will no longer need to request it. However, if the introspective processor continues to update that tree, the computation processor's copy could grow stale, so it needs to be updated. Additionally, a decision tree may be sent in response to a direct request made by the computation processor. In both cases, the trees arrive at the same queue on the computation processor and are transferred to the computation processor's decision tree cache.

6.2 Computation Processor

The computation processor has a local decision tree cache. Trees in this cache are periodically updated by the introspective processor, thus keeping the prediction mechanism up-to-date. The size of this cache can be varied, but the number of decision trees used by a program is so small that replacement is rare, and hit rate in this cache is barely an issue.

The computation processor must use branches from its local decision tree cache to make predictions that can be used during a run. This means that it is necessary to make predictions in a single cycle, which necessitates the use of specialized hardware to traverse a decision tree fast enough.

When a branch is encountered, its decision tree is fetched from the local cache. If it's a cache miss, a request is sent to the introspective processor for the desired decision tree. In this case, some other mechanism is used to make a prediction, or perhaps no prediction at all is made. If it's a cache hit, then the tree must be traversed using the data currently in the AGBHR in order to make a prediction.

Conceptually, traversing a decision tree involves starting at the root node, following one of its legs to a child node, and repeating this until a leaf has been reached. Each leg corresponds to some AOP, and that leg should only be traversed if the AOP exists in the current AGBHR. However, doing these comparisons sequentially would take more than a single clock cycle, which would make the prediction useless. Thus, all of the comparisons are performed in parallel by multiple comparators.

Figure 10 gives a sketch of the hardware involved in traversing a tree. We would like to estimate the time involved here. Consider a Type 3 tree with depth three (Figure 13). Further,

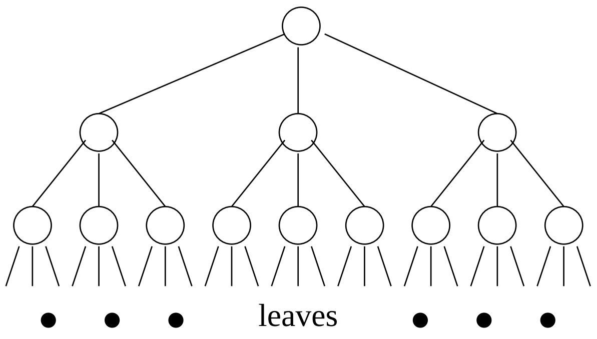

Figure 13: This is a decision tree of height 3 and width 3. The height is determined by the number of non-leaf levels. The width is the number of legs/node. If we are using the default "Not in AGBHR" leg, this is included in the width. The 27 leaves at the bottom are each a 2-bit saturating counter.

assume that each node has two AOP/result arcs to children and one default arc. A quick computation yields $2 + 6 + 18 = 26$ arcs that must be matched against the AGBHR. If we have a 12-element AGBHR, we need $12 \times 26 = 312$ comparators. Each comparator must compare two AOPs. If we are using the low-order 12 bits of each address as well as a 4-bit ordinal, then we have 16-bit AOPs. So we need 312 16-bit comparators in order to perform all of the comparisons in parallel. As suggested before, it would probably make sense to build a tight 12×26 matrix of dynamic matching logic.

Given the comparator results, we can generate a 27-bit vector specifying whether or not each path is valid given the current AGBHR (one bit for each of the 27 possible paths through the tree). Remember that more than one path may be simultaneously valid in a Type 3 tree, so the vector may have more than a single bit set to 1. We now consider the design of the Prediction Selector in Figure 10, which takes the results from the comparators and the MSBs of the 27 leaves (2-bit saturating counters), and which produces a single branch prediction.

The straightforward approach is to take the leftmost valid path and use its prediction. Since legs are added from the left, this will give a highly correlated valid leaf. Figure 14 shows such a design for the Prediction Selector. The 27 valid path bits (comparator results) are passed through a priority encoder which outputs five bits specifying the leftmost valid path. These bits are used as control for a mux to select between the 27 possible predictions (MSBs of the leaves).

The drawback of this approach is that a priority encoder is relatively complicated, and it must execute in series with the mux. Using a standard design for a 27-to-5 priority encoder, we have a gate delay of approximately 8 gates. Likewise, for a 27-to-1 mux, we have a gate delay of approximately 10 gates, for a total delay of approximately 18 gates for the Prediction Selector, which is too much for our single cycle prediction goal.

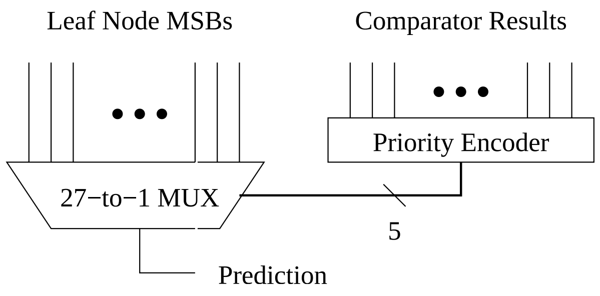

Figure 14: One possible design for the Prediction Selector in Figure 10. The comparator results are used to select from among the possible predictions. A priority encoder is needed since multiple paths may be valid.

