Branch Instrumentation in SUIF

Cliff Young and Michael D. Smith
Division of Applied Sciences
Harvard University

Abstract

Conditional branches limit the speed of modern microprocessors. Researchers need tools to examine program branch behavior. HALT, the Harvard Atom-Like Tool, allows SUIF users to instrument conditional branch instructions in their programs. Instrumentation code enables research into the branch problem: how programs use conditional branches and how they can be handled efficiently during program execution. In particular, we use instrumentation to drive simulators of branch prediction hardware, to perform branch stream analysis, and to record statistics for profile-driven optimizations and code transformations. Predecessors of HALT and its downstream analysis phases have been the basis for our research in branch prediction.

1 Introduction

Conditional branches form barriers to execution in modern microprocessors. Most modern processors use both *pipelining* and *multiple issue* in order to execute instructions in parallel (and therefore run programs more quickly). Because the result of a conditional branch instruction is typically not known until the execute stage of a pipelined processor, the earlier stages have to guess the direction of a conditional branch. If the guess is correct, the pipeline can continue execution without interruption. Incorrect guesses, however, require that the pipeline be flushed and execution be resumed along the correct path. The number of pipeline stages that must be flushed is typically called the *mispredict penalty.* Multiple issue increases the effective mispredict penalty, because flushing a single pipe stage might result in flushing many instructions worth of work. As pipelines grow deeper and wider, the cost of misprediction grows higher, motivating researchers to find better ways to predict branches.

Researchers have approached the branch problem in two major ways: *statically* and *dynamically.* Static branch prediction schemes fix their predictions before the program runs, while dynamic branch prediction schemes vary their predictions to track the execution of a program. Prior to 1991, the best branch prediction schemes assigned a single, simple predictor per branch[1]. Static per-branch schemes predict each branch in the program to go a particular direction based either on profiles from previous runs or on heuristics [3, 6], while dynamic per-branch schemes assign a

hardware state machine to predict each branch in the program [6, 10]. More recent branch prediction schemes exploit complex patterns of behavior in individual branches or between different branches in the program. Dynamic "2-level-adaptive" schemes use branch history shift registers to capture these patterns of behavior [9, 13]. Static schemes that exploit such patterns have also been built. Krall [5] exhibited a program transformation that exploits local (per-branch) patterns of behavior; Young and Smith [14] exhibited a different transformation called static correlated branch prediction (SCBP) that exploits global patterns of behavior.

To do research on the branch problem, it suffices to condense a program run into a *branch stream*. A branch stream lists branch executions and directions in run-time order. Branch streams can be collected using hardware or software means. In the Harvard Atom-Like Tool (HALT), we follow an approach similar to Atom [2], modifying the software program so that the branch stream can be monitored by user-supplied analysis code. This approach is inexpensive because it requires no extra hardware, but it can introduce distortions into the system, because the instrumented program runs more slowly than the original program. In studies of entire systems, it is important to compensate for these instrumentation effects. For most of the user-level programs typically studied in the branch prediction literature, these effects are usually ignored.

HALT works as a SUIF pass [12], adding calls to analysis code either to original SUIF files or to SUIF machine library files [11]. Since the output of HALT is also SUIF code, further SUIF passes may be run after inserting analysis code. Analysis code is written by users, allowing different functions to be performed. We typically use minimal analysis code that writes a branch trace to a file, then we post-process the branch trace with various analysis programs. Much more complicated on-line simulations can also be performed. After collecting and analyzing traces, we can then transform the program to take advantage of the trace information. We implement both hardware simulations and the SCBP code transformation as downstream analysis phases.

The next section discusses HALT in more detail. Section 3 describes using HALT for hardware simulations of branch prediction schemes, and Section 4 describes using HALT to drive a version

1. We distinguish between prediction schemes and predictors. Predictors are typically very simple mechanisms, like static-always-predict-taken, or the dynamic 2-bit-counter state machine. Prediction schemes gang together multiple predictors into a more complex but more accurate system.

of the SCBP code transformation. Section 5 describes current status and availability of our suite of branch research tools.

2 HALT

The principle behind software instrumentation is very simple: add code to the existing program that processes the desired information. To do this, we follow Atom's approach of allowing users to build their own analysis routines. These analysis routines are linked with the final program, allowing users to specify arbitrary work on the instrumented information. We are currently much more limited than Atom in what we can instrument, supporting only initialization, branch, function entry and exit, and termination routines. Instrumentation can be applied selectively to procedures or individual branches, so the whole program need not run inefficiently.

