Control Flow Prediction For Dynamic ILP Processors

Dionisios N. Pnevmatikatos — Computer Sciences Department, University of Wisconsin-Madison, Madison, WI 53706

Manoj Franklin — ECE Department, Clemson University, Clemson, SC 29634

Gurindar S. Sohi — Computer Sciences Department, University of Wisconsin-Madison, Madison, WI 53706

Abstract

We introduce a technique to enhance the ability of dynamic ILP processors to exploit (speculatively executed) parallelism. Existing branch prediction mechanisms used to establish a dynamic window from which ILP can be extracted are limited in their abilities to: (i) create a large, accurate dynamic window, (ii) initiate a large number of instructions into this window in every cycle, and (iii) traverse multiple branches of the control flow graph per prediction. We introduce *control flow prediction* which uses information in the control flow graph of a program to overcome these limitations. We discuss how information present in the control flow graph can be represented using *multiblocks*, and conveyed to the hardware using *Control Flow Tables* and *Control Flow Prediction Buffers*. We evaluate the potential of control flow prediction on an abstract machine and on a dynamic ILP processing model. Our results indicate that control flow prediction is a powerful and effective assist to the hardware in making more informed run time decisions about program control flow.

1. Introduction

Conditional branch instructions are a necessary evil in programs. They introduce *control dependencies*, the flow of control being described by the *control flow graph (CFG)* of the program [1]. Ordinarily, these control dependencies must be respected: no instruction that occurs after a branch can be executed before the branch. The plethora of conditional branches in many programs, however, forces the development of alternatives, because very little instruction-level parallelism (ILP) would be exposed if this simple-minded rule were obeyed.

Most of these alternatives employ some form of *speculative* execution. The branch outcomes are predicted using a prediction strategy, and instructions from the predicted path are considered for execution (one alternative is to consider instructions from both paths). By doing so, we establish a path through the CFG, which is an approximation of the run-time execution path. Once this path (or *trace*, or *window*), is established, hardware or software mechanisms can be used to detect independent instructions and establish a schedule to execute them in parallel. This general philosophy for alleviating the impact of control dependencies is germane to any static or dynamic technique.

Knowledge of the CFG is essential to any technique that tries to expose speculative ILP. A recent study by Lam and Wilson has underscored the importance of the knowledge of control-dependence relationships amongst instructions for ILP processors [13]. Their study shows that an (abstract) ILP processor which performs branch prediction and speculative execution but allows only a single flow of control can extract a parallelism of only about 7. If the ILP processor exploits the control dependence information to execute instructions before branches which they do not depend upon, the parallelism limit is increased to about 13. Finally, if the processor can exploit the control dependence information, resolve multiple branches in one cycle, and follow multiple flows of control, the available parallelism increases to about 40.

Whereas static techniques to alleviate the impact of control dependencies, such as *Trace scheduling* [7, 8], *Predicated Execution* [11], *Superblock* and *Hyperblock scheduling* [6, 15], and *Boosting* [18], exploit information present in the CFG, dynamic techniques typically do not. Instead, they rely only on dynamic branch prediction techniques to construct the dynamic window. Dynamic branch prediction schemes make decisions about a branch when the branch is encountered, for example, when it is fetched and decoded. The decision is made using a Branch History Table (BHT) [14, 17], which typically stores a condensed history of past taken/not taken outcomes; this history is updated when the branch outcome is known.

We believe that dynamic branch prediction, as we know it today, is fundamentally limited in establishing a large and accurate dynamic window, since it makes localized decisions without any knowledge of the global control structure of the program. This lack of knowledge results in several problems. First, to predict a branch, its identity must be known, i.e., the branch must be encountered by the hardware. With normal branch prediction, a prediction is made when the branch instruction is fetched by the fetch unit. While some recent prediction mechanisms [16, 20] do not require the branch address to predict its outcome, the identity of the branch must still be known so that the predicted target address can be determined (either using a BTB structure [14], or by decoding the branch instruction). After the prediction decision, instructions from the predicted path need to be fetched, and the next branch in the predicted path encountered (and its identity determined), before its target can be predicted. Since the number of instructions between two consecutive branches can be arbitrary, it is not always possible to determine the identity of the next branch, and make a prediction, in the very next cycle after a branch prediction is made. If a branch prediction can not be made in every cycle, the prediction bandwidth, and consequently the overall number of instructions per cycle (IPC) that can be executed will suffer. If we like to sustain a rate of one prediction per cycle, the identity of the next branch to be predicted must be known when a prediction is made. This problem has been identified and partly addressed in [21], where the prediction mechanism can perform one prediction per cycle (two or more predictions per cycle in [19]) as long as the next branch lies inside the block of instructions fetched from the instruction cache.

The second problem has to do with the *initiation size*, that is, the number of instructions that can enter into the dynamic window in a given cycle. Clearly the best-case IPC is limited by the number of instructions that can enter into the dynamic window in a cycle. If it is possible to ‘‘get past’’ or traverse only one branch at a time, a single node of the CFG can be initiated per cycle, and the average initiation size is limited by the average straight-line code size — a rather small value for many programs. What we need is a mechanism that makes it possible to traverse multiple branches at a time, initiate a set of CFG nodes at a time into the dynamic window, and quickly identify the (more distant) CFG node to execute next.

The third problem has to do with the accuracy/size of the dynamic window. Some of the branches with low prediction accuracies are part of if-then-else control structures. As we shall see in an example later, such branches can result in small or inaccurate dynamic windows, containing very few instructions that would actually be executed.

This paper introduces the concept of *control flow prediction (CFP)*. The idea is to go a step beyond common branch prediction and allow the hardware to have information about parts of the CFG of the program so that it can make better decisions about navigating through the CFG. The better the navigation, the greater the ILP that can be exposed and exploited. Overall, control flow prediction increases the bandwidth of the prediction mechanism, increases the initiation size, permits the overlapped execution of multiple independent flows of control, and allows multiple branch instructions to be resolved simultaneously. These are intermediate steps we need to take in order to increase the size of the dynamic window and achieve our ultimate goal, which is to expose and exploit the maximum ILP.