We could save some time by using the approximate solution shown in Figure 15. The valid bit for each path is ANDed with the prediction that would be made by the corresponding leaf node (the MSB of the 2-bit saturating counter), so each bit is now 1 if and only if the path is valid *and* the prediction is Taken. These 27 results are ORed together, and this is used as the prediction. Thus, the decision tree predicts Taken if *any* valid path predicts Taken. (It is a simple matter to convert this design into one using only NANDs and NORs.) This design gives us a gate delay of 4 gates.

Tests have shown that predictions made using this approximation differ from predictions made using a priority encoder only 0.009% of the time (averaged across all benchmarks). Thus, we can use this simplified design with its significantly reduced gate delay with almost no loss in prediction accuracy.

Given this design, we can estimate the gate delay for a single prediction. The comparators work in parallel, so they take a total of (say) 4 gate delays. It takes one gate delay to generate the path valid bits and another gate delay to generate the prediction bits for all 27 paths. Determining whether there is at least one high bit among these 27 bits takes another 3 gate delays. Thus, we could have a result in about 9 gate delays or so. More optimization is possible here, and dynamic logic and wider gates (possible in SOI technology) will further reduce the latency for prediction.

6.3 Introspective Processor

The introspective processor is responsible for updating CFCs, adding nodes to decision trees and sending those trees to the computation processor. CFC updates must be performed after each executed branch on the computation processor. This occurs very often (every one, two or three cycles generally), so the CFC update mechanism is implemented in hardware to make the common case fast.

In contrast, the addition of a node to a tree, the sending of a tree

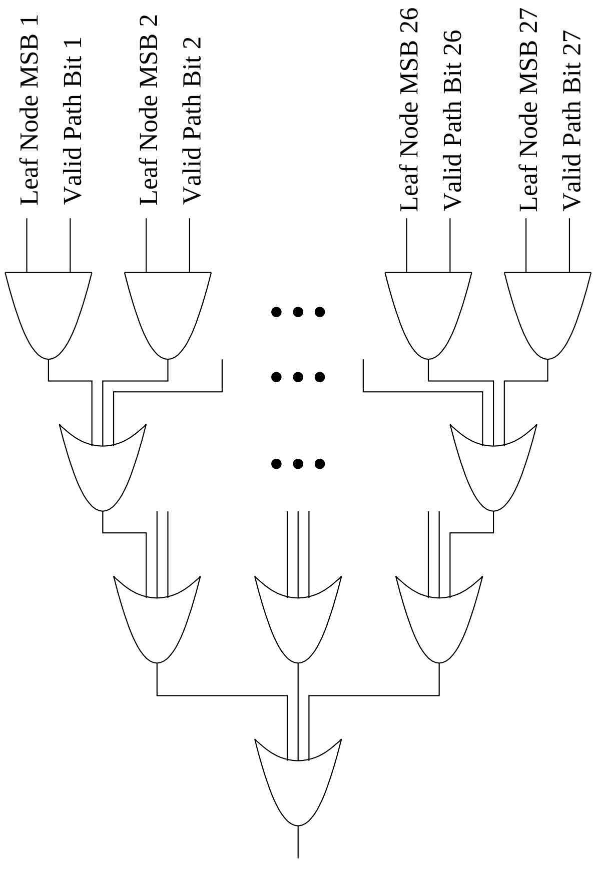

Figure 15: A simple but approximate design for the Prediction Selector in Figure 10. The mechanism predicts Taken if *any* valid path predicts Taken.

to the computation processor and the replacement of a CFC all occur much less frequently, so these are handled by trapping to software. This avoids a good deal of hardware complexity that would be needed to manage these tasks completely in specialized hardware.

Overview Each time a branch is executed on the computation processor, a message is sent to the introspective processor containing the address and outcome of the branch in question. In Section 7, we report the results of investigations into using a certain number of low order bits to identify each branch, rather than the entire address. The end result is that 12 or 16 bits are perfectly adequate for these algorithms on a 32-bit system. However, this is only true for the correlating branches. The branch in question must always be referred to by its complete address, since we don't want to start mixing up decision trees due to aliasing.

The introspective processor must also have access to the complete contents of the AGBHR. However, since it receives each branch and outcome, it can simply keep its own copy of the AGBHR (which is modified each time a branch arrives) without having to receive it in a message. The addresses stored in the AGBHR can use 12 or 16 low order bits of the branch, as mentioned before, in order to reduce hardware requirements. Only the address for which a decision tree is being created must be the complete address.

The introspective processor only needs to keep track of two types of data structures: CFCs and decision trees. One possibility is to store these in one general main memory. However, since these are the exclusive data items that will be handled by the introspective processor, a better idea is to have a separate "CFC cache" and "decision tree cache" (with no main memory). Each of these caches acts as a main memory for a single type of data structure.

CFC Update Hardware The CFC update process must be done in hardware to keep up with the flow of messages from the computation processor. Each time a message arrives from the computation processor, the introspective processor must handle one CFC for each element of the AGBHR. Handling a CFC means either creating a CFC, incrementing an existing CFC by one or decrementing an existing CFC by one. If we use a 12-element AGBHR, this means that 12 CFCs must be found in the CFC cache on each message arrival. This would clearly take too long if all 12 were located randomly throughout the cache.

For this reason, each branch for which a decision tree is being constructed has a contiguous block in the CFC cache allocated to it. When a message arrives, the entire block can be fetched using the address of the branch, and thus all corresponding CFCs can be fetched quickly. Since only one decision tree is being created for each branch at any one time, each branch address has only a single block in the CFC cache.

