

IDtrace - A Tracing Tool for i486 Simulation

Jim Pierce¹ and Trevor Mudge
University of Michigan

1. Jim Pierce was supported in this work by a grant from the Intel Corp.

Abstract

This paper describes IDtrace, a binary instrumentation tool which produces execution traces for the ix86 instruction set architecture. Long execution traces can be generated quickly and easily for input to a wide variety of performance evaluation tools. Issues involved in the construction of such a tool are listed along with illustrations of the uses of different generated traces. One example observes the behavior of a branch prediction technique and another compares the cache behavior of the i486 with that of the MIPS R3000.

1 Description of IDtrace

Trace driven simulation plays an important role in the design and tuning of computer architectures. Thus tools which can produce long traces quickly and easily are valued by system designers. This paper discusses IDtrace[1], a software tracing tool for the i486 which produces input data for a wide variety of performance evaluation tools including code profilers and branch prediction and cache simulators. IDtrace instruments program code so that traces are generated during the program's execution. Thus long traces can be produced quickly and easily without additional hardware. Software tracing tools can instrument programs at the source, object, or executable level. Modifying the executable, called late code modification, requires text disassembly, code modification and relocation, and binary file reconstruction. While this approach is the most difficult, instrumentation does not require the source files, the library code is automatically instrumented, and the tool is almost trivial to use.

IDtrace uses late code modification to instrument Unix SysV R4 ELF statically-linked binaries created by Intel/AT&T and USL CCS C compilers. No sources or symbol tables are needed. Applications can be instrumented to output profile, memory reference, or full execution traces. The resulting binary is about twelve times larger than the original and execution time for the instrumented program ranges from two times that of the original for a profile trace to about twelve times for a full execution trace.

2 Implementation Issues

IDtrace inserts binary code before each basic block and data memory referencing instruction and relocates all control instructions to account for text expansion. Some inherent architectural attributes of the i486 and compiler actions combine to make binary instrumentation difficult. The i486's complex instruction set contributes to a large, slow instrumented binary. The i486 has 182 memory referencing instructions and many reference memory multiple times with different addresses. In addition, the `rep` prefix allows an indeterminate number of references per instruction. Furthermore, indirection causes relocation problems. Compiler code structure must be deciphered to derive the location of jump tables since code relocation requires that their lists of addresses be updated. Indirect calls cannot be relocated at all prior to runtime since the target address is unknown. These instructions are handled by a runtime lookup into a table containing the original and instrumented function addresses. Other difficulties arise from variable instruction lengths and the small register set of the i486.

3 Experiments Using Traces

In this section we give a sampling of the capabilities of IDtrace. All experiments were run on an Intel 50MHz i486 machine running USL Unix SysV R4. The programs used are the C benchmarks from the SPEC92 benchmark suite.

3.1 Branch Prediction

The branch trace output can be used as input to a branch prediction simulator to measure the performance of different branch prediction algorithms. During our preliminary study it was noted that most prediction misses are caused by only a few conditional branches, i.e., while most branches are almost always predicted correctly, a few rarely are and cause most of the misses. The following results were generated by a prediction simulator using a dynamic 2-bit saturating counter prediction algorithm with a history table of 1024 2-bit entries. The data in Table 1 shows that a very small percentage of conditional branches (usually less than 5%) cause 90% of the prediction misses. Further examination revealed that the misses are caused by

multiple branches contending for the same counter in the history table and just a few branches behaving in bad, cyclical patterns which a counter cannot predict well.

3.2 Cache Simulation

The availability of IDtrace makes it possible to compare cache performance of two very different approaches to instruction set design typified by the i486 and the MIPS R3000. To maintain compatibility with earlier members of the ix86 family, the i486 has variable length instructions and fewer CPU registers (8 vs. 32) resulting in higher code density but more memory data references. We found these differences not to be as serious as might be expected because of the i486 stack acting as an extended register file and usually residing in the cache.

Program	Unique Jccs	Branch Prediction Accuracy	Number Causing 90% of Misses
eqntott	333	82.8%	2 (1%)
ear	402	94.9%	3 (1%)
compress	217	84.8%	6 (3%)
sc	1427	91.4%	44 (3%)
xlisp	425	82.9%	19 (4%)
espresso	1434	84.6%	134 (9%)

TABLE 1. Mispredicted Branch Distribution

The R3000 traces are generated by Pixie [3] on a DECstation 5000 and the cache simulator used is a modified version of the multicache simulation tool, Tynero [2]. Table 2 shows some preliminary observations in that the R3000 has roughly two to four times the miss ratio of the i486 on most benchmarks. On smaller caches the number of misses is about the same, however the i486 makes about twice as many references. On larger caches the R3000 has more misses which increases the miss ratio disparity. This implies that the R3000 program has a larger working data set. The factor of two reference difference is due to the lack of registers in the i486, which results in a much greater degree of spillage. This is handled by pushes and pops and base pointer references off the stack. Once the

top of the stack is resident in the cache few of these references will be misses.

Line Size (bytes)	Misses in thousands (% Ratio)			
	i486: 290M references		R3000: 134M references	
	32	64	32	64
8K 1-way	9329 (3.2)	8093 (2.8)	9836 (7.4)	8371 (6.3)
8K 2-way	7038 (2.4)	5036 (1.7)	8310 (6.2)	6541 (4.9)
8K 4-way	6566 (2.3)	4607 (1.6)	7462 (5.6)	5506 (4.1)
64K 1-way	1112 (0.4)	1030 (0.4)	1892 (1.4)	1843 (1.4)
64K 2-way	607 (0.2)	551 (0.2)	1314 (1.0)	1204 (0.9)
64K 4-way	504 (0.2)	447 (0.2)	1108 (0.8)	1030 (0.8)

TABLE 2. Cache misses running espresso on i486 and R3000 for different cache configurations.

Cache Config.	Push Write Miss Ratio	Non-P/P Write Miss Ratio	Pop Read Miss Ratio	Non-P/P Read Miss Ratio
8K 2-w	1.0%	4.3%	0.5%	2.4%
8K 4-w	0.5%	3.7%	0.2%	2.1%
16K 2-w	0.4%	2.8%	0.2%	1.1%
16K 4-w	0.3%	2.4%	0.1%	1.2%

TABLE 3. Miss ratios for push, pop, and non-push/pop references running espresso on the i486.

To confirm this IDtrace was modified to output special tags for push and pop stack references and Tynero was modified to record separate counts. Table 3 shows that the miss ratio of push/pop references is far less than that of non-push/pop references for larger caches. Thus the large number of stack references generate few misses on the i486 and the number of misses on the two processors are roughly equivalent.

4 References

- [1] J. Pierce, "IDtrace: A Trace Generation Tool for the ix86 Instruction Set," Technical report, Intel Corp., Hillsboro, OR, Sept. 1992.
- [2] J. Quinlan, and K. Lai, "Tynero: A Multiple Cache Simulator," Technical Report, Intel Corp., Hillsboro, OR, May 1991.
- [3] M. Smith, "Tracing with Pixie," Technical Report, Center for Integrated Systems, Stanford University.