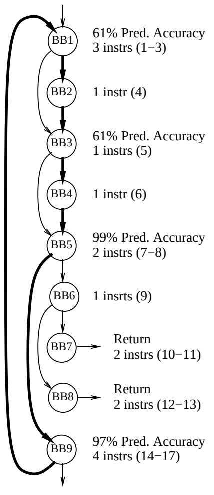
The remainder of this paper is organized as follows. In section 2 we introduce multiblocks and show how they are used for control flow prediction. Section 3 discusses Control Flow Prediction Tables and Buffers — hardware structures for control flow prediction, and in section 4 we carry out a quantitative analysis of control flow prediction on an abstract machine. In section 5 we evaluate control flow prediction on a concrete machine model, the *multiscalar* model. In section 6 we discuss the relationship of control flow prediction to other concepts for alleviating the impact of control dependencies in a dynamic ILP processor, and we conclude in section 7.

2. Control Flow Prediction—Exploiting Control Flow Graph Characteristics

Let us see how we can exploit the information present in the CFG of a program. By inspecting a program's CFG, it is possible to infer that some basic blocks will be executed regardless of the outcome of the previous branches. To illustrate this, let us take a simple example. Figure 1(a) presents the C code for the inner loop of the *Cmppt* routine of the *eqntott* benchmark from the SPEC92 suite. Figure 1(b) presents its CFG, showing the number of instructions in each basic block (BB) for a MIPS R2000 executable and sample prediction accuracies (for a counter-based predictor) of the branches that terminate the basic blocks.

```
for (i = 0; i < ninputs; i++) {
    aa = a[0]->ptand[i];
    bb = b[0]->ptand[i];
    if (aa == 2)
        aa = 0;
    if (bb == 2)
        bb = 0;
    if (aa != bb) {
        if (aa < bb) {
            return (-1);
        } else {
            return (1);
        }
    }
} /* rof */
```

(a)



(b)

Figure 1: C Code and CFG for the inner loop of function Cmppt in Eqntott

With common branch prediction, when BB1 is entered, the prediction mechanism has to wait until the branch at the end of BB1 is encountered. Depending on the prediction, either BB2 is executed or not, and then BB3 is executed. If the branch was predicted to go to BB3, and the prediction was wrong, we would have to squash the execution of BB3, execute BB2, only to come back and execute BB3 again. Continuing further, the probability of reaching BB5 without any incorrect predictions is 37.2%, even though BB5 is control independent of BB1 and is guaranteed to execute regardless of the outcomes of the branches at the end of BB1 and BB3!

What could we do if the CFG of the above example were known? Once BB1 is entered, we could initiate the execution of BB3, or even BB5, without worrying about the branches at BB1 and BB3. (The underlying hardware must be able to simultaneously execute code from different parts of the dynamic trace/window and enforce dependencies between the different parts of the computation.) If we do so, then we accomplish several things. First, we will increase the bandwidth of the prediction mechanism that navigates through the CFG. Second, we will effectively traverse more than one branch per cycle, consequently increasing the initiation size, and increasing the peak IPC. Third, we will have created a larger, and more accurate, dynamic window. (The probability of correctly initiating the execution of BB5 is 100% as opposed to 37.2% with branch prediction.) A larger dynamic window allows more ILP to be exposed and exploited [3, 4]. Next, we describe multiblocks, a vehicle for conveying the necessary control flow information to the hardware prediction mechanism.

2.1. Multiblocks and their Use

A multiblock is a subgraph of the CFG of a program; the first instruction of a multiblock is defined to be its *entry* point. Starting from the CFG of a program, whose nodes are basic blocks, the CFG can be transformed into a graph whose nodes are multiblocks. Information about multiblocks can be conveyed to the hardware (as we show in section 3), and used to make informed decisions about navigating through the CFG. Once a multiblock is entered, its exit points can be determined in a straightforward and accurate manner (so that the next multiblock to be executed can be accurately identified), even though the exact path through it may be unknown. As we will see later, the execution of multiblocks can be overlapped, allowing the overlapped execution of multiple flows of control.

Given the flexibility to consider an arbitrary subgraph of the CFG (including loops) as a multiblock, what kind of subgraphs should we be interested in when constructing multiblocks? The answer is tied to the data dependencies between the instructions in the multiblock and the underlying execution model. There are many reasons why one might want to restrict the scope of a multiblock. For instance, if the execution model can exploit inter-multiblock parallelism, then it is better to pack dependent instructions into a multiblock. Thus, each iteration of a data-independent loop can be considered as a multiblock, permitting the initiation of one iteration per cycle. In the case of a DOACROSS loop where iterations are dependent, it could be advantageous to encapsulate the entire loop in a multiblock. Figure 2 illustrates this point. This code, taken from the SPEC92 benchmark xlisp, consists of a doubly nested loop; the inner loop traverses a linked list, and its iterations are both control and data dependent. However, each activation of the inner loop is independent of the previous one. If the entire inner loop is defined to be a single multiblock, it is possible to start many activations of the inner loop, without waiting for the previous ones to complete.

```
/* check the environment list */
for (fp = xlenv; fp; fp = cdr(fp))
  for (ep = car(fp); ep; ep = cdr(ep))
    if (sym == car(car(ep)))
      return (cdr(car(ep)));
```

Figure 2: C code for function Xlygetvalue in Xlisp.

Another consideration in the choice of multiblocks is the number of targets a multiblock can have. Allowing many targets increases the flexibility in constructing multiblocks, and produces larger multiblocks. However, as the number of targets increases, the dynamic prediction mechanism needs more state information, and the prediction accuracies are likely to decrease. Therefore, a compromise may be to allow multiblocks to have at most two targets. An exception to this is when a multiblock has more than two targets, but all except one or two are rarely exercised at run time, in which case the prediction mechanism can ignore these and predict one of the two likely ones.

A final consideration in deciding the scope of multiblocks is whether all the instructions in it are statically adjacent or not. This has implications from the point of ease of conveying the multiblock information to the hardware. If all instructions in a multiblock are statically adjacent, then it can be specified by giving its entry point and length; otherwise, all the multiblock parts must be specified, requiring significantly more space.

For the rest of this paper we assume a simple form of multiblocks — a loop-free collection of statically adjacent basic blocks with a maximum of two targets — and all references to multiblocks shall allude to these simple (restricted) multiblocks. To illustrate the multiblock concept under these constraints, consider our Cmppt example (c.f. Figure 1). Starting at node BB1, we can form the following multiblocks: BB1, BB1-BB2, BB1-BB3, BB1-BB4, BB1-BB5 and BB1-BB8, out of which BB1-BB8 is the maximal multiblock. (Notice that the exits from BB 7 and 8 are function returns, and are therefore to the same target.) The basic block sequences BB1-BB6, BB1-BB7 and BB1-BB9 are not counted as multiblocks because they have three targets.