The size of this contiguous block is of prime importance, since this limits the number of CFCs that may be associated with a branch at any given time. Premature replacements in a CFC block could cause inaccurate or inefficient trees to be built. CFCs are created for each of the 12 elements of the AGBHR for each tree node being studied. Depending upon the complexity of the algorithm implemented (number of legs per node, etc.), the "optimal" size of the block varies. However, even the more complex algorithms perform well with a block size of 128 CFCs, while a block size of 256 CFCs tends to allow trees to be constructed slightly more quickly.

Figure 16 summarizes the parallel CFC Update Hardware. When a message arrives, the introspective processor thus fetches the appropriate CFC block from the CFC cache. Each element in the AGBHR must be found in the block and incremented or decremented (or created if it does not yet exist).

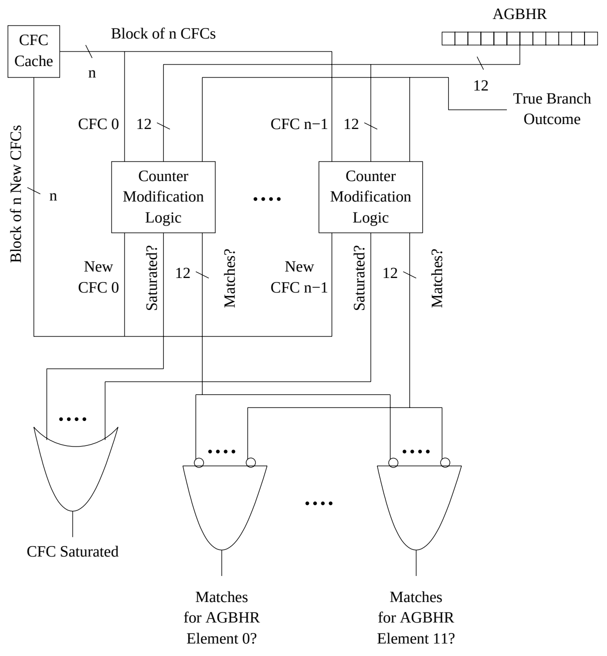

Figure 16: This is an overview of the learning hardware on the introspective processor that handles CFC updates. This hardware is activated each time an instance of a branch execution arrives from the computation processor. An overview of the design of a Counter Modification Logic module is shown in Figure 17.

Each CFC in the fetched CFC block is sent to a separate Counter Modification Logic (CML) module, along with a complete copy of the contents of the AGBHR and the outcome of the branch in question. The inner workings of a CML module are shown in Figure 17.

Each element of the AGBHR is compared against the corresponding information of the CFC. The "path" associated with the CFC represents the path taken from the root node to the node being studied by the CFC. Since the trees are of fixed dimensions (for a given design), the simple solution is to number the nodes of the tree in some order. Thus, the "path" is simply the number of the node in question, so it takes $log(number\ of\ nodes\ in\ tree)$ bits to represent it.

The result of each comparator in the CML is killed (set to 0) if the CFC happens to be invalid (not yet initialized). If any match is found, the CFC counter is incremented or decremented, depending on the outcome of the branch in question.

The Incrementer/Decrementer logic produces the value of the counter (which may be unchanged) as well as a bit signaling whether the counter has saturated. The new value of the counter is put back together with the general information associated with the CFC (which is unchanged), and this is output as the new CFC. Additionally, the results from the comparators are produced as outputs as well.

Thus, each CML module generates three outputs: an updated CFC, a "saturated" bit, and a bus containing the results from the comparators. The updated CFCs from all the CML modules are combined into a block and sent back to the CFC cache. The other two outputs are both used to trap to software in rare cases further described in the next section.

Software Support There are three instances in which we trap to software on the introspective processor:

- request for a tree by the computation processor
- CFC saturation (a node needs to be added to a tree)
- CFC creation (if the CFC Update Hardware cannot find a matching CFC)

If a tree request arrives from the computation processor, the introspective processor fetches the requested tree and sends it off as quickly as possible. If no such tree exists yet (for example, early in the execution of a program), then the introspective processor simply ignores the request. Any return communication regarding the incomplete state of the tree would be complex and unnecessary.

If any of the "saturated" bits are true, the system traps to software in order to handle the adding of a node (or nodes) to the decision tree. This software has access to the saturated bits produced by the CMLs and to the newly updated CFC block. The decision tree is fetched from the cache and the appropriate node is added to the tree, based on the contents of the saturation bits. The tree is then put back into the cache, and some CFCs (those with a path that matches the node that was just added) in the block need to be invalidated. In order to avoid doing 128 sequential fetches and comparisons, these invalidations are done in parallel by hardware.

The third output of the CML modules, the results of the comparators, is used for CFC creation. Each CML module compares each element of the AGBHR with one CFC. Thus, if we want to look at the comparisons of a single element of the AGBHR, we must look at one result from each of the CML modules. For each element of the AGBHR, if no match is found among any of the existing CFCs (see Figure 16), the system traps to software to handle the creation of a new CFC or possibly the replacement of an old one.