Branches to be instrumented should be labeled with a "branch_unique_num" annotation. The annotation should contain just a single non-negative integer which identifies the branch. A separate labeling pass currently labels all branches in a SUIF file; users may also use the SUIF Visual Browser [7] to select branches to instrument. Multiple static branches may share the same number, but those static branches will then be indistinguishable to the analysis code.

HALT works by inserting calls to analysis routines as directed by the branch_unique_num annotations. HALT ensures that calls to analysis code preserve volatile program state across the instrumentation point. This is currently done conservatively, so, for example, all caller-saved registers are saved before calling analysis code. Later versions of HALT may use more sophisticated live-range information to determine which registers must be saved. HALT instruments function entries and exits if the function body contains any instrumented branches.

The SUIF file produced by HALT contains external references to the following functions:

```
void _bt_startup(void);
void _bt_cleanup(void);
void _bt_call(void);
void _bt_return(void);
void _bt_record(unsigned brnum, int cond);
```

These functions are called at the points that correspond to their names: `_bt_startup` and `_bt_cleanup` are called at the entrance and exit of `main()`, allowing the analysis routines to initialize or clean up variables and open or close files. `_bt_call` and `_bt_return` are called

at the entrance and exit of functions containing instrumented branches. `_bt_record` is called before each instrumented branch instruction. The arguments to `_bt_record` give the branch number from the branch_unique_num annotation and the direction the branch went at run time. These functions must be linked with the HALT-transformed object file; they do not need to be compiled under the SUIF compiler.

By defining the five analysis routines above, researchers can construct whatever kind of branch analysis software they require. We also supply sample analysis routines with HALT. The sample analysis code writes a trace file to an unused file descriptor. By redirecting this file descriptor (either piping to another program or saving to disk), we can attach different analysis passes. The trace format is minimal, listing just branch number and direction[2]. More complex traces can certainly be built, but the current downstream passes expect this brief format. By saving traces to disk, we can also ensure experimental repeatability.

HALT is not as general as ATOM. We expect that we can expand HALT in the future to instrument other interesting parts of program behavior. For our current research needs, it suffices to support branch instrumentation. Since HALT already supplies instrumentation-insertion routines, it will be simple for other researchers to extend HALT instrumentation to support their interests. Harvard will serve as a clearinghouse for HALT releases, should researchers want to share their analysis and instrumentation code.

3 Hardware Simulation

Hardware simulation is simple to perform under HALT. Using the sample analysis code, users can build branch-profiling executables. Traces from runs of these executables can be saved to disk, ensuring experimental repeatability. The HALT distribution includes a trace-driven hardware simulator that allows researchers to model recent hardware branch prediction schemes like GAs, PAs, gshare, and hybrid schemes [1, 8], and also to experiment with building other scheme configurations. A simple library of 2-bit counter and history register classes makes modification easy. This library and simulator were used in our earlier work on branch prediction [4, 14, 15].

2. Calls and returns are indicated by condition values -1 and -2, respectively.

4 Static Correlated Branch Prediction

As originally implemented by Young and Smith [14], SCBP has four major stages: profiling, local analysis, global analysis, and code layout. HALT provides support for profiling, and we rely on other SUIF passes to perform some parts of layout. The traces from HALT-transformed executables can be redirected to analysis passes and further SUIF code transformation passes. This section discusses details of our implementation of SCBP, which uses HALT for profiling.

4.1 Local Analysis

SCBP uses a data structure called a *history tree* to describe each branch in the program. A history tree is a compact representation of the set of all paths of length *k* (*k* is a fixed parameter called the *history depth*) that reach a branch in a program. By combining common path prefixes into interior nodes, we can represent this set as a tree. In HALT, history trees are constructed by an analysis pass that consumes a trace and produces a list of history trees, one for each program branch. By specifying an existing history tree file, one can combine multiple training runs additively. More sophisticated means of combining training runs are not yet supported. History tree collection currently takes *O(nk)* time, where *k* is the history depth and *n* is the number of dynamic branches in the trace.

After collecting a history tree, SCBP performs local minimization (also called local analysis). In most cases, program branches behave the same regardless of the paths by which they are reached. For these cases, simple static branch prediction suffices to predict each branch. Only when a branch behaves differently on different reaching paths must more complex code transformations be performed. The recursive pruning algorithm described by Young and Smith ensures that only beneficial correlation paths are considered by the SCBP code transformation.

4.2 Global Analysis and Code Transformation

After local minimization, global analysis determines the smallest number of copies of each original program basic block that need to be made, and connects those copies appropriately. The connected copies are then output as a new SUIF file. Downstream SUIF passes can then ensure efficient code layout.