The reduced CFG, containing maximal multiblocks as nodes, for the same code is shown in Figure 3; the first multiblock (containing BB1 through BB8) is called MB1-8 and the second one (containing just BB9) is called MB9. With this reduced CFG, only two predictions are required per iteration of the loop as opposed to four predictions that an ordinary branch prediction scheme would require. Second, an average of 5.73 instructions can be initiated per cycle into the dynamic window (an average of 7.46 for different executions of MB1-8 and 4 for MB9) versus 2.87

instructions per cycle that would have been possible with ordinary branch prediction. Third, the need to predict the branches at BB1 and BB3 has been eliminated. This eventually results in a larger dynamic window. Last, we can initiate (and execute, if data dependencies allow) one iteration of this loop every two cycles, if adequate hardware support is available.

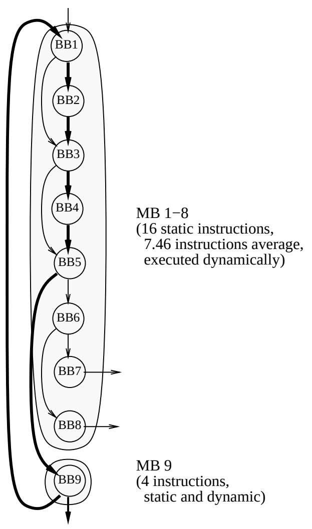


Figure 3: Reduced CFG for the Cmppt inner loop.

3. Structures for Control Flow Prediction

3.1. Control Flow Table (CFT)

Control flow and multiblock information can be represented and conveyed to the hardware in a number of ways. The simplest and most intuitive way is in the form of a table containing one entry for each multiblock. We call this table the Control Flow Table (CFT). Each CFT entry contains all the necessary fields to fully describe a multiblock. Because all instructions in a multiblock are contiguous in memory, the multiblock's entry point and length are sufficient to fully specify its boundaries. We also store the multiblock's list of targets, and perhaps other useful static information that we might want to convey to the processor. Table 1 shows the CFT for our Cmppt example.

How is the CFT accessed? Since there is only one CFT entry per multiblock, the obvious way is to construct the CFT by assigning a unique number to each multiblock in the program, and use it to index the CFT. However, the CPU uses instruction addresses to sequence through a program. Therefore, a mapping from the multiblock numbers to the instruction addresses is required, so that the prediction mechanism can convey the address of the predicted instruction to the CPU. Furthermore, branches with unknown targets (indirect jumps and returns) require in addition a reverse mapping. The alternative way to access the CFT is to index it using instruction addresses. This eliminates the need to map multiblock numbers to instruction addresses, but results in a sparse CFT.

Address	Target1	Target2	Length
MB 1-8	MB 9	Return	16
MB 9	MB 10	MB 1-8	4

Table 1: CFT entries for the Cmppt inner loop

If the CFT is stored in memory then size is an important factor, and accessing the CFT entries by multiblock numbers is the preferable option. The mapping from multiblock numbers to addresses can be implemented by having an address field in the CFT entry; however, the reverse mapping required for returns and indirect jumps becomes difficult. To speed up the CFT accesses we advocate caching the CFT entries as discussed below; this also solves the fragmentation problems.

3.2. Control Flow Prediction Buffers (CFPB)

The Control Flow Prediction Buffer (CFPB) is a cache of CFT entries, with each entry appended with sufficient information to make dynamic prediction decisions. If a counter based branch prediction mechanism is used, the CFPB resembles a BTB [14] augmented with multiblock information. The CFPB is accessed once for every multiblock activation to determine the multiblock's size and targets. Between the two targets, the prediction mechanism selects one target. If the selected target is the special value *Return*, then the actual target is predicted using a return stack [12]. If the selected target is *Unknown*, indicating an indirect branch, the predictor can optimistically use the last outcome of this branch. Table 2 shows the CFPB for our Cmppt example, assuming per-multiblock prediction information.

Address	Prediction State	Target1	Target2	Len
MB1-8	Taken	MB 9	Return	16
MB9	Taken	MB 10	MB 1-8	4

Table 2: CFPB entries for the Cmppt inner loop

Multiblock information is associated with the entry point of a multiblock, therefore the next access to the CFPB for the predicted multiblock can start in the next cycle using the predicted target. This access will also provide the prediction state information of the predicted multiblock, resulting in a prediction throughput of one prediction per cycle when accesses to the CFPB are hits.

Although we have presented the CFPB as a modified BTB using history based branch prediction and per-multiblock state information, the CFPB poses no restriction on the prediction mechanism that can be used. If the structure of the table(s) used by the prediction mechanism is different from the structure of the CFPB (as in the cases of [20] and [16]), then the prediction state field in the CFPB can be eliminated, and the CFPB is just a cache of the CFT entries.

Because the CFPB is a cache, fragmentation is not an issue and we can use instruction addresses to access the CFPB. This eliminates the need for mappings between the instruction and multiblock addresses. If the hit ratios in the CFPB are sufficiently high, then the time to retrieve a CFT entry on a CFPB miss becomes less of a concern, and slower but more compact CFT representations become possible.

3.3. On-the-fly Reconstruction of CFT Entries

Most of the information needed to reconstruct the CFG and multiblocks of the program (and therefore the CFT) is available in the executable. Rather that compute the entire CFT *a priori* and store it in memory, the relevant CFT entries can be constructed at run-time, on a CFPB miss. However, given realistic constraints on time, storage, and hardware complexity, it is unlikely that a hardware mechanism will be able to identify optimal multiblocks. One way to ensure that the optimal multiblocks will be used, is to have the compiler guide the hardware mechanism by supplying the optimal length for each multiblock. Once the length is known, the bounds of the multiblock uniquely defined and the hardware mechanism can simply traverse the instructions of the multiblock and collect the set of all branch targets that fall out of its bounds and of any backward branch. The algorithm for this process is shown in Figure 4.

```
/* Starting and ending addresses are known */
foreach instruction in the multiblock {
    if the current instruction is a branch {
        if (the branch target is greater than the
            ending address OR the branch is backward)
        add the branch target to the target set
    }
}
if the last instruction is not an unconditional branch
    add the fall through address to the target set
```

Figure 4: Algorithm for On-The-Fly reconstruction of CFT entries

The compiler can encode the multiblock length in the instruction stream using a special NOP instruction (for example an ‘‘`addi R0, R0, length`’’ in the MIPS instruction set) without requiring any instruction set modifications or special mapping tables. The use of such special instructions increases the code size of the program, but since a multiblock will contain several basic blocks we expect this increase to be small. Furthermore, these special instructions will not increase the execution time of the program as long as they are recognized by the instruction fetch mechanism and are removed from the instruction stream before they enter the execution pipeline.