For this task, the software has access to the CFC block in question as well as the AGBHR and the results of the comparators. Hardware comparators check all the valid bits in the block in parallel, and then a priority encoder selects out the first invalid CFC. If one exists, the software uses it. If no invalid CFCs exist, then hardware comparators compare against the two high

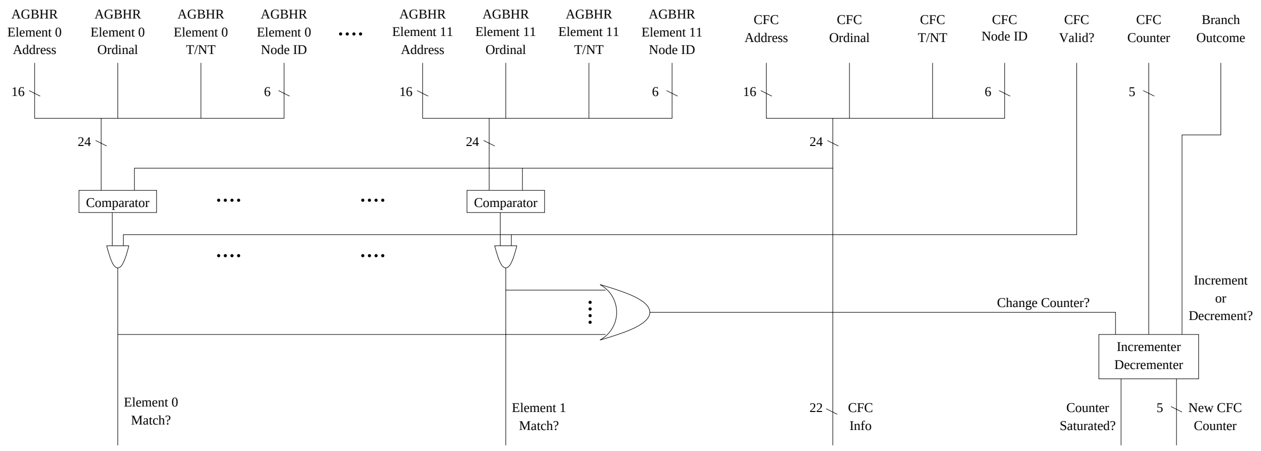

Figure 17: This is an overview of a single Counter Modification Logic module. It is responsible for updating a single CFC and signaling a saturation.

Task	Number of Instructions
Tree Request	4
CFC Saturation	14
CFC Creation	10-38

Table 1: Instruction count for the three software operations performed by the introspective processor.

order bits of each CFC. A CFC whose high order bits are 01 or 10 should not be near saturation (whereas a CFC with high order bits 11 or 00 may be close to saturation). At this point, the software randomly selects one of the possible CFCs for replacement (a CFC is an acceptable candidate if it is valid and has high order bits 01 or 10). Once a candidate is found, the new information is put into place and the CFC block is written back to the CFC cache.

Introspective Processor Utilization The introspective processor can be significantly simpler than the general purpose computation processor, but it must still be capable of keeping up with all incoming requests. We'd like to get an idea of the software complexity of each of the three possible interrupts. Table 1 shows the instruction count for each of these. These counts do not include the time for the hardware assistance mentioned above, so the actual execution time may be increased by as many as 2 clock cycles.

For the simulations reported in Section7, we ran actual code on the introspective processor in order to discover the true load. The end result is that, for all benchmarks tested, the introspective processor is idle for at least 50% of available software cycles. In most cases, usage is around 20-30%. This means that the introspective processor is in fact capable of keeping up with incoming requests. Later, we shall discuss a possible use for these idle cycles.

7 Experimental Results

For our experiments, we used the SimpleScalar Toolset written by Todd Austin [5]. Modifications were made to SimpleScalar by Mark Whitney (UC Berkeley) to allow it to run multiple processors in parallel. Each processor is simply an instantiation of the basic SimpleScalar simulator, with FIFO queues being used for communication between the processors. Delays on the FIFO queues allow simulation of real time communication delay as well as a bandwidth limit on data transferred.

Ten of the twelve SPEC 2000 integer benchmarks [7] have been used in the simulations. Unless otherwise specified, results are reported for runs of the first 500 million branch instructions of each benchmark program. This translates to runs of a few billion instructions total from each program. Numbers were found to converge by this point, so full program runs were unnecessary. The branch prediction mechanism used as the baseline is the YAGS prediction scheme [8].

Unless otherwise specified, each test uses Type 3 decision trees (results for tests using Type 1 or 2 decision trees clearly state this). All of the tests allow multiple decision trees per branch, and all of the tests use decision trees which include the default "Not in AGBHR" legs discussed previously. Unless otherwise stated, each test uses the low order 12 bits of each branch address in the AOPs and AGBHR elements, and decision trees of height three (including the root but not the leaves) and width three (that is, three legs per node, including the "Not in AGBHR" leg).

On the hardware side, the tests use a 12-element AGBHR, a decision tree cache with a maximum capacity of 1000 decision trees on the computation processor, a decision tree cache with a maximum capacity of 10000 decision trees on the introspective processor and a CFC cache with a maximum capacity of 1000 blocks of 128 CFCs each on the introspective processor. Additionally, the introspective processor has two copies of the CFC Update Hardware (so two CFC blocks may be updated independently each cycle), and queue overflow is simply dropped (that is, if branch executions arrive at the introspective processor more quickly than they can be handled by the hardware, the excess branch instances are dropped and not used to learn trees). The introspective processor is always a 2-way superscalar machine and, unless otherwise stated, the computation processor is a 4-way superscalar machine.

