Control Flow Speculation in Multiscalar Processors

Quinn Jacobson
Electrical & Computer
Engineering Department
University of Wisconsin
qjacobso@ece.wisc.edu

Steve Bennett[1]
Measurement, Architecture
and Planning
Intel Corp., Hillsboro, OR
sbennett@ichips.intel.com

Nikhil Sharma[1]
Design Verification Unit
Synopsys Inc.
Mountain View, CA
nikhil@synopsys.com

James E. Smith
Electrical & Computer
Engineering Department
University of Wisconsin
jes@ece.wisc.edu

Copyright 1997 IEEE. Published in the Proceedings of the Third International Symposium on High Performance Computer Architecture, February 1-5, 1997 in San Antonio, Texas, USA. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

Control Flow Speculation in Multiscalar Processors

Quinn Jacobson
Electrical & Computer Engineering Department
University of Wisconsin
qjacobso@ece.wisc.edu

Steve Bennett[1]
Measurement, Architecture and Planning
Intel Corp., Hillsboro, OR
sbennett@ichips.intel.com

Nikhil Sharma[1]
Design Verification Unit
Synopsys Inc.
Mountain View, CA
nikhil@synopsys.com

James E. Smith
Electrical & Computer Engineering Department
University of Wisconsin
jes@ece.wisc.edu

Abstract

The Multiscalar architecture executes a single sequential program following multiple flows of control. In the Multiscalar hardware, a global sequencer, with help from the compiler, takes large steps through the program's control flow graph (CFG) speculatively, starting a new thread of control (task) at each step. This is inter-task control flow speculation. Within a task, traditional control flow speculation is used to extract instruction level parallelism. This is intra-task control flow speculation.

This paper focuses on mechanisms to implement inter-task control flow speculation (task prediction) in a Multiscalar implementation. This form of speculation has fundamental differences from traditional branch prediction. We look in detail at the issues of prediction automata, history generation and target buffers. We present implementations in each of these areas that offer good accuracy, size and performance characteristics.

Keywords: Multiscalar Architecture, Control-flow Speculation, Multi-way Branch Prediction, Target Buffer

1 Introduction

Traditional processor architectures execute sequential programs following a single flow of control. To achieve high performance, both superscalar and VLIW implementations of these architectures build a large window of instructions and attempt to issue multiple independent instructions per clock cycle. The instruction window is built by the hardware (superscalar), the compiler (VLIW) or both by speculating a single flow of control and rearranging instructions to increase parallel execution. The ability to build a large window is limited by the requirement that control flow speculation (branch prediction) be done sequentially. Sequential program flow limits the speed with which the window can be built, and, incorrect predictions limit the useful size of the window because instructions following an incorrect prediction are nullified.

Multiprocessing and multi-threaded architectures achieve high performance by executing many instructions in parallel from multiple flows of control. Programs must be written in a parallel programming style (which continues to be a difficult problem), or must have parallelism extracted by the compiler (such as loop level or vector parallelism) into mostly *independent* sub-programs. Dependences between these flows of control must be handled explicitly by the programmer or the compiler. This approach does not work well for many non-numeric applications because (1) the parallelism present in the program is difficult to detect during program creation or compilation, (2) the program is written in a language such as C which makes compiler analysis difficult, or (3) the granularity of communication is too small to be handled efficiently by the compiler or hardware.

The Multiscalar architecture [4][6][14] executes a single sequential program following multiple flows of control. It relies heavily on hardware support to maintain sequential semantics while executing a program in a parallel fashion. This allows the architecture to execute existing codes efficiently and allows the programmer to use a natural, sequential programming model. The architecture provides a uniform memory space, single logical register file and sequential semantics. To achieve high performance the Multiscalar hardware uses two levels of control flow speculation. This multi-level speculation model allows the hardware to:

- build a very large effective instruction window from a sequential program,
- build this window very quickly, and
- maintain the large window in the presence of some miss predicted branches.

In the Multiscalar hardware, a *global sequencer*, with help from the compiler, takes large steps through the program's control flow graph (CFG) speculatively, starting a new thread of control at each step. Each of these steps requires the implicit prediction of possibly many branches. We call this *inter-task* control flow speculation and refer to the group of instructions between steps as a *task*. The large

1. This work was done while the authors were at the University of Wisconsin.

steps in the control flow increase the effective window size, thereby increasing the chance of finding independent instructions which may be executed in parallel.

Within a task, traditional control flow speculation is used to extract instruction level parallelism. We call this *intra-task* control flow speculation. The model allows speculation within a task to be imperfect without necessarily interfering with the higher level inter-task speculation. The next task may be control independent of some or all of the control flow of preceding tasks.

The focus of this paper is the development of mechanisms for inter-task control flow speculation in a Multiscalar implementation. This form of speculation has some fundamental differences from traditional branch prediction. We look in detail at the issues of prediction automata, history generation and target buffers. We present implementations in each of these areas that offer good accuracy, size and performance characteristics.

2 Overview of Multiscalar Processors

2.1 Multiscalar Executable

A Multiscalar instruction set architecture (ISA) is similar to other ISAs. However, a Multiscalar ISA adds special instructions to communicate information from the software to the hardware to support the two level sequencing model. A multiscalar executable consists of tasks which are encapsulated groups of instructions that may contain arbitrary control flow. The task is bounded by task start and task end instructions. The task start instruction loads a *task header* into special state registers. The task header contains a bit mask indicating which registers may have new values created within the task and contains information about tasks that may succeed it. A task must end in a control transfer instruction which has special bits to indicate that it is a task exit point. The target address determined by the exit instruction is address of the task start instruction for the next task.

At a high level, program execution may be viewed as traversing a task flow graph (TFG). Figure 1 shows an example TFG. A TFG is a directed graph with tasks at nodes and the arcs representing control flow between the tasks. A TFG is analogous to a control flow graph (CFG) built from a scalar executable. Each task is a traditional CFG.

There is no bound on the number of potential exits that a task can have. Our implementation limits the number of exits in the header to four. For each exit the header contains three pieces of information about the control flow.

- *Exit Specifier*: The type of control instruction that exits the task. This is one of the control flow types detailed in Table 1. This information is encoded in 5 bits.
- *Target Address*: If the target address of the exit instruction is known (as it is in BRANCH and CALL exits) this field contains the actual target address of the instruction. Otherwise this field is left null by the compiler. In our environment, addresses are 32 bits.
- *Return Address*: If the exit instruction is a CALL or INDIRECT_CALL, the value in this field is the address executed after the called routine returns. It may be pushed onto a return address stack by the hardware.