4. Evaluation on an Abstract Machine

We now consider the effectiveness of control flow prediction in achieving the stated objectives. As mentioned earlier, getting past control dependencies and establishing a trace is one problem, extracting ILP from this trace and establishing a schedule for the execution of instructions is another. Though related (the type of window that we need to establish depends upon the type of window that the underlying machine can extract ILP from), the two problems are best dealt with separately. Therefore, we first evaluate the potential of the control flow prediction concept on an abstract machine. Our abstract machine maintains a dynamic window from which its extracts ILP. Instructions are initiated into the window (by the control flow prediction mechanism); the machine executes the instructions in the window when it can, enforcing necessary dependencies in the process. The instructions which the machine chooses for execution at any given time can be from different parts of the window, perhaps from different flows of control in the program. With these computation abilities, the overall parallelism-extraction abilities of the machine are limited by how large a dynamic window size can be established by the CFP mechanism, and by how many (useful) instructions could be initiated into this window.

We use 15 programs as benchmarks in our evaluation. *Compress*, *Eqntott*, *Espresso*, *Gcc*, *Sc*, and *Xlisp* are C programs and *Doduc* and *Spice* are FORTRAN programs from the SPEC92 benchmark suite, We also use *Yacc*, *Tex* and three simulators written in C: *Tycho*, a cache simulator [10], *SuperMips*, a simulator of a superscalar CPU using the MIPS instruction set, and *ThisSim*, the simulator used for the studies of this section. Finally, we use two Object Oriented Database benchmarks written in C++: *Sun-benchmark* [5], and *Tektronix* [2].

Table 3 show the basic statistics for the programs, including the dynamic number of instructions we consider for execution, the percentage of these instructions that are branches (conditional and unconditional), the code size (in instructions) and the corresponding Control Flow Table size (in entries).

For the remaining results presented in this paper, when we present results without control flow prediction, we mean results obtained using branch prediction and when we present results with control flow prediction, we mean results obtained with a control flow analysis of the program to detect and form multiblocks, and using a CFPB to make prediction decisions. We allowed only two targets per multiblock, and we placed no restrictions on the size of each multiblock. In many cases, we were able to coalesce 20 or more basic blocks into a multiblock, although we observed only marginal improvement in all our metrics when more than 5 or 6 basic blocks were coalesced. (The ability to create a bulk of the multiblock in the CFG by coalescing only a few basic blocks bodes well for on-the-fly creation of CFPB entries.) Third, we assumed all hits in the CFPB (and the BTB without control flow prediction). (While this assumption will not alter our results in terms of the units used to present the results, it would alter the results per units of time; the magnitude would depend upon the CFPB and BTB miss penalty.) Fourth, both the BTB and the CFPB use a GAs(8,64) two-level branch predictor [20].

Program	Dynamic Instructions (Millions)	Branch Ratio		Static	
		Cond.	Uncond.	Code size (Instrs)	CFT size (Entries)
Compress	22.68	0.149	0.040	6144	885
Doduc	500.00	0.064	0.025	46080	3624
Eqntott	1000.00	0.319	0.010	10240	1715
Espresso	1000.00	0.163	0.012	47104	6644
Gcc	1000.00	0.156	0.042	172032	25653
Sc	500.00	0.202	0.034	32768	6013
Spice	1000.00	0.115	0.061	95232	8729
Sunbench	640.03	0.137	0.065	16384	1799
SuperMips	500.00	0.111	0.056	14336	1851
Tektronix	937.57	0.131	0.083	17408	1984
Tex	214.67	0.143	0.055	60416	9976
ThisSim	650.60	0.102	0.046	7168	957
Tycho	408.47	0.123	0.060	11264	1546
Xlisp	500.00	0.157	0.091	21504	3637
Yacc	26.37	0.237	0.020	12288	1737

Table 3: Benchmark program characteristics

In our results, the *initiation size* is the number of useful instructions that can enter into the dynamic window in a cycle. When control flow prediction is not used, the initiation size is just the number of instructions between branches. When control flow prediction is used, the initiation size is the number of *useful* instructions in the multiblock that contribute to the computation.

Figure 5 presents the cumulative distribution of the initiation size without control flow prediction, and Figure 6 presents the same distribution with control flow prediction. Figure 7 presents the cumulative distribution of the number of useful instructions between mispredicted branches (or the dynamic window size) without control flow prediction, and Figure 8 presents the same for control flow prediction. Table 4 summarizes the average results. In the table, we have listed the mean initiation size and the mean window sizes, both with and without control flow prediction, and the branch prediction accuracies of the branches that need to be predicted in both cases. We have also listed the average number of branches traversed per cycle when control flow prediction is used. This is assuming that a prediction is made in each cycle. (Without CFP, the number of branches traversed in a prediction is 1).

The first observation from Table 4 is the increase in the number of branches traversed per cycle with control flow prediction. For example, in the case of *Doduc*, with control flow prediction we traverse 2.22 branches per cycle. For many programs there is a significant increase in the number of branches traversed per cycle, indicating the effectiveness of multiblock formation in merging several small basic blocks into a larger multiblock.

Program	Mean Initiation Size	Mean Window Size	Branch Prediction Accuracy
Compress	5.24	64	89.59
Doduc	11.17	436	96.42
Eqntott	3.03	49	93.60
Espresso	5.77	120	94.75
Gcc	5.02	72	91.11
Sc	4.22	126	96.04
Spice	5.64	599	98.55
Sunbench	4.93	504	98.55
SuperMips	5.95	319	97.20
Tektronix	4.63	241	96.79
Tex	5.02	169	95.87
ThisSim	6.68	331	97.06
Tycho	5.42	168	95.16
Xlisp	4.00	144	95.63
Yacc	3.87	103	95.84
Harmonic Mean	4.95	135	95.41