7.1 Choosing a Decision Tree

Three types of decision trees of varying complexity have been presented. To determine the actual performance gains obtained for the extra hardware of the more complex trees, these three tree types have been tested using the default environment described above. Additionally, we use a 4-way superscalar computation and a 2-way superscalar introspective processor.

The YAGS branch prediction scheme [8] is used as the baseline for comparison. The "DTrees" scheme refers to a prediction mechanism consisting entirely of the decision tree algorithm described herein. The "DTrees with mini-YAGS" scheme refers to a combination of the two schemes. In this scheme, a meta-predictor is used to select between the DTrees scheme and the regular YAGS scheme. The total number of transistors used in the computation processor implementation is equal in all three schemes. Thus, much smaller YAGS tables are used in the combined scheme to allow for some transistors to be used for DTrees.

Figure 18 shows the relative performance (as determined by total cycle count normalized to the YAGS scheme) for the various schemes for the ten benchmarks. With Type 1 decision trees, the basic DTrees scheme does on average approximately as well as YAGS alone, but the hybrid scheme does show some performance gain over either of the individual ones. With Type 2 trees, the DTrees scheme by itself gets a 4-5% performance improvement over the basic YAGS scheme on average, while the hybrid scheme gets an average of 7-8% better performance. Type 3 trees exhibit the best performance improvement, as expected, gaining by 11% on average using the pure DTrees scheme and by almost 16% on average using the hybrid scheme.

7.2 Parameter Variations

Various tests have been performed to study the sensitivity of the prediction accuracy to the size and shape of decision trees used. The results of these tests have been used to determine realistic values for these parameters in the performance tests,

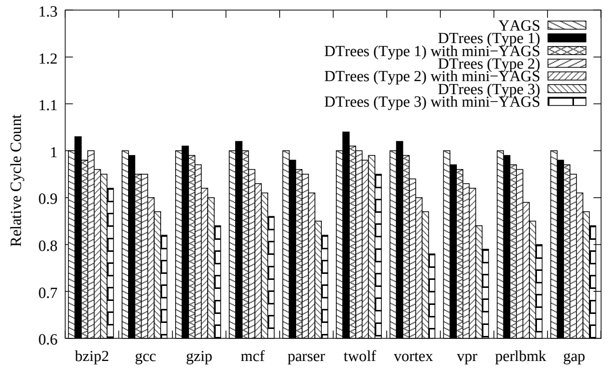

Figure 18: Normalized cycle count for executions using different decision tree types and different algorithms, as compared to the YAGS scheme alone (4-way superscalar computation and 2-way superscalar introspective processor).

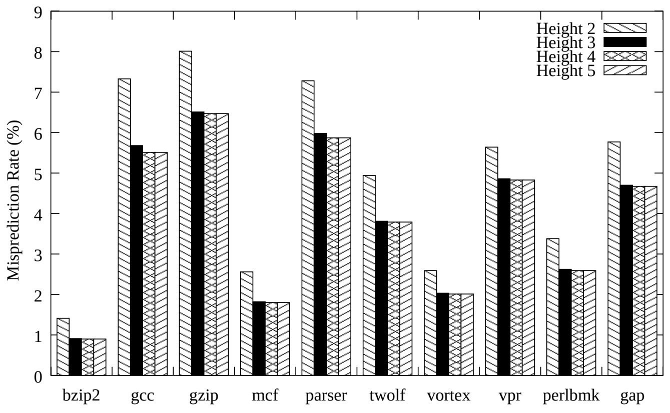

Figure 19: Misprediction rate for the decision tree algorithm (multiple trees per branch) using different maximum heights for the trees (including the root but not the leaves).

as well as to determine the "typical" hardware requirements discussed in Section 6.

Smaller decision trees result in simpler hardware and fewer transistors dedicated to the prediction mechanism. Unfortunately, they also result in decreased accuracy. We would like to minimize the size of decision trees used (both in height and in width) while sacrificing as little prediction accuracy as possible. Figure 19 shows that increasing tree height past three (including the root but not the leaves) does not significantly improve the accuracy of the algorithm. This is true for all ten of the benchmarks.

We also need to consider the width of the tree (the number of legs per node). Remember that one of the legs is the default "Not in AGBHR" leg, so a tree with three legs/node actually has only two distinct correlations per node. Figure 20 shows that slight gains can be achieved by increasing the number of legs/node from three to four, but almost no gain is achieved

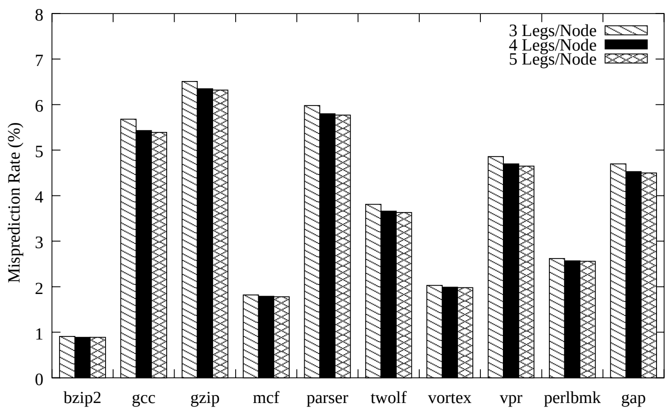

Figure 20: Misprediction rate for the decision tree algorithm using different maximum number of legs per node (including the “Not in AGBHR” leg).