The HALT implementation of SCBP collects history trees more efficiently than the method used in the original Young and Smith paper. Instead of building all paths in the history tree, then

recording those paths that were executed, our new implementation of SCBP builds and records only the executed paths. This saves a considerable amount of storage both in memory and on disk. However, it complicates the reconciliation algorithm, which makes reconnecting matching copies more complex.

We have glossed over many details of the SCBP algorithm to avoid duplicating earlier publications. For more information about the SCBP algorithm, see Young and Smith.

5 Summary and Future Work

HALT provides a basis for research into branch prediction and branch handling in modern processors. Since it is based on SUIF, it can use SUIF's extensibility to support work on a variety of platforms, and even to target experimental architectures. Because it also works with the SUIF machine instruction library, it can be used to perform experiments at the instruction set and micro-architecture level. We believe that the difficult part of writing HALT involves supporting safe instrumentation calls on new architectures; applying instrumentation to other interesting targets (e.g. memory operations, multi-way branches, and call graph profiling) is simple once the instrumentation routines have been built.

The HALT distribution currently includes a simulator for hardware branch prediction schemes; a later version will include the implementation of the SCBP code transformation in another SUIF pass. HALT works with both the original SUIF and SUIF machine instruction libraries. HALT currently supports only MIPS architecture machine instructions, but it will support HPPA and x86 machine instructions when those code generators become available. The current distribution of HALT is available at ftp://eecs.harvard.edu/pub/hube/halt.tar.gz.

We continue to work on HALT and SCBP support in SUIF. A number of areas make for obvious extensions, and we are working on them now. We hope to support inlining and other mechanisms to exploit cross-procedure-call correlation. We plan to add more sophisticated support for multiple data sets and for multiway branches. We expect to build research instruction schedulers that will work downstream from the SCBP code transformation, taking advantage of improved prediction accuracy. And we plan to add more general instrumentation functions to HALT as we see the need.

6 References

[1] P. Chang, E. Hao, T. Yeh, and Y. Patt, "Branch Classification: a New Mechanism for Improving Branch Predictor Performance," in *Proc. 27th Annual ACM/IEEE Intl. Symp. and Workshop on Microarchitecture*, November 1994.

[2] A. Eustace and A. Srivastava. "ATOM: A Flexible Interface for Building High Performance Program Analysis Tools". *Proc. Winter 1995 USENIX Technical Conf. on UNIX and Advanced Computing Systems*, pp. 303–314, Jan. 1995

[3] J. Fisher and S. Freudenberger, "Predicting Conditional Branch Directions From Previous Runs of a Program," *Proc. 5th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, Oct. 1992.

[4] N. Gloy, M. Smith, and C. Young. "Performance Issues in Correlated Branch Prediction Schemes," *Proc. 28th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, November 1995.

[5] A. Krall, "Improving Semi-static Branch Prediction by Code Replication," *Proc. ACM SIGPLAN '94 Conf. on Prog. Lang. Design and Implementation*, Jun. 1994.

[6] J. Lee and A. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, 17(1), Jan. 1984.

[7] J. Lim, "A Visual Browser for SUIF", *Proc. First SUIF Compiler Workshop*, Stanford University, Jan. 1996.

[8] S. McFarling, "Combining Branch Predictors," *WRL Technical Note TN-36*, June 1993.

[9] S. Pan, K. So, and J. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *Proc. 5th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, Oct. 1992.

[10] J. Smith, "A Study of Branch Prediction Strategies," *Proc. 8th Annual Intl. Symp. on Computer Architecture*, Jun. 1981.

[11] M. D. Smith, "A machine instruction library for SUIF," *Proc. First SUIF Compiler Workshop*, Stanford University, Jan. 1996.

[12] Stanford Compiler Group, "The SUIF Library: A set of core routines for manipulating SUIF data structures," available for anonymous FTP from suif.stanford.edu.

[13] T. Yeh and Y. Patt, "Two-Level Adaptive Branch Prediction," *Proc. 24th Annual ACM/IEEE Intl. Symp. and Workshop on Microarchitecture*, Nov. 1991.

[14] C. Young and M. Smith, "Improving the Accuracy of Static Branch Prediction Using Branch Correlation," *Proc. 6th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, Oct. 1994.

[15] C. Young, N. Gloy, and M. Smith, "A Comparative Analysis of Schemes for Correlated Branch Prediction," *Proc. 22nd Annual Intl. Symp. on Computer Architecture*, June 1995.