Table 4(a): Results without Control Flow Prediction

Program	Mean Initiation Size	Mean Window Size	Branch Prediction Accuracy	Traversed Branches Per Cycle
Compress	8.40	86	89.71	1.33
Doduc	21.64	649	94.67	2.22
Eqntott	5.53	371	98.49	1.82
Espresso	8.57	213	95.98	1.37
Gcc	9.44	105	91.02	1.47
Sc	6.35	205	96.36	1.51
Spice	8.08	661	98.55	1.11
Sunbench	8.40	566	98.14	1.45
SuperMips	13.25	844	97.71	2.17
Tektronix	8.07	415	97.15	1.54
Tex	6.24	207	96.10	1.16
ThisSim	17.46	779	96.07	3.15
Tycho	7.72	241	95.85	1.23
Xlisp	5.11	157	95.34	1.16
Yacc	4.96	150	96.51	1.22
Harmonic Mean	7.84	230	95.77	1.47

Table 4(b): Results with Control Flow Prediction

Control flow prediction reduces the number of branches that require prediction by traversing multiple branches in a single prediction. The effect on the branch prediction accuracy is not uniform across the benchmark programs. For *Eqntott* and *Espresso*, many of the poorly behaved branches were encompassed into multiblocks and the prediction accuracy of the remaining branches increased. In some of the other cases, the prediction accuracy decreased, indicating that some of the accurately predicted branches were encompassed into multiblocks. The reduction in branch prediction accuracy is not a negative result since our main goal is to increase the initiation size and the window size, and the branch prediction accuracy is only one factor that determines the window size.

We now consider the mean initiation and window sizes. From Figures 5 and 6 and Table 4, we see that control flow prediction results in a considerable increase in the initiation size in most cases. For *SuperMips* and *ThisSim*, the improvement in the mean initiation size is greater than a factor of 2. The improvements for the other benchmarks are also respectable and the harmonic mean of the mean initiation size has increased from 4.95 to 7.84. From Figures 7 and 8 and Table 4, we see that the mean window sizes have also increased considerably for all benchmarks, except for *Spice* and *Xlisp*. The harmonic mean of the mean window size has increased from 135 to 230.

Why did the simulators *SuperMips* and *ThisSim*, programs that we had control in writing, perform so well, and why didn't *Xlisp* and *Spice* do better as far as the window size was concerned? The reason is not an intentional doctoring of the code in *SuperMips* and *ThisSim*. Because these programs are very long running, we had given extra attention to reducing their running times when they were written (and long before they were used as benchmarks for this study). To improve their execution speed, we had deliberately avoided the use of function calls in inner loops. Moreover, the inner loops of these simulators were peppered with if-then-else constructs and were more amenable to our multiblock formation and control flow prediction techniques.

Xlisp is highly recursive, with lots of function calls. Similarly, *Spice* also made fairly liberal use of subroutine calls. Since we made no effort to capture program control flow information past function call boundaries, nor made any attempts to in-line function calls, multiblock formation was not able to help much. Likewise *Tycho* was designed to be readable and portable, and makes fairly liberal use of function calls, constraining the formation of multiblocks to predict program control flow. If functions were inlined, and if the multiblock restrictions were relaxed, allowing more targets per multiblock and perhaps allowing loops as multiblocks, we believe that many of our results will be even more impressive.

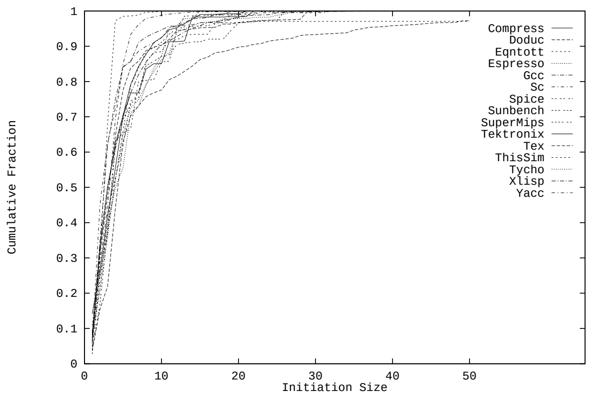


Figure 5: Cumulative distribution of initiation size without Control Flow Prediction

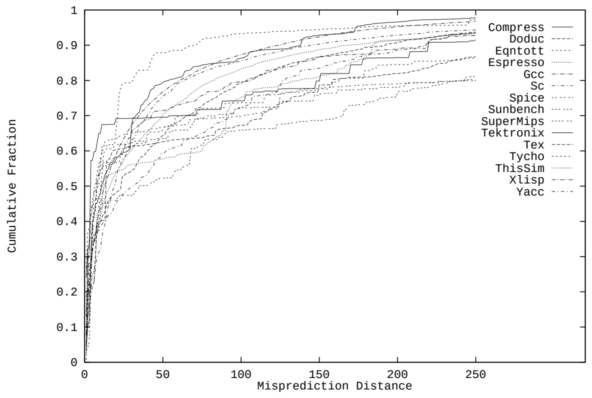


Figure 7: Cumulative distribution of misprediction distances without Control Flow Prediction

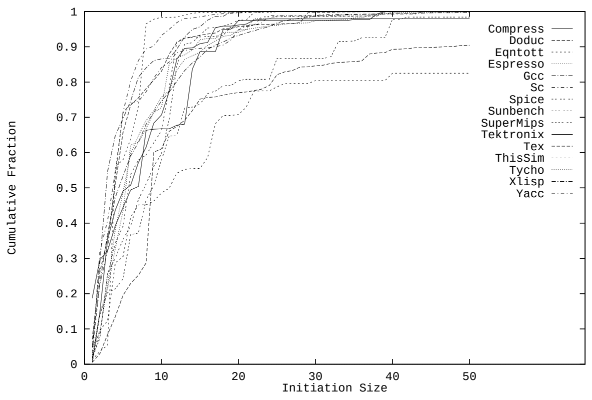


Figure 6: Cumulative distribution of initiation size with Control Flow Prediction

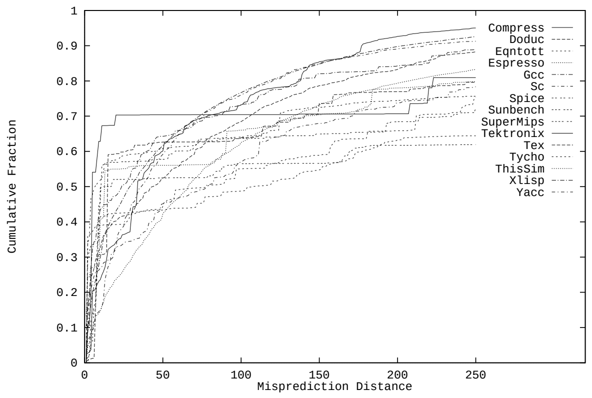


Figure 8: Cumulative distribution of misprediction distances with Control Flow Prediction

Finally, we consider the effectiveness of a CFPB. Table 5 shows the miss ratios for several CFPB configurations; 256, 512, and 1024 entries, with direct-mapped, 2, 4, 8 way set associative and fully associative configurations. The results are not surprising. A CFPB with a reasonable number of entries misses rarely, except for *Gcc* and *Tex* (both these programs have quite complex control structures with many basic blocks and multi-blocks). Due to the relatively small buffer size, associativity helps. The small miss ratios, coupled with the fact that the CFT can be quite large (c.f. Table 3), suggests that on-the-fly construction of CFT entries is preferable than storing the CFT in memory.