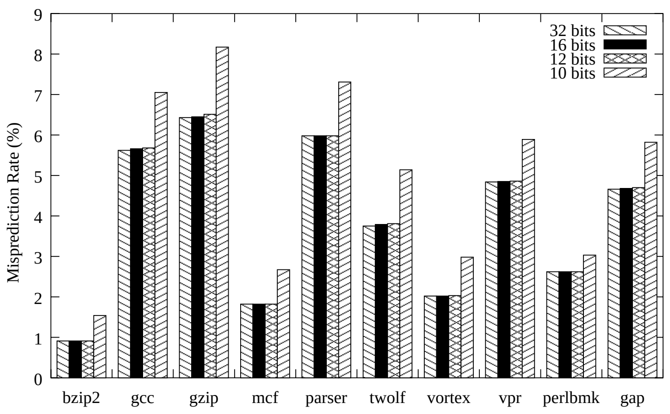

Figure 21: Misprediction rate for the decision tree algorithm using a varying number of address bits per element in the AGBHR and AOPs.

by making the trees any wider. As has been discussed in the Implementation section, the width of the tree very dramatically impacts the number of transistors needed by the prediction hardware on the computation processor. For this reason, a width of three legs/node has been used as the standard across all of the other tests, sacrificing a small amount of accuracy for implementation feasibility.

Another decision that must be made is the number of low order address bits to use in the AOPs. Recall that each decision tree is associated with the complete 32-bit address of some branch (in order to completely eliminate aliasing between trees), but the elements of the AGBHR and the AOPs in the trees use some number of low order bits to identify branches, thus reducing the size of the trees (and the hardware) but allowing some aliasing to occur. Figure 21 clearly shows that the low order 12 bits of each address are adequate to achieve near optimal performance for all ten benchmarks. Note that, since branches are word-aligned, only 10 bits need to be implemented in the hardware if we use the 12 low order bits. This reduces both the storage requirements in the decision tree caches, AGBHR, etc., and the number of comparators needed for the tree traversal hardware.

Finally, we must consider the fact that the best algorithms allow multiple trees per unique branch. In order to determine how many trees are actually being built and thus how large the decision tree caches need to be, we study the percentage of branches in each program that have no trees at the end of the run, that have exactly one tree, and that have more than one. Figure 22 shows the results of this test. Clearly, the vast majority of unique branches have so few instances that they don’t need any tree at all, while a tiny minority of branches (less than 2% in all cases) need more than one tree to be predicted accurately.

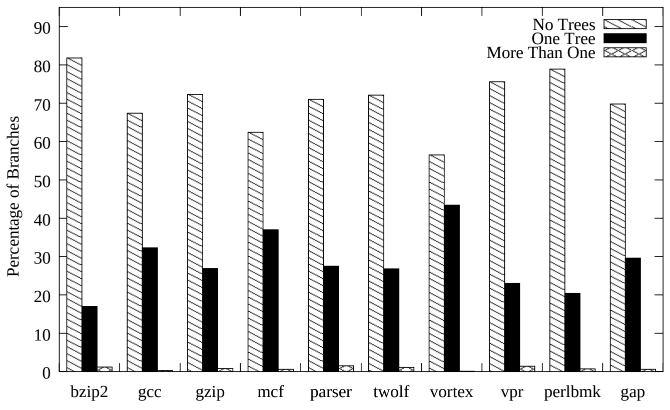

Figure 22: Percentage of unique branches for which no decision trees are built, exactly one tree is built, or more than one tree is built for the decision tree algorithm.

7.3 Varying Hardware Constraints

As the issue width of the computation processor is increased, the prospective gains from effective speculation are increased as well. Tests using Type 3 decision trees were performed thrice, with a 2-way, 4-way and 8-way superscalar computation processor. Since the introspective processor is supposed to be relatively simple, it is a 2-way superscalar machine in all cases. The remaining parameters were set as described earlier.

Figure 23 shows the relative performance (normalized cycle count) using a 2-way superscalar computation processor. DTrees outperforms YAGS on nine out of the ten benchmarks by an average of about 10%. The combined scheme further outperforms the simple DTrees scheme by an additional 5%.

Figures 24 and 25 show similar results for a 4-way and 8-way superscalar computation processor, respectively. Both DTrees and the combined scheme perform relatively better with in-

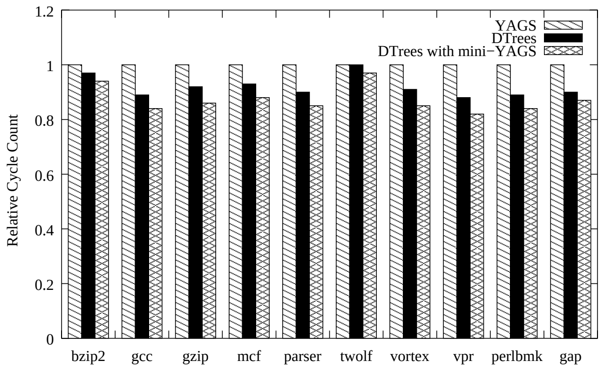

Figure 23: Normalized cycle count for executions using algorithms with Type 3 trees (2-way superscalar computation and 2-way superscalar introspective processor).