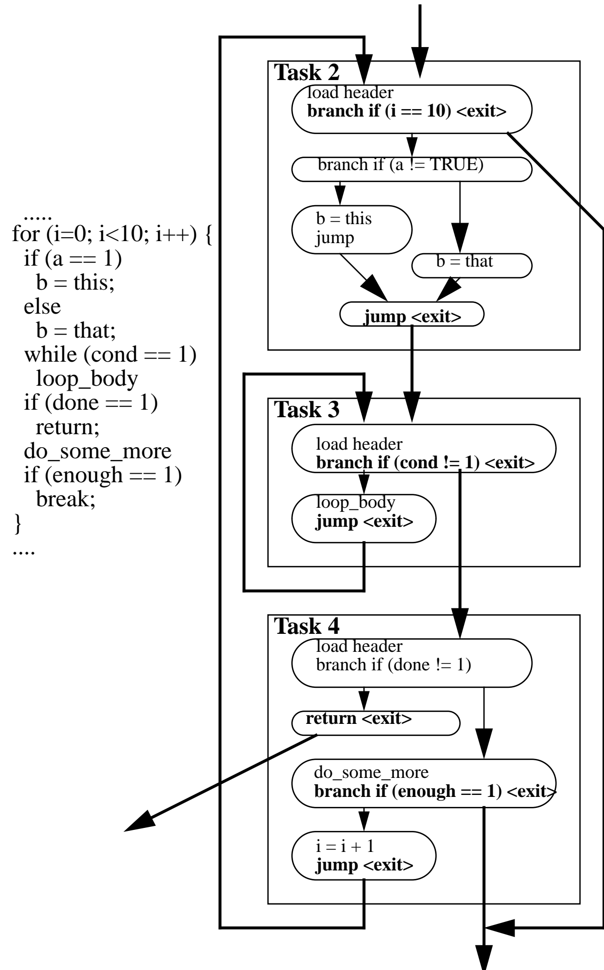


Figure 1 Example Task Flow Graph

Control Flow Type Terminating Task	Corresponding Scalar Instruction	Target Known at Compile Time	Number of Targets	Target Address Prediction
BRANCH	(un)conditional branch	Yes	1	Easy
CALL	call (PC relative)	Yes	1	Easy
RETURN	return	No	unlimited	Harder
INDIRECT_BRANCH	indirect branch	No	unlimited	Hard
INDIRECT_CALL	call (indirect)	No	unlimited	Hard

Table 1 Multiscalar Inter-task Control Flow Types

The exit instructions of a task must be control transfer instructions; conditional branches are only exits when they are taken. Each TFG arc is also a CFG arc of the underlying program. In our implementation, control transfer instructions contain three bits of exit information. One bit specifies that the instruction is an exit. The other two bits associate the exit instruction with one of the four exit points specified in the header. This last information is needed to update the next task predictor. Each exit instruction is classified as one of five types, as detailed in Table 1.

2.2 Multiscalar Processor Hardware

A high level view of a Multiscalar implementation is shown in Figure 2. The Multiscalar hardware employs two levels of sequencers to build a large window of instructions in the machine and extract parallelism from an inherently sequential program. At run-time, the Multiscalar *global sequencer* traverses the program's TFG. Tasks are distributed to processing units after predicting the path the program will follow through the TFG. The processing units are arranged in a ring. At any time, one unit will be executing the non-speculative "current" task and the other units in the ring will be executing speculative tasks. The ring operates as a circular queue with a head and a tail pointer (pointing to the current non-speculative task and the most recently started speculative task, respectively). Tasks commit in strictly FIFO order.

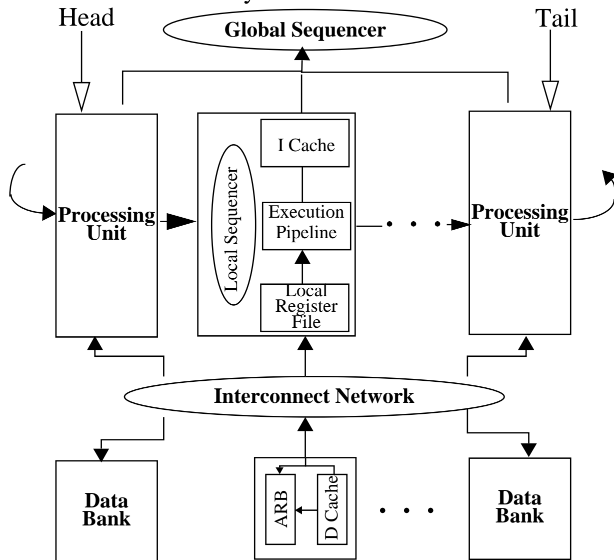


Figure 2 The Multiscalar Hardware

Hardware provides for the synchronization and the forwarding of data around the ring. Both register and memory state have to be communicated to maintain sequential semantics across a single address space and register file. The hardware to perform this has been discussed in other publications [1][5].

The global sequencer does not examine each instruction in a task before predicting the next task; rather it predicts the starting address of the next task to be executed using information from the task header of the most recently predicted task and dynamic prediction hardware. This predicted task is launched on the next free processing unit. When the task at the head of the processing unit queue completes, it informs the global sequencer of its actual, non-speculative target address. If the target address predicted to follow the head task was incorrect, all tasks behind the head are *squashed* (all work performed is discarded) and execution is redirected to the correct task. The task misprediction penalty, in terms of lost potential work, can be large.

Alternatively, a hardware implementation could have a global sequencer that predicts tasks without a header. Removing the header would minimize the impact to an instruction set for supporting the Multiscalar execution model. The hardware compensates for the lack of header information by building up history. This option will be discussed in Section 5.4.

Within a task each processing element may predict the outcome of control flow instructions within the task's CFG and redirect its pipelines using speculative execution. This intra-task control flow speculation is similar to conventional scalar control flow speculation. In practice, however, there is a difference between intra-task speculation and conventional scalar control-flow speculation. In a Multiscalar processor, the individual processing elements do not see the whole dynamic instruction stream. They have a local, incomplete view of the code which led to the currently executing instructions. This may hurt dynamic intra-task prediction accuracies. The predictor used for intra-task prediction in our current Multiscalar simulators is a bimodal predictor which only suffers minimal accuracy loss due to incomplete history. We do not look into the matter of intra-task speculation any further in this paper. Rather, we focus on the job of inter-task speculation which requires new mechanisms.

3 Methodology

3.1 Simulator

Our results are obtained using a multiscalar functional simulator and compiler developed at the University of Wisconsin. The focus of the research reported here is task prediction accuracy, not overall Multiscalar execution time. However, in the conclusion we do report some overall performance numbers, reflecting the impact of task prediction. The overall performance numbers are obtained from a detailed timing simulator which takes into account speculative state.

There are a two major issues we do not take into account in the functional simulator.

- *Update Timing*: Updates of dynamic data structures for

prediction are made immediately after prediction (there is no delay). A real implementation may make predictions based on stale information while waiting for non-speculative outcome information to return from the execution processors.

- *Pollution*: Our functional simulator does not continue past a mispredicted task, therefore no pollution of dynamic data structures for prediction occurs because of speculative updates from mispredicted tasks. Our results are accurate in this regard if the mispredict recovery mechanism completely repairs data structures modified after a misprediction.