5. Evaluation on a Multiscalar Processor

Having analyzed the potential of control flow prediction using an abstract machine model, we now use the concept on a concrete machine model. In this section, we discuss how the multiblock concept was used in the *multiscalar* processing model (erstwhile *Expandable Split Window (ESW)* model) [9].

5.1. The Multiscalar Processor

Figure 9 shows the block diagram of a multiscalar processor. It consists of several independent, identical *execution stages*, each of which is equivalent to a typical datapath found in modern processors. The execution stages are connected together as a circular queue; the head and tail pointers to this queue are managed by a control unit (control flow predictor), which also performs the task of assigning multiblocks to the stages. It is important to note that all that the control unit does when it assigns a multiblock to a stage is to tell the stage to execute the multiblock starting at a particular PC value; it is up to the stage to fetch the required instructions, decode and execute them, until control flows out of the multiblock. Since the controller performs no instruction fetching or decoding, it can perform a control flow prediction every cycle using a CFPB, and assign a new multiblock every cycle to the

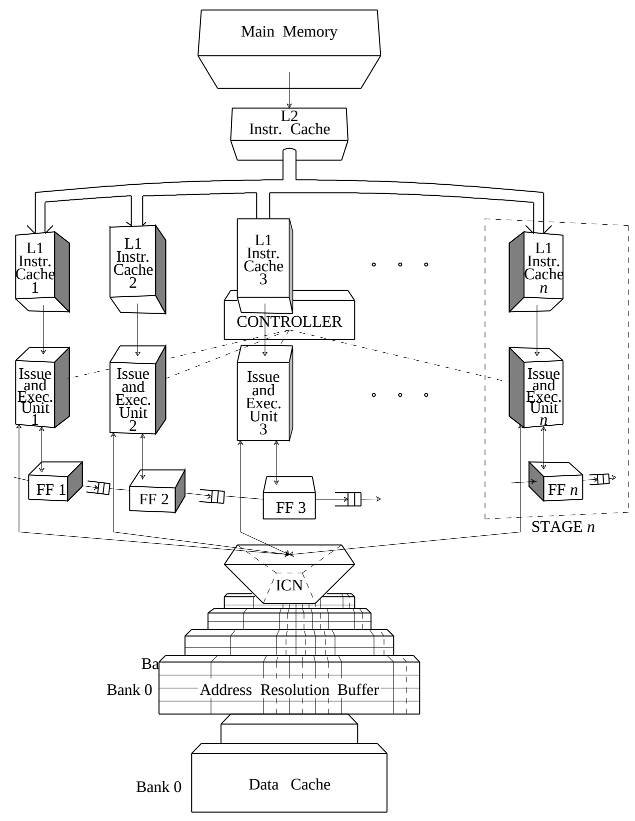


Figure 9: Block diagram of a Multiscalar Processor

Program	256 Entries					512 Entries					1024 Entries				
	Associativity					Associativity					Associativity				
	1	2	4	8	Full	1	2	4	8	Full	1	2	4	8	Full
Compress	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Doduc	4.87	3.52	2.60	2.71	2.65	3.46	1.52	0.05	0.00	0.00	1.86	0.68	0.00	0.00	0.00
Eqntott	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Espresso	2.23	1.20	0.60	0.41	0.25	1.03	0.49	0.11	0.07	0.02	0.59	0.11	0.02	0.01	0.00
Gcc	20.42	14.80	12.20	10.34	8.55	12.20	7.77	5.70	4.65	4.01	7.15	4.03	3.09	2.74	2.55
Sc	2.39	1.27	0.90	0.90	0.69	1.16	0.49	0.42	0.31	0.22	0.51	0.13	0.08	0.04	0.00
Spice	0.42	0.20	0.14	0.14	0.12	0.23	0.09	0.04	0.02	0.00	0.12	0.03	0.00	0.00	0.00
Sunbench	10.73	0.39	1.00	0.18	0.00	3.99	0.11	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00
SuperMips	3.85	0.61	0.12	0.01	0.00	1.66	0.18	0.00	0.00	0.00	1.01	0.00	0.00	0.00	0.00
Tektronix	9.36	2.64	3.16	0.32	0.00	4.68	0.81	0.02	0.00	0.00	3.33	0.70	0.00	0.00	0.00
Tex	10.47	7.51	5.25	4.45	3.63	5.45	2.84	1.93	1.00	0.48	2.97	1.03	0.29	0.14	0.06
ThisSim	0.98	0.00	0.00	0.00	0.00	0.97	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Tycho	2.34	0.01	0.00	0.00	0.00	2.19	0.00	0.00	0.00	0.00	1.58	0.00	0.00	0.00	0.00
Xlisp	10.63	5.63	2.04	0.54	0.09	3.69	1.79	0.30	0.01	0.00	2.40	0.41	0.04	0.00	0.00
Yacc	1.52	0.43	0.22	0.06	0.02	0.90	0.17	0.02	0.02	0.02	0.41	0.06	0.02	0.02	0.02

Table 5: CFPB miss ratios (percent)

execution stage at the tail (as long as there are free execution stages). This allows the execution of multiblocks to be overlapped, and multiple branches (*i.e.*, the branches within a multiblock) to be resolved per cycle. The active stages, the ones from the head to the tail, together constitute the large dynamic window of operations, and they contain multiblocks, in the sequential order in which the multiblocks appear in the dynamic instruction stream. In any given cycle, up to a fixed number of ready-to-execute instructions begin execution in each of the active stages. Hardware means are provided for forwarding values across stages, and to enforce register and memory dependencies [9].