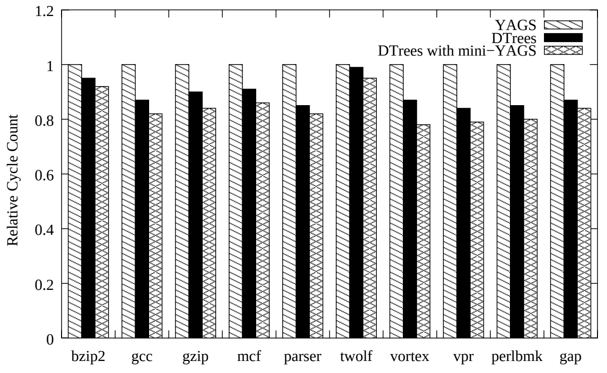

Figure 24: Normalized cycle count for executions using algorithms with Type 3 trees (4-way superscalar computation and 2-way superscalar introspective processor).

creased issue width, since the increased IPC of the wider issue processor means that prediction accuracy has greater impact on use of available resources. In the 8-way simulation (Figure 25), the performance gain of the combined scheme approaches or exceeds 20% for several benchmarks.

All of these tests have been performed on a fixed hardware budget. We also consider how these results are affected by a change in the available hardware. The transistors under consideration are those being used for prediction on the computation processor. The primary use of transistors in the YAGS scheme is in the Pattern History Tables (PHTs), while the bulk of the transistors in the DTrees scheme is in the comparators and in the local decision tree cache. In the combined scheme, the transistors are divided evenly between the two schemes.

Figure 26 compares the three schemes as the hardware budget (approximate transistor count) of the prediction mechanism on the computation processor is varied. Each point represents the average performance across all ten benchmarks. Again, the results have been normalized to the YAGS scheme, and a 4-way superscalar computation processor has been used.

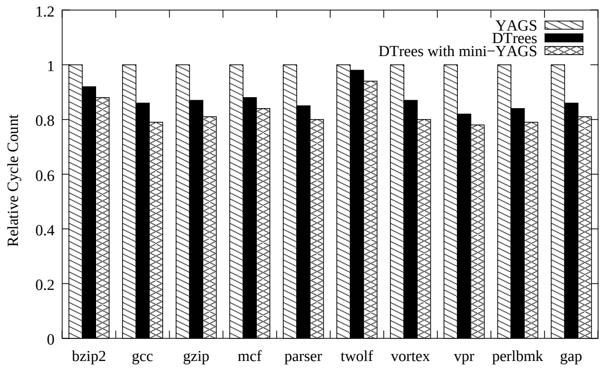

Figure 25: Normalized cycle count for executions using algorithms with Type 3 trees (8-way superscalar computation and 2-way superscalar introspective processor).

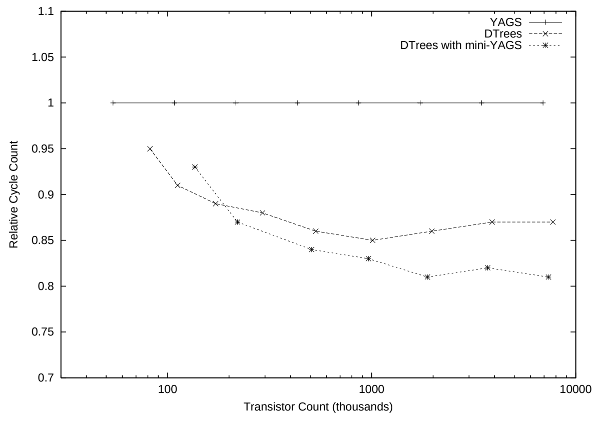

Figure 26: Normalized cycle count for executions using algorithms with Type 3 trees as the hardware budget on the computation processor is varied (averaged across all 10 benchmarks).

As expected, the bare DTrees algorithm needs a certain amount of base hardware before it begins performing well. We need enough hardware for the comparators and a local decision tree cache large enough that it won't be constantly thrashing. For designs of 150,000 transistors or more, the bare DTrees algorithm shows performance improvement (decreased cycle count) of 10-15% over the YAGS scheme. As the hardware on the computation processor is expanded, more complex decision trees and a larger local cache can be included in the design. However, both of these components of the DTrees algorithm begin to exhibit diminishing returns, so additional hardware doesn't necessarily buy further performance gain.

Likewise, the combined scheme does not become effective until we incorporate enough transistors in the design to include smaller versions of both YAGS and DTrees. As expected, more significant performance gains are observed, reaching almost 20% performance improvement for some configurations.

7.4 Analysis

Given these performance results and the general trends described in Section 2, we can now take a look at the overall gains offered by the introspective approach. The prediction hardware on the computation processor uses approximately as many transistors as a corresponding YAGS mechanism would take. On the introspective side, we need to consider the hardware required for the introspective processor and the decision tree and CFC caches. Of course, the transistor count depends heavily on the design of the caches, but a good estimate is that the total transistor count of the introspective portion of the chip is approximately 50-100% of the transistor count of the computation portion. Thus, we are adding 50-100% to the hardware complexity of the chip.

According to the general trends presented in the Introduction, a doubling of the number of transistors on a chip has historically resulted in a 25-30% performance improvement (at least over the last few years). Thus, a 50-100% increase in transistor count (as in our design) would imply a 13-30% increase in performance, depending on the exact design chosen. Our results show a 15-20% performance improvement for most of the benchmarks. However, since these speedups are due to different effects (conventional speedups result from wider issue, improved technology, etc., whereas our speedups come specifically from added parallelism and decreased rollback due to better prediction), gains may differ depending on the exact workload and configuration.