3.2 Benchmarks

The characteristics of tasks are dependent on the compiler heuristics used to break a program into tasks. The accuracy of task prediction is therefore dependent upon the compiler. In our experience though, the relative performance of predictors is very consistent across different benchmarks and compilations.

Benchmark	Input File	Static Tasks	Dynamic Tasks	Distinct Tasks Seen
gcc	stmt.i	12525	4036539	3164
compress	in (1MB)	103	5517241	39
espresso	bca.in	3788	41458206	1260
sc	loada3	3744	8353930	575
xlisp	li-input.lsp	1756	2735019	522

Table 2 Benchmarks, Inputs and Task Information

We used five of the SPEC92 integer benchmarks to evaluate various task prediction schemes. Table 2 lists these benchmarks, their inputs, and task level behavior. Note that the number of distinct tasks is rather small for all benchmarks except *gcc*. This large working set of tasks makes inter-task prediction difficult in *gcc*.

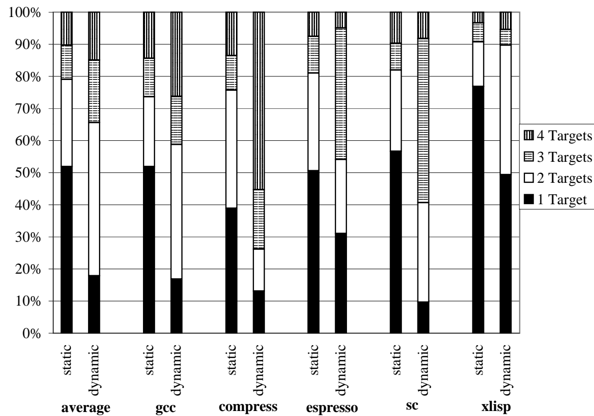


Figure 3 Number of Exits per Task

Figure 3 and Figure 4 show the static and dynamic makeup of task exits. Recall that the current Multiscalar implementation allows up to four exits per task. We see in Figure 3 that most tasks have fewer than four exits, many having only a single exit. This is encouraging, because tasks with only a single exit are easy to predict. Each exit is classified according to control-flow type in Figure 4.

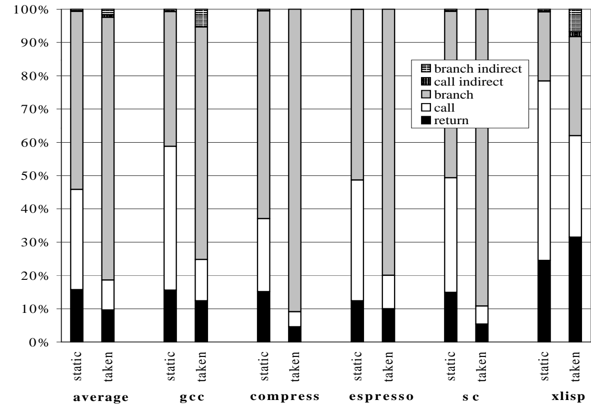


Figure 4 Types of Exit Instructions

4 Previous Work

There has been a wealth of work in scalar branch prediction that we will borrow from. We briefly outline this previous work in this section.

4.1 Branch Prediction

Two-level branch predictors were proposed by Yeh and Patt [17] and Pan, et al. [12] and have been implemented in commercial processors such as the Pentium Pro [7]. A high-level view of these predictors is shown in Figure 5. The branch address indexes (through some hashing function) into the History Register Table (HRT) which is a table of one or more *history registers*. The value from the HRT is combined (though another hashing function) with the branch address to form an index into a Pattern History Table (PHT). The value in this table is a prediction automaton which determines the prediction made. The structures are updated both speculatively (history registers) and non-speculatively (prediction automata).

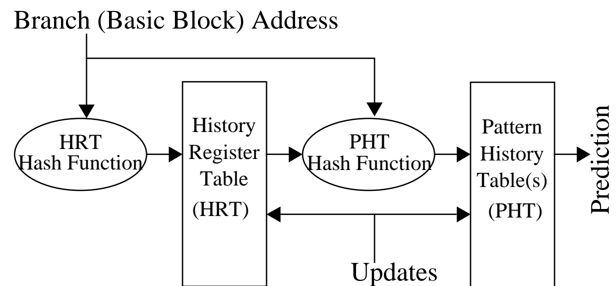


Figure 5 Two-level adaptive prediction mechanism

The HRT hash function normally extracts bits from the branch address to index the HRT. The PHT hashing function may be a simple function concatenating bits of the address with bits from the history table, or some folding scheme with exclusiveOR may be used to reduce alias-

ing. [10]

4.1.1 Prediction Automata

Most branch prediction applications have used 2-bit saturating counters. The 2-bit counter encodes two pieces of information, prediction (taken or not taken) and bias (strong or weak). This bias provides a form of hysteresis, so that a prediction that has been correct for successive executions of a branch will not be changed after a single misprediction.

4.1.2 History Generation

In two-level dynamic branch predictors, there are one or more history registers that are maintained by shifting in some representation of the predicted outcomes of branches. We call this process *history generation*. A number of history generation methods have been investigated in the literature:

- *exit-based*: After each prediction, the predicted branch outcome is shifted into the low order bit of the history register. This single bit represents taken or not taken. Conceptually, this single bit represents the exit taken from the branch instruction, and we call this method *exit-based*. This method was used by Yeh and Patt and Pan, et al.
- *path-based*: After each prediction, some bits of the target address of the branch are shifted into the low order bits of the history register. This creates a more accurate representation of the *path* followed to reach a certain point in the program execution. Hence, predictors that use this method are called *path-based*. This approach was studied in [11].

4.2 Branch Address Prediction

In addition to predicting whether a branch is taken or not taken, it is important to be able to determine quickly the address of the taken path. To facilitate smooth pipelining of successive fetches in the presence of control flow changes, a number of mechanisms have been proposed and implemented in commercial processors:

- A *dedicated adder* in the fetch mechanism can be used to compute PC relative target addresses before they are computed by the ALU(s) in the execution engine.
- A *branch target buffer* (BTB) [9][13] stores addresses of conditional branch targets. A BTB is simply a cache memory. Typically, the BTB indexing function is similar to other cache structures, utilizing only bits from the branch address. Indirect addresses may be cached in the BTB in addition to PC-relative addresses.
- A *return address stack* (RAS) [8] may be used for return address prediction. In Multiscalar processors, as in scalar processors, a reasonably deep RAS is nearly perfect in predicting return addresses [2].

Microarchitectural structures have been proposed that combine exit and target address prediction to facilitate pipelined implementations [16][3].

5 Adapting Branch Prediction to Tasks

In applying scalar dynamic branch prediction to intertask prediction in Multiscalar processors, we equate Multiscalar tasks to scalar basic blocks and leave the overall structure of the predictor mechanism unchanged. However, there are a number of issues that arise when we perform such a straightforward application of the ideas.