5.2. Experimental Results

To study how much a conventional branch prediction mechanism limits the performance of the multiscalar processor, and to study the efficacy of multiblocks for the multiscalar processor, we used a detailed simulator. This simulator uses the MIPS R2000 instruction set and functional unit latencies, and models all parts of the multiscalar processor. It accepts executable images of programs, and executes them; it is not trace driven.

We conducted two sets of experiments — one with branch prediction and the other with control flow prediction. Both sets use a pattern-based prediction scheme (cf. section 4) plus a call/return stack [12] to predict the target of procedure call returns. For the former set, when a branch instruction is fetched by an execution stage, a lookup is done in a BTB to determine the branch target. Then, a dynamic prediction is performed to decide the basic block that is to be executed by the next execution stage. To give an optimistic treatment to the scheme with branch prediction, an infinite size BTB with 100% hit ratio is used. Thus, the target of a branch is assumed to be determined as soon as the branch instruction is fetched. For the latter set of simulations, after the controller assigns a multiblock to the tail execution stage, it performs a lookup in a CFPB to determine the multiblocks's targets. Then, a dynamic prediction is performed to decide the multiblock that is to be executed by the next execution stage. For our simulations, we assumed a 2-way set-associative CFPB with 512 entries. If there is a miss in the CFPB, it takes $\lceil (l-1)/f \rceil + 1$ cycles to determine the targets of a multiblock, where l is the length of the multiblock, and f is the number of instructions fetched by an execution stage in a cycle[1]. In our simulations, a multiblock can have up to 32 instructions. The multiblocks were formed based on control flow analysis only (without considering data dependency analysis), and thus our results for control flow prediction are somewhat pessimistic. The remaining parameters are given below:

- Up to 2 instructions are fetched/decoded/issued per cycle from each of the active execution stages; out-of-order execution is used in each stage. For branches enclosed within a multiblock, speculative execution is performed in the execution stage along the fall-through path.
- The data cache is 128Kbytes, direct-mapped, and has an access latency of 2 cycles. The data cache and the ARB have an interleaving factor of 32. The data cache miss latency is 4 cycles (assuming the presence of a second level data cache).
- Each stage has a 4Kword L1 instruction cache and the L2 instruction cache has a hit ratio of 100%.

For benchmarks, we used a subset of the programs used in section 4; notice that the executables have been compiled for a single-issue machine (a DECstation 3100). All benchmarks were simulated up to 100 million instructions. Table 6 presents the performance results obtained for the multiscalar processor with branch prediction and control flow prediction using 8 and 12 execution stages. The sustained IPC values take into account only the useful non-NOP instructions executed, and not the speculative instructions that were discarded. The first thing to notice from the table is that a multiscalar processor based on conventional branch prediction is essentially handicapped, whatever resources we throw at it. The sustained issue rates are poor, despite the high branch prediction accuracies (cf. Table 4). The second striking result is that except for *Gcc* and *Xlisp*, the multiscalar processor with control flow prediction sustains much higher issue rates for all programs. Finally, control flow prediction enables the multiscalar processor to traverse and resolve multiple branches per cycle, as can be clearly seen for the case of *Eqntott*: dividing the sustained IPC for 12 execution stages (4.51) with the mean initiation size without control flow prediction from Table 4(a) (3.03), we find that an average of 1.49 branches are *resolved* per cycle.

The improvements for *Gcc*, *Spice*, and *Xlisp* have not been as dramatic as the improvements for the remaining programs. Note that these were programs that did not show much improvement in the abstract machine model also. These programs need software assist, such as function inlining and more sophisticated multiblock selection. For example, in *Gcc*, parallelism exists (albeit comparatively lower), and does so in reasonable-sized instruction windows [3], but accurately establishing windows of this size is currently a problem. A major reason is due to loops that iterate only 2 or 3 times, resulting in poor prediction. We intend to combine the entire execution of such a loop into a single multiblock, and assign it to a single execution stage, rather than spread the different iterations across multiple execution stages as we do now. We expect this to

[1] In MIPS R2000 programs the last instruction in a multiblock can not cause a change in control flow (because of the delayed branching), and the hardware can determine the multiblock's targets after fetching $l-1$ instructions. Since f instructions are fetched by an execution stage in a cycle, it takes $\lceil (l-1)/f \rceil$ cycles to fetch the $l-1$ instructions.

boost the performance of *Gcc*, *Spice*, and *Xlisp*.

5.3. The Role of the Compiler

Although for our measurements we used only control flow analysis to determine the multiblocks, the optimal choice of multiblocks is intimately tied to the data dependencies and the underlying execution model. All of this information is available to the compiler which is best suited to identify these optimal multiblocks. Moreover, the code generation strategies can be changed to take advantage of control flow prediction. For example, the compiler can replicate code in appropriate places to reduce the number of targets of a multiblock, or arrange the instructions inside the multiblock so that the most likely instructions appear earlier.

Program	Sustained Instructions Per Cycle			
	Branch Prediction		Control Flow Prediction	
	8 Units	12 Units	8 Units	12 Units
Compress	1.70	1.70	2.45	2.88
Eqntott	1.83	1.84	3.30	4.51
Espresso	1.75	1.75	3.00	3.80
Gcc	1.49	1.49	1.91	2.01
Sc	1.46	1.46	2.62	2.92
Spice	1.82	1.82	2.18	2.47
Sunbench	1.41	1.43	2.70	3.11
Tycho	1.73	1.73	3.00	3.26
Xlisp	1.41	1.42	2.02	2.15

Table 6: Instruction issue rates with Branch Prediction and Control Flow Prediction

Our research group is currently modifying the Gcc compiler to produce code for the multiscalar architecture, taking advantage of control flow prediction. The modified Gcc moves up within a multiblock instructions that communicate values to later multiblocks, and moves down instructions that receive values from earlier multiblocks. This rearrangement reduces the waiting time for these instructions. Once all the dependencies are known, the compiler can identify the best grouping of basic blocks into multiblocks, and generate appropriate code for them. We expect that these techniques will improve the performance, since the code we are currently using for our experiments is scheduled for a single-issue processor.

6. Relation to Other Work

Before concluding, we would like to discuss the relationship between our concept of control flow prediction and other techniques for alleviating the impact of control dependencies for dynamic ILP processors. The most important comparison is with the concept of guarded instructions. Guarded instructions allow the compiler to produce a good static schedule by converting some of the control dependencies into data dependencies, and in some cases they can achieve the same effect as multiblocks and CFP. For example, in our Cmppt example, the branches at the ends of BB1 and BB3 could be eliminated with guarded instructions. Extensive application of guarded instructions, however, has problems. First, to use guarded instructions in nested control structures requires multiple guards, and possibly significant changes to the instruction set. Second, guarded instructions can increase the total number of instructions *executed*. For example, instructions from both the taken and the fall-through parts of a branch will be executed, with only the instructions from the actual path contributing to useful execution. Third, guarded instructions cannot achieve the full flexibility of multiblocks in CFP (for example, they cannot be used to initiate entire loops for execution).