It is important to consider three additional factors when looking at these numbers. First of all, the general trends section looks at increasing transistor counts over time. Successive generations of chips come out every three years or so, which means that the 25-30% performance improvement includes increased clock rate, which is not included in our results. This skews the results in favor of the general trends.

Secondly, as chip complexity is increased, verification difficulty increases superlinearly. For example, even though the Intel P4 has about twice as many transistors as the PIII, it is more than twice as difficult to verify such a complex design. However, our design involves two smaller processor which can be verified individually (although this is somewhat offset by the additional verification which needs to be performed on the interface between the two processors). This means that, for similar gains, the introspective approach will more readily be realized than will the standard approach.

Finally, much of the introspection work is done by the specialized hardware on the introspective processor (the CFC Update Hardware). As mentioned earlier, the actual processor runs software for less than 50% of its available cycles for all benchmarks, much less in some cases. This gives rise to the possibility that multiple introspective algorithms could use the same introspective processor (for example, branch prediction and memory prefetching). Of course, more research needs to be done to be certain that this can be accomplished feasibly, but the strong possibility exists that the cost of the introspective processor could be amortized over multiple algorithms.

8 Future Work

The prime area of further research on this topic is the expansion of the system state being studied by the introspective processor. Currently, only the contents of the AGBHR are used to learn decision trees. This limits us to correlations between branches. However, it may very well be that some branches are highly correlated with data values, and thus inclusion of the register set in available system state would improve the accuracy of the decision trees constructed.

Additionally, decision trees or other graphical models could be used to learn patterns other than branch outcomes. Introspection could be used to study the state of the system and build graphs that can effectively predict data values, future memory accesses, etc. The basic idea remains the same however: study portions of present system state and build models that can accurately predict portions of future system state.

9 Conclusion

With the number of transistors per chip constantly rising, we are reaching the point where more complicated uniprocessor designs are simply not the best way to achieve improved performance. We have presented an argument for introspective computing as an alternative use of these abundant transistors. The introspective computing model involves devoting processor cycles for continuous, online analysis and adaptation.

As one possible use of this model, we devote introspective resources to perform run-time optimizations such as branch prediction. We have shown that one of the simplest graphical structures, decision trees, can be effectively used to characterize the behavior of a vast majority of branches. We have shown how these trees can be learned and constructed by an introspective processor in a timely manner, then used on the computation processor for accurate single-cycle predictions in the same run. The resulting gain in performance (more than 20% in some cases) is not unlike that which results by doubling the number of transistors while widening an existing superscalar design. Perhaps introspection will ultimately prove to be a better way to achieve such gains.

References

[1] IBM autonomic computing research web page. http://www.research.ibm.com/autonomic.

[2] IBM power 4 chip multiprocessor. http://www.research.ibm.com/journal/rd46-1.html.

[3] Transmeta corporation crusoe processor. http://www.transmeta.com.

[4] Todd Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, Nov 1999.

[5] Doug Burger and Todd M. Austin. The Simplescalar Tool Set Version 2.0, 1997.

[6] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. In *ACM Transactions on Programming Languages and Systems, 19(1)*, 1997.

[7] Standard Performance Evaluation Corporation. SPEC 2000 benchmarks, 2000.

[8] A.N. Eden and T. Mudge. The YAGS branch prediction scheme. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, Nov 1998.

[9] Marius Evers, Sanjay J. Patel, Robert S. Chappell, and Yale N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[10] A. Fern, R. Givan, B. Falsafi, and T. Vijaykumar. Dynamic feature selection for hardware prediction. In *Technical Report TR-ECE 00-12, School of Electrical and Computer Engineering, Purdue University*, 2000.

[11] Dirk Grunwald, Artur Klauser, Srilatha Manne, and Andrew Pleszkun. Confidence estimation for speculation control. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[12] Lance Hammond, Ben Hubbert, Michael Siu, Manohar Prabhu, Mike Chen, and Kunle Olukotun. The stanford hydra cmp. In *Proceedings of Hot Chips 11*, Aug 1999.

[13] Daniel A. Jimenez, Stephen W. Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, Nov 2000.

[14] Daniel A. Jimenez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, Jan 2001.

[15] C.-C. Lee, C.C. Chen, and T.N. Mudge. The bi-mode branch predictor. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, Nov 1997.

[16] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: A modular reconfigurable architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[17] S. McFarling. Combining branch predictors. Technical Report Technical Report TN-36, Digital Western Research Labs, June 1993.

[18] T. Pering and R. Broderson. Dynamic voltage scaling and the design of a low-power microprocessor system. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[19] S. Sechrest, C.-C. Lee, and T.N. Mudge. Correlation and aliasing in dynamic branch predictors. In *Proceedings of the 23rd International Conference on Computer Architecture*, May 1999.

[20] E. Sprangle, R.S. Chappell, M. Alsup, and Y.N. Patt. The agree predictor: A mechanism for reducing negative branch history interference. In *Proceedings of the 24th International Conference on Computer Architecture*, June 1997.

[21] E. Waingold, M. Taylor, D. Srikrishna, V Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, Sep 1997.

[22] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[23] Cliff Young, Nicolas Gloy, and Michael D. Smith. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[24] Cliff Young and Michael D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.