5.1 Prediction Automata

The saturating counters used to make scalar branch predictions are not sufficient to predict tasks in Multiscalar processors because there may be more than two exits. Predicting the exit taken out of the four possible exits is a multi-way branching problem. There are a number of alternatives with which to replace the saturating counters in the PHT:

- *voting counters (VC)*: At each entry in the PHT there is a saturating counter for each exit. If the counter corresponding to one exit is greater than all others, we predict this exit. If there is a tie for the highest counter value, we pick either (1) the most recently used (MRU) exit among the ties, or (2) randomly among the ties. Note that the MRU method requires additional storage and implementation complexity. We investigate each of these methods below. When the actual exit is known, the counter corresponding to this exit is increment, while the others are decrementing. We investigate the use of both 2-bit and 3-bit counters.
- *last exit* (LE): Each entry in the PHT records the exit taken the last time this entry was accessed; we predict this exit the next time this entry is accessed. When the actual exit is known, it is stored in the PHT. Note that LE is really just a degenerate case of VC, with each counter being one bit. Storing a simple exit number reduces storage costs, however.
- *last exit with hysteresis* (LEH): The last taken approach can be augmented by adding a small (1 or 2 bit) saturating counter that is incremented on correct predictions and decremented on incorrect predictions. The prediction is replaced when the counter is zero and the prediction is wrong. This provides a mechanism for keeping a prediction that was correct multiple times from being replaced too quickly.

Figure 6 compares the performance of seven different automata used in conjunction with a very aggressive path-based predictor (discussed later). All the benchmarks had similar relative performance for the automata so we only present numbers for *gcc*. The seven approaches stratify into three performance curves. For all the benchmarks the last exit (LE) approach has the highest miss rate. Both 2-bit voting counters (VC MRU and VC RANDOM) and the

last exit with 1-bit of hysteresis (LEH 1-bit) perform nearly identical and are indistinguishable in the graphs. All three have comparable degrees of hysteresis; replacing a proven prediction only after two miss predictions. Both 3-bit voting counters (VC MRU and VC RANDOM) and the last exit with 2-bits of hysteresis (LEH 2-bit) perform nearly identical and have the lowest miss rates for all the benchmarks. Since LEH uses fewer bits than VC, we use the LEH-2 bit automaton in all following studies.

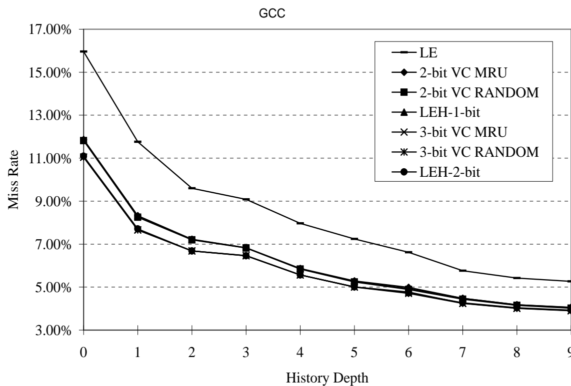


Figure 6 Comparison of Prediction Automata

5.2 History Generation

The history generation methods of branch prediction have to be modified slightly to support task prediction:

- *Exit-based history generation*: Shifting one bit to encode branch behavior does not work because in task prediction there are up to four possible outcomes of each task sequencing step. The modification is very straight forward: we shift two bits instead of one to encode which of the four possible exits was predicted.
- *Path-based history generation*: Similar to branch prediction, we shift in some of the low order bits of the target address of the predicted task.

There is some difference in the information that the history in task prediction represents. In branch prediction the history at the time of a prediction represents the most recent control-flow behavior. In task prediction the history is really disconnected pieces of control-flow information from various points in the past. The history most likely does not include the control-flow immediately before the point where the prediction of the next task is made because this piece of control flow is buried inside the task. We found that despite this behavior, there is still correlation in paths taken through the TFG.

We examine the prediction accuracies that can be achieved using the various history generation schemes adapted from branch prediction. The prediction schemes we compare are:

- A *Global Exit History Scheme (GLOBAL)* where there is a single history register shared by all tasks. The history register is generated by shifting in two bits which encode the predicted exit.
- A *Per-Task Exit History Scheme (PER)* where there is one history register and one table of prediction automata per static task. This scheme is analogous to Yeh's PAp configuration [15]. Each history register records which of the four exits was taken for the previous invocations of the particular static task. In a real implementation a finite number history registers and tables would be associated with tasks by some hashing function; approximating but not guaranteeing a 1-to-1 relationship between tasks, history registers and pattern tables.
- A *Path-based History Scheme (PATH)* where there is a single history register shared by all tasks. We discuss later how to encode the identity of a task.

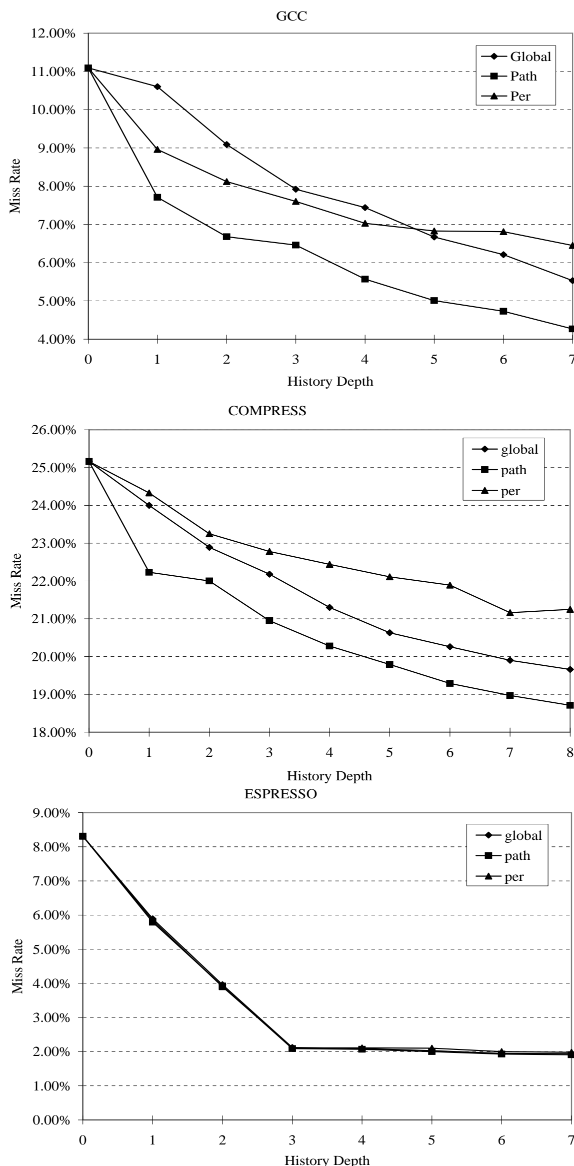
There are no doubt other approaches to history generation, but we limited our study to these three which represent the dominant approaches in branch prediction.