Overall, we believe that while guarded instructions are a very useful concept, they alone cannot be used to overcome the limitations of conventional branch prediction techniques which we outlined in this paper. Fortunately, guarded instructions can be used in conjunction with control flow prediction, and this is something we are currently investigating. In fact, CFP can be used in conjunction with any software technique to enhance the parallelism for a dynamic ILP processor, such as superblock or hyperblock scheduling [6, 15]. A judicious use of these techniques can generate large nodes in a CFG that the software can optimize, and CFP can be used to navigate intelligently through the transformed CFG dynamically.

7. Conclusions

In this paper we introduced the concept of control flow prediction. This concept is a natural extension of dynamic branch prediction — use non-local information about the flow of control to go beyond branch prediction (and overcome its limits). We discussed how nodes in the control flow graph can be grouped into multiblocks to facilitate control flow prediction. Structures to carry out control flow prediction, namely Control Flow Tables and Control Flow Prediction Buffers were also described.

We evaluated the potential of control flow prediction on an abstract machine. The evaluation results illustrated the power of control flow prediction in rapidly establishing an accurate dynamic window from which ILP could be extracted. For our programs, the mean size of the dynamic window increased by a factor of 1.7, and the initiation size increased by a factor of 1.58 when control flow prediction was used instead of ordinary branch prediction. Moreover, control flow prediction allowed an average of 1.47 branches to be traversed per cycle, whereas branch prediction allowed only 1.

We also evaluated the control flow prediction concept the concrete multiscalar machine model. The results confirmed that the performance of an ILP processor that relies on dynamic branch prediction to establish an accurate dynamic window is limited. Control flow prediction consistently improved the sustained issue rates for all the programs. The improvement for 8 execution stages was in the range of 1.19 to 1.91 with a harmonic mean of 1.54; for 12 execution stages, the improvement was in the range of 1.34 to 2.45 with a harmonic mean of 1.76.

Overall, control flow prediction has tremendous potential to overcome the limitations of branch prediction in dynamic ILP processors. We view this as a first step in conveying high-level information about program behavior (which is essential to any ILP processor) to the hardware, in an effort to enhance the abilities of the hardware to exploit fine-grain parallelism. Some of the issues under investigation include sophisticated multiblock structures, enhancing multiblock formation with function inlining, and the use of CFPB information for guided prefetching of instructions to reduce the instruction cache miss penalty.

Acknowledgements

This work was supported in part by NSF grant CCR-9303030, ONR grant N00014-93-1-0465 and by an IBM graduate fellowship.

References

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[2] T. Anderson, A. Berre, M. Mallison, H. Porter III, and B. Schneider, ‘‘The HyperModel Benchmark,’’ *Extending Database Technology*, 1990.

[3] T. Austin and G. Sohi, ‘‘Dynamic Dependency Analysis of Ordinary Programs,’’ *Proc. 19th Annual Int’l Symposium on Computer Architecture*, May 1992.

[4] M. Butler, T. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, ‘‘Single Instruction Stream Parallelism Is Greater than Two,’’ *Proc. 18th Annual Int’l Symposium on Computer Architecture*, May 1991.

[5] R. Cattell, ‘‘Object Oriented Performance Measurement,’’ in *Proc. of the 2nd International Workshop on OODBMS*. Sept. 1988.

[6] P. Chang, S. Mahlke, W. Chen, N. Warter, and W. Hwu, ‘‘IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors,’’ *Proc. 18th Annual Int’l Symposium on Computer Architecture*, May 1991.

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

[8] J. Fisher, ‘‘Trace Scheduling: A Technique for Global Microcode Compaction,’’ *IEEE Transactions on Computers*, vol. C-30, July 1981.

[9] M. Franklin and G. Sohi, ‘‘The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism,’’ *Proc. 19th Annual Int’l Symposium on Computer Architecture*, May 1992.

[10] M. D. Hill and A. J. Smith, ‘‘Evaluating Associativity in CPU Caches,’’ *IEEE Transactions On Computer*, Dec. 1989.

[11] P. Y. T. Hsu and E. S. Davidson, ‘‘Highly Concurrent Scalar Processing,’’ *Proc. 13th Annual Int’l Symposium on Computer Architecture*, June 1986.

[12] D. Kaeli and P. Emma, ‘‘Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns,’’ *Proc. 18th Annual Int’l Symposium on Computer Architecture*, May 1991.

[13] M. Lam and R. Wilson, ‘‘Limits of Control Flow on Parallelism,’’ *Proc. 19th Annual Int’l Symposium on Computer Architecture*, May 1992.

[14] J. K. F. Lee and A. J. Smith, ‘‘Branch Prediction Strategies and Branch Target Buffer Design,’’ *IEEE Computer*, vol. 17, pp. 6-22, Jan. 1984.

[15] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann, ‘‘Effective Compiler Support for Predicated Execution Using the Hyperblock,’’ *Proc. of the 25th Annual Workshop on Microprogramming and Microarchitecture*, 1992.

[16] S.-T. Pan, K. So, and J. T. Rahmeh, ‘‘Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation,’’ *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, Oct. 1992.

[17] J. E. Smith, ‘‘A Study of Branch Prediction Strategies,’’ *Proc. 8th Annual Int’l Symposium on Computer Architecture*, May 1981.

[18] M. Smith, M. Lam, and M. Horowitz, ‘‘Boosting Beyond Static Scheduling in a Superscalar Processor,’’ *Proc. 17th Annual Int’l Symposium on Computer Architecture*, May 1990.

[19] T. Yeh, D. Marr, and Y. Patt, ‘‘Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache,’’ *Proceedings of ICS-7*, July 1993.

[20] T. Yeh and Y. Patt, ‘‘A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History,’’ *Proc. 20th Annual Int’l Symposium on Computer Architecture*, May 1993.

[21] T. Yeh and Y. Patt, ‘‘A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution,’’ *Proc. of the 25th Annual Workshop on Microprogramming and Microarchitecture (Micro 25)*, pp. 129-139, 1992.