To compare these history generation methods, we simulated ideal implementations of the three approaches. We define ideal to mean there is no aliasing in any of the data structures. Aliasing occurs when the predictor can not distinguish between two different cases because of limited information. For GLOBAL, an ideal implementation means that the entire history register is used for indexing into the table of prediction automata, and the PHT is large enough to associate a unique entry with every possible history value. For PER, an ideal implementation means that there is a unique history register and a unique table of prediction automata for every static task. Each PHT has the same properties as in the ideal global scheme: it is large enough so that every history register value indexes to a unique PHT entry. For PATH, an ideal implementation can uniquely identify the path that led to the current point in the TFG and can associate a unique prediction automaton with this path.

Figure 7 shows the prediction miss rates for the three history generation schemes. The schemes are compared over a range of history depths. Note that a history depth of zero is equivalent to associating a single prediction automaton with each static task; no correlation is exploited. PATH always performs better than GLOBAL. GLOBAL has a 30% and 50% higher miss rate for *gcc* and *xlisp* respectively for a history depth of 7. For the other three benchmarks the difference is smaller: from 2% to 5%. PATH outperforms PER on 4 out of the 5 benchmarks. PER has a miss rate which is 51%, 12% and 15% higher than PATH for *gcc*, *compress* and *xlisp* respectively at a history depth of 7. The difference is only 4% for *espresso*, reflecting how relatively easy inter-task prediction is for this benchmark. For *sc*, PATH has a 35% higher miss rate than PER at a history depth of 7.

It is difficult to compare PATH to PER qualitatively because they are based on fundamentally different con-

cepts. PATH attempts to find correlation linking where a program came from to where it's going. PER looks for cyclical behavior patterns at particular decision points. PATH performs better than PER suggesting that task flow is more strongly correlated to immediately preceding task flow than to cyclical behavior. This result is to some degree a function of the benchmarks we are using. Both PATH and GLOBAL are trying to capture the same information: the predecessor tasks leading up to the current task. PATH captures this information better because it can uniquely identify predecessors. In exit-based histories, two tasks who take their "first" exit to get to the current task are indistinguishable unless the pattern (to the depth of the history) of taken exits to reach each of them is unique.



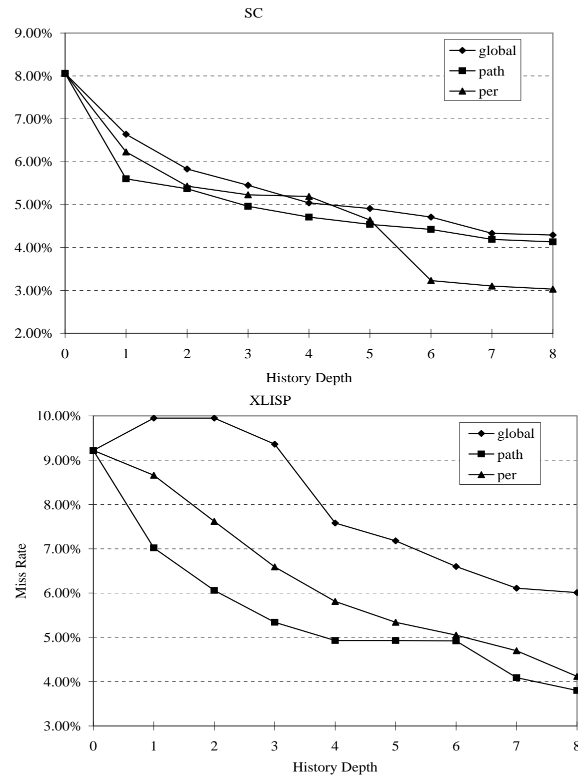


Figure 7 Performance for Ideal (alias-free) Prediction

Another argument in favor of using a path-based history scheme is that exit-based history schemes (GLOBAL and PER) tend to have poor utilization of the PHT address space. Most tasks have only one or two exits (see Figure 3) yet two bits are still used to encode each task step.

Ultimately, we found that for real implementations, PATH maintained its performance advantage over the other schemes, this is discussed in Section 6.3. In subsequent sections, we will focus on PATH.

5.3 Address Prediction

After predicting which of the four exits will be taken, the address of the predicted exit needs to be determined. For return exits, the address is determined using a return-address stack (RAS). For calls and branches the target address corresponding to the exit is given in the task header. For indirect calls and indirect branches, a target address must be predicted - the compiler can not currently provide the hardware with any help in this regard, nor can a simple prediction structure like a RAS be used for this type of target address prediction. Of the five benchmarks studied, two had a substantial number of indirect branches and indirect calls. For *gcc* and *xlisp,* 5% and 8% of the exits taken are indirect branches or indirect calls respectively. We concentrate our efforts on these two benchmarks.

A Task Target Buffer (TTB) similar to a Branch Tar-

get Buffer (BTB) is the obvious choice for making address predictions for indirect branches and calls. Each entry of the TTB is a target address and a 2-bit saturating counter that provides hysteresis, similar to the exit prediction automata discussed earlier. The TTB is indexed with some bits of the starting address of the task.

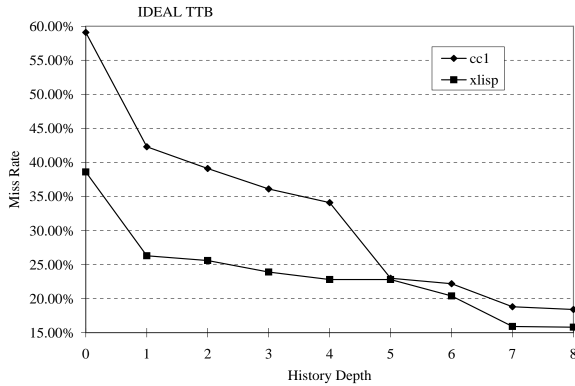


Figure 8 Performance of Ideal (alias-free) CTTB

We found that the TTB performed very poorly for address prediction of indirect branches and calls. The TTB has a miss prediction rate of 59% for *gcc* and 39% for *xlisp* with an infinitely large TTB. This high miss rate has a significant impact on the final task prediction rate. Therefore we investigated more sophisticated schemes for address prediction. Specifically, we examined the use of a correlating table indexing method, based on the path-based global history approach used for exit prediction. An index into the TTB is generated using the same function of path-history and current address that is used for path-based exit prediction. We call this structure a Correlated Task Target Buffer (CTTB). A CTTB using the path-based history has substantially lower miss rates than a TTB using a standard index derived only from the current task address. Figure 8 shows the miss rates for an ideal CTTB implementation for various depths of history. These results are for an infinite size CTTB with no aliasing.

5.4 Other uses for a Correlated Target Buffer

It is possible to perform task prediction without requiring control flow information in the header (from the compiler), which makes up the majority of the header. The CTTB uses the same indexing scheme as the exit predictor, and its prediction automaton is more general. Task prediction could be performed solely with a CTTB. This approach could potentially make task predictions more quickly.

The disadvantage of the CTTB-only method is that it offers worst performance and size characteristics. From the size point of view, each CTTB entry is 8 times as large as an exit prediction table entry. To realize similar prediction accuracies, a CTTB needs as many entries as an exit predictor. When a CTTB is used only for indirect address prediction it can be considerably smaller since fewer exits compete for the table storage, causing less destructive aliasing at small table sizes. From the performance point of view, the major impact is that return addresses are not predicted as well since a CTTB-only method can not use a RAS. Although the CTTB can predict many of the return addresses through correlation to recent history, miss rates in the range of 10% are not uncommon for returns. There are a number of other issues which all lead to slightly lower prediction accuracies. Indirect branches and indirect calls tend to have lower prediction accuracies in the CTTB-only scheme. Although the CTTB is much larger in this case, there is much more contention and aliasing, because all types of inter-task control flow instructions are competing for space in the buffer. The chances of constructive or neutral aliasing are reduced. There are some additional compulsory misses that could otherwise be avoided (recall that task headers includes the target address of unconditional and conditional control flow; these are easily predicted exits, but the target address is not known when they are first encountered in the CTTB-only scheme).

In Section 6.4 we present some results for real implementations of the CTTB-only prediction method.

6 Implementations of Path-Based Predictors

The path-based history generation scheme is used for exit prediction as well as for the CTTB. The path-based history scheme presented up until now was based on being able to capture information to identify uniquely the history of tasks leading to the current task. In this section we present an approximation of ideal path-based history generation that can be implemented with reasonably sized structures. The path-based history generation scheme is implemented with a table of automata indexed by a combination of a path-history register and some bits indicating the current task. When implementing a path-based prediction mechanism, we seek to encode the maximum information about the current task and the preceding path, using a minimal number of bits. The first design decision is how to represent the tasks in a path. We choose to identify a task using the least significant bits of its starting address, because these bits have the highest probability of being different for two different tasks. This is similar to how Nair identified basic blocks [11].

6.1 Key Design Features of Path Based Predictors

Ideally, a path would be identified by the full starting address of all tasks along the path. In implementing the predictor, the size of the PHT is limited; therefore the index is not large enough to hold the ideal amount of information. The performance of an implementation is dependent upon encoding as much information as possible about

the path onto the limited number of index bits. We use two design techniques to encode sufficient information about the path in the indexing bits. These are heuristics that we have found to work well; we attempt to give some insight into why they work.

The first design technique creates an intermediate index by concatenating bits of previous and current task addresses; this intermediate index may be longer than the PHT index. The final index is constructed from the intermediate index by folding the intermediate index into the number of bits needed to index the PHT. This folding is done by subdividing the intermediate index into subfields and exclusiveORing the subfields with each other. There is an important tradeoff involving the number of task address bits and the folding required to form the final index. On one hand, using more bits from the task address increases the information available for identifying individual paths. On the other hand, folding and exclusiveORing can lose information, and using more bits from the task address requires more folding to form the final index.

In general, the lower order address bits of a task convey more useful information because these bits are more likely to differ for preceding tasks. That is, some address bits tend to be more significant than others for differentiating task paths. After studying this tradeoff, we concluded that in most cases the index generated from a folded intermediate index is able to convey more information about the path than a shorter, unfolded index. And folding works better when the corresponding bits of different task addresses do not line up, thereby reducing the loss of the more significant information which tends to be in the same address bit positions.

The second design technique uses fewer bits from older tasks when constructing the intermediate index. This is intuitively reasonable because information about more recent tasks in the control flow tends to be more relevant to future control flow. Moreover, information about more recent tasks in the control flow implies some information about earlier tasks.

In the application of path-based prediction to exit prediction, there is an optimization for tasks with only one exit; a single exit is always predicted and no updates are made to the history table. This reduces aliasing by reducing the number of updates to the history table.

6.2 Parameters of the path-based predictor

We specify a path-based predictor using five parameters. Four of these parameters are used to specify the bits which constitute the intermediate index. The fifth parameter specifies the number of times the intermediate index is folded to generate the index.

The four parameters for specifying the makeup of the intermediate index are:

- **D**epth of the path history, the number of tasks preceding the current task that are used to represent the path
- the number of bits from each **O**lder task address (Current_Task - 2 to Current_Task - D)
- the number of bits from the **L**ast task address (Current Task - 1)
- the number of bits from the **C**urrent task address

The fifth parameter is the number of **F**olds. The index is generated by taking the intermediate index and breaking it into **F** equal sub-fields, which are then XORed together.

The length of the intermediate index, which must be a multiple of F, is:

$IntLength = (D - 1) * O + L + C.$

The number of entries in the correlating table is a function of the five parameters:

$SizeOfTable = 2^{IntLength} = 2^{((D - 1) * O + L + C) / F}$

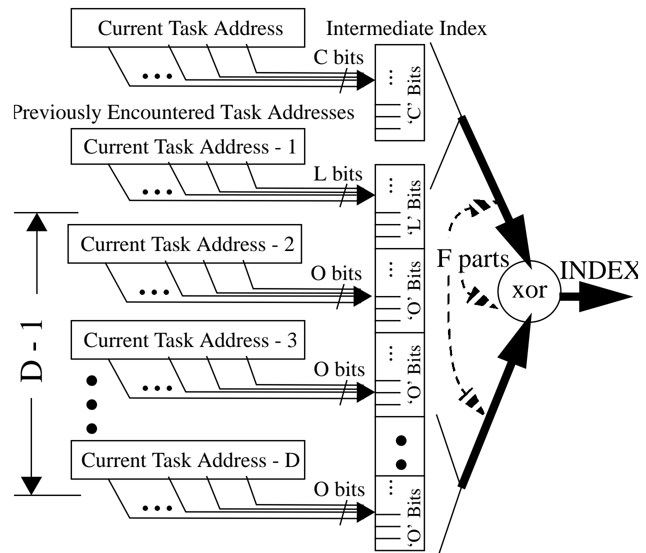


Figure 9 Intermediate Index and Index Generation

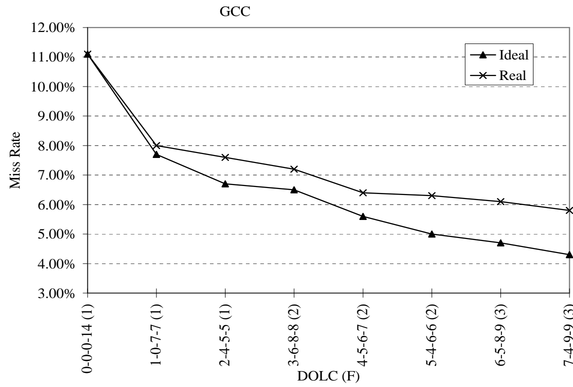
Developing an optimal intermediate index representation and folding function were not the goal of this research; there is much room for exploration. However, this organization gave us a framework to engineer reasonable sized, high-performance predictors.

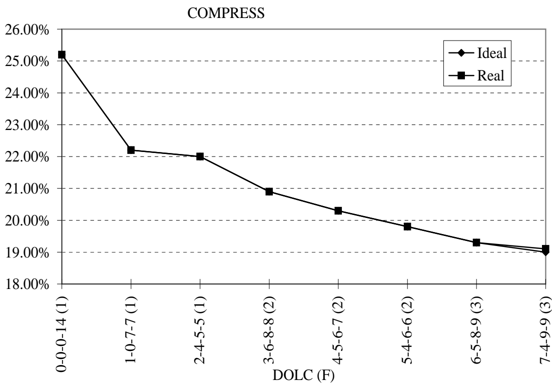
In later sections we will describe implementations in terms of the five parameters presented here. We use the following convention: D-O-L-C (F). For example a 6-5-8-9 (3) implementation is 6 **D**eep, using 5 bits from **O**ld tasks, 8 from the **L**ast task, 9 bits from the **C**urrent task, and is broken into 3 parts to **F**old together. For this example the intermediate index is 42 bits, the actual index is 14 bits and the table has 16K entries.

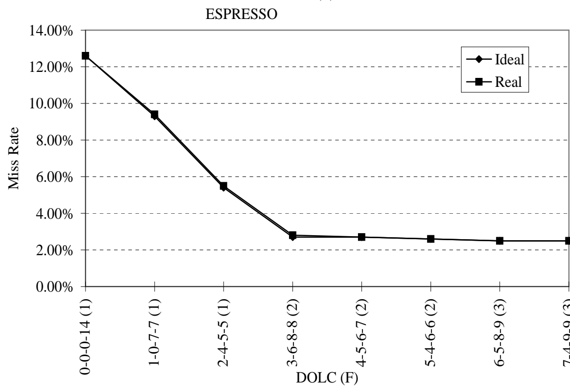
6.3 Exit Prediction

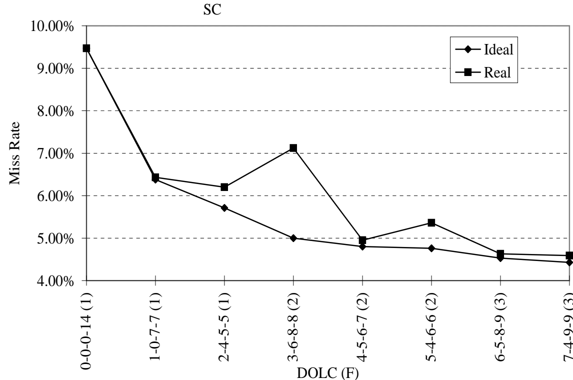
Real implementations of the path-based predictor perform close to the ideal (alias-free) implementation. Figure 10 compares the relative performance of a real implementa-

tion to the ideal for a range of depths and a constant table size of 8 kB (14 bits of index * 4 bits per entry). These configurations were determined through experimentation to be good configurations, but are not known to be optimal.









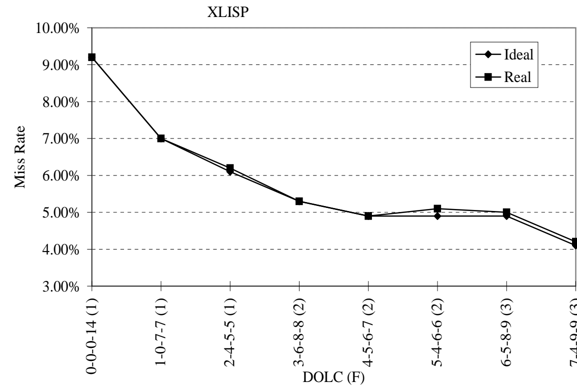


Figure 10 Comparison of Ideal Predictors to Real Implementations

gcc shows the largest deviation from the ideal because the index and table are not large enough to capture all possible states. Figure 11 shows the number of different states the predictor sees in the ideal and real implementations for *gcc* and *espresso*. *Espresso* is representative of the other benchmarks. A larger implementation would help increase the prediction accuracy for *gcc* considerably.

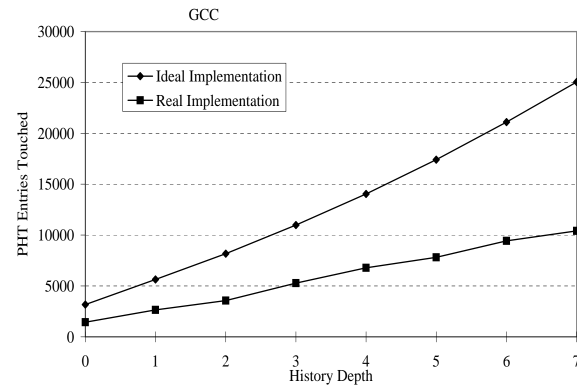


Figure 11 States touched in the PHT

In *sc*, we see two distinct drops in prediction accuracies at depths of 3 and 5. These points correspond to the depths where new degrees of folding (1 to 2 and 2 to 3 degrees of folding respectively) were introduced in the implementations. The advantages of a longer history (in

the intermediate index) quickly overpower the penalty introduced by occasionally folding “useful” bits together; the accuracy of the predictor reapproaches the ideal after these points.

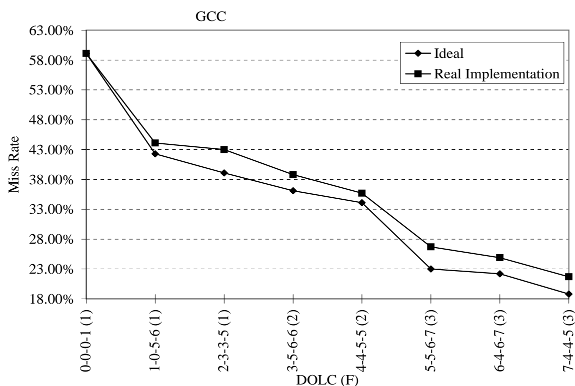
We do not present results for implementations of GLOBAL or PER due to space limitations. This can be justified easily because the implementations of the path-based history predictors tend to do better than the ideal implementations of the other two schemes. Our depth 7 implementation of PATH has a lower miss rate than the ideal depth 7 PER predictor for all the benchmarks except for *sc*. Our depth 7 implementation of PATH has a lower miss rate than the ideal depth 7 implementation of GLOBAL for all the benchmarks except *gcc*, where it is within 5%.

A path-based prediction scheme perform better than other prediction schemes for inter-task prediction. However, path-based predictors have not yet proven themselves to be competitive with exit-based schemes for traditional branch prediction, partly due to their recent introduction. Furthermore, the path-based method does not encode path information as efficiently as exit-based histories. In branch prediction, the exit-based history can be kept with only a single bit per entry, which is not true in inter-task prediction or other multi-way branching problems where a few bits are required per entry. Another reason for the advantage seen by path-based schemes in inter-task prediction is that the higher concentration of calls, returns and indirect branches makes it harder for exit-based approaches to capture the path history.

6.4 CTTB Address Prediction

6.4.1 Prediction for Indirect Branches & Indirect Calls

In this section we present results for predicting the target addresses of indirect branches and indirect calls with reasonably sized implementations of a CTTB. Figure 12 compares reasonable sized implementations of a CTTB to ideal (infinite size) implementations for a range of depths. All the implementations have a table size of 8 kB (11 bits of index * 4 bytes per entry).



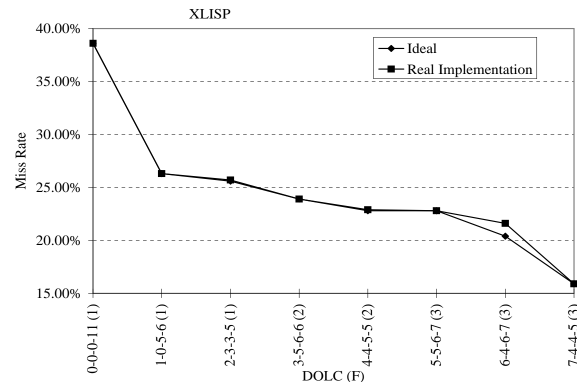


Figure 12 Comparison of Real to Ideal Predictors for Address Prediction

For *xlisp* these implementations perform nearly identical to the ideal. For *gcc* the implementations diverge from the ideal by up to 15%. This divergence is caused by the table not being large enough to capture the number of states (unique paths) observed for the benchmark. In these implementations the number of entries for the CTTB is one-eighth of the entries we implemented for the exit predictor. Scaling the CTTB to the number of entries of the predictor in order to do CTTB-only prediction would be quite costly.

6.4.2 CTTB-only Task Prediction

In this section we present results for CTTB-only inter-task prediction, which predicts without using information from the header. Table 3 compares CTTB-only prediction accuracy to an exit predictor which determines target addresses using the header, a RAS and a small CTTB. The results here are for predicting the actual address of the next task. We compare a CTTB-only method (14-bits of index) to an exit predictor (14-bits of index) with a RAS and a small CTTB (11-bits of index). The predictors here all have a history depth of 7. We see that the CTTB-only method, requiring 64kB of state, performs from 4% to 54% worse than the alternative method. The largest deviation is for *gcc* and *xlisp*, 54% and 41%, respectively.

Prediction Method	gcc	comp	espr	sc	xlisp
CTTB-only predictor (64kB storage)	10.5	19.8	2.6	5.3	7.9
Exit predictor with RAS & CTTB (16kB storage)	6.8	19.1	2.5	4.6	5.6

Table 3 Miss Rates for CTTB-only vs. Exit Predictor with RAS & CTTB

7 Conclusion

In this paper we studied the issues involved in inter-task prediction and presented a task predictor that has very high prediction accuracies for reasonable sized structures. The key design points presented were:

- The last exit with hysteresis prediction automaton offers

the best accuracy/size performance tradeoff.
- A path-based history scheme works best for task prediction.
- A correlated target buffer is essential for good address prediction of indirect jumps and indirect calls.
- Although task prediction without a header is possible it does not offer comparable accuracy/size performance.

Another important area covered in this paper was efficient ways to implement path-based history schemes. There were two important heuristics presented:
- Creating a larger intermediate index value and folding it to form the actual index
- Decreasing the number of bits older tasks contribute to the history relative to recently encountered tasks.

In this paper we used prediction accuracy as our metric. In general higher prediction accuracy leads to better execution performance. In some cases data dependencies limit the gains of prediction accuracies. Also, not every prediction is equally important to performance.

We conclude by presenting some performance numbers to demonstrate that in general better prediction does increase performance. Table 4 presents instructions per cycle (IPC) performance numbers generated with a timing simulator. We present IPC numbers for a simple predictor (using the task address to index the PHT), GLOBAL, PER, PATH and an upper bound of perfect inter-task prediction. For all the implementations the PHT is 16KB and a history depth of 7 is used. All implementations use a CTTB for indirects and a RAS for returns. The processor core has four 2-way OOO processing units. For *gcc* and *xlisp* where PATH had the most substantial prediction accuracy advantages, the better task prediction increases the IPC by 5% to 12% over the next best prediction scheme. PATH performs at least as well as other predictors for all the benchmarks.

	gcc	comp	espr	sc	xlisp
Simple	1.55	1.44	2.61	2.13	1.59
GLOBAL	1.59	1.47	2.67	2.21	1.77
PER	1.48	1.44	2.68	2.22	1.76
PATH	1.68	1.47	2.70	2.22	1.89
Perfect	1.83	1.85	2.75	2.26	2.03

Table 4 IPC from detailed timing simulator

Acknowledgments

This work was supported in part by NSF Grant MIP-9505853 and by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

The authors would like to thank Chris Lukas and Gurindar Sohi for their input; Scott Breach for developing the multiscalar simulators and determining the IPC performance numbers; Andreas Moshovos for modifying the functional simulator to study task prediction; T. Vijaykumar for developing the multiscalar compiler.

References

[1] S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. The anatomy of the register file in a multiscalar processor. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, December 1994.

[2] S. E. Breach. Personal Communication, November 1995.

[3] T. Conte, et al., Optimization of instruction fetch mechanisms for high issue rates. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[4] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[5] M. Franklin and G.S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers* volume 45, no 5, May 1996.

[6] M. Franklin. *The Multiscalar Architecture*. Ph.D. thesis, Computer Sciences Department, University of Wisconsin-Madison, November 1993.

[7] L. Gwennap. Intel's p6 uses decoupled superscalar design. *Microprocessor Report*, pages 9–15, February 16 1995.

[8] D. R. Kaeli and P. G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 1991.

[9] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17(1):6–22, January 1984.

[10] S. McFarling, Combining Branch Predictors, DEC WRL TN-36, June 1993.

[11] R. Nair. Dynamic path-based branch correlation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995.

[12] S.-T. Pan, et al., Improving the accuracy of dynamic branch prediction using branch correlation, In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 1992.

[13] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, May 1981.

[14] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[15] T.-Y. Yeh and Y. Patt. Alternative implementations of two-level adaptive training branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[16] T.-Y. Yeh. *Two Level Adaptive Branch Prediction and Instruction Fetch Mechanism for High Performance Superscalar Processors*. Ph.D. thesis, Dept. of Electrical Engineering & Computer Science, University of Michigan, 1993.

[17] T.-Y. Yeh and Y. Patt. Two level Adaptive Branch Prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, November 1991.