lDtrace — A Tracing Tool for i486 Simulation

Jim Pierce[1]

Trevor Mudge

University of Michigan

Technical Report CSE-TR-203-94
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan

Abstract

This technical report describes IDtrace, a program that produces execution traces for ix86 instruction set architectures using late-code modification. IDtrace provides a low cost method for producing input data for a wide variety of performance evaluation tools such as code profilers, branch prediction simulators, and cache simulators. IDtrace can generate several types of output traces: profile, branch, memory reference, and full execution traces. It currently runs on ix86 SysVR4 Unix systems.

The report briefly introduces various trace generation methods and discusses the advantages and disadvantages of late code modification over other code instrumentation techniques. The majority of the report describes the use of IDtrace and outlines the formats for all generated trace files. In addition, the issues involved in constructing such an instrumentation tool, including the challenges imposed by the i486's CISC-like features, are discussed. Architectural attributes such as the large number of memory referencing instructions, the complex instructions and addressing modes, and the variable instruction lengths make instrumentation difficult and sometimes impossible at the binary level. These problems and their possible solutions are discussed.

Finally, the report includes some experimental results to illustrate the applicability of IDtrace. The experiments involve profiling the instructions frequencies of a subset of the SPEC92 benchmarks, evaluating a common branch prediction technique, deriving optimal cache configurations based on several application programs, and comparing the cache behavior of the i486 with that of the MIPS R3000.

1. This work was supported by a grant from the Intel Corporation.

Table of Contents

1.0 Introduction

Trace driven simulation plays an important role in the design and tuning of computer architectures. If highly accurate traces are required that capture operating system effects and which are not distorted by the collection mechanism, hardware monitors are usually the only solution [3][10]. However, it is often sufficient, particularly for initial design studies, to collect traces using software tools that are hosted on a system having a CPU that matches the instruction set architecture (ISA) under study. This paper discusses IDtrace, a software tracing tool for the i486 that produces input data for a wide variety of performance evaluation tools including code profilers, branch prediction simulators, and cache simulators. In keeping with most software tracers it is quite fast (10-12 times slower than real time) and can be used in applications such as secondary cache simulation which require very long address traces [1][11]. It is also possible to use it in situations where trace sampling is considered sufficiently accurate [12]. Traces are produced by first instrumenting a program with IDtrace and then running it on an existing system.

We noted above that hardware monitoring can give the most accurate and general traces. There are a number of ways of obtaining traces through hardware monitoring. Perhaps the most straightforward is to attach a logic analyzer to the processor pins or system bus, detect special situations, and use the analyzer to store these events. This approach was taken by Nagle et al. to measure TLB misses on a DECstation 3100 [10]. An alternative approach is to add special processing and recording hardware to the system bus which monitors traffic and detects and saves selected events. BACH is an example of a i486-based tool which gathers traces in this manner [3]. While these methods present the fastest way to gather information, their speed can present a problem. A full address trace is difficult to gather and store because of the high data rate at which the references are produced. Tracing must be interrupted each time the recording capacity of the monitoring device is reached. Furthermore, these methods require the user to have both significant hardware expertise and the capability to obtain or fabricate the monitoring tools.

Software monitors or instrumentation tools modify a program or its execution so that traces or statistics are recorded at runtime. As a whole, these tools are much slower than hardware monitoring and they cannot observe all hardware events. Furthermore, they are not unobtrusive. The tool's use affects the simulation and these effects must be taken into account and/or minimized. However, long continuous traces can be gathered easily and cheaply. An early approach to software monitoring was to use an OS trap on every instruction to record execution information [4]. This method is extremely slow and for instruction sets with a high density of memory references it is almost useless. A more common software approach is the use of a code instrumentation tool. Such a tool adds additional code to a program so that when the modified program is executed it will output some sort of trace while maintaining its original functionality. Techniques of code modification are discussed in the next section. The remainder of the paper will discuss an instrumentation tool for ix86 architectures called IDtrace. Section 3.0 shows how to use IDtrace and explains various options. Section 4.0 outlines the issues involved and the challenges faced in building such an instrumentation tool and Section 5.0 discusses its limitations. Finally, Section 6.0 contains the results of several experiments illustrating how IDtrace can be used to profile the instructions frequencies of a subset of the SPEC92 benchmarks, to evaluate a common branch prediction technique, to find optimal cache configurations with respect to several applications, and to compare the cache behavior of the i486 with that of the MIPS R3000. The appendices contain descriptions of file and output trace formats.

2.0 Software Instrumentation Techniques

Program instrumentation can be done at several levels. The easiest is source level instrumentation where code is added to each assembly level source file and the code is reassembled to produce the instrumented objects. A second method is to start with the program's compiled objects. These are disassembled, modified along with the relocation and symbol tables, and then linked using the system's

linker. A third approach is to modify the final executable. This requires disassembly, modification, and code relocation. Each of these methods have advantages and disadvantages which we briefly survey below.

2.1 Source Code Modification

Assembly level source modification is the easiest to perform. Most compilers generate object code in two phases, first the conversion to assembly and then a call to the machine's assembler to create the binary object file. Relatively little effort is required to add additional instructions to the assembly file to record runtime information such as memory references branch targets. The existing assembler and linker can be used to create the executable. While this tool is relatively quick and simple for the tool designer to build it is the most difficult to use and maybe the least useful, for the following reasons:

- Source Availability — The user may not always have access to the source code.
- Source Size — If the source is large, repeated compilation of modified and unmodified sources is time consuming.
- Library Routines — Under normal circumstances library calls will not be annotated since they are not part of the source. This could distort results. Each library module could be also be modified but this requires access to the library sources, space for modified and unmodified modules, and more time for multiple compilations.
- User Knowledge — The user of the tool must understand the compiler's code generation process to the extent of knowing which library objects are needed, how special code inside the libraries can be instrumented, and how memory is allocated for variables and tables.
- Kernel Routines — Kernel code is difficult or impossible to instrument in this manner.

There are many examples of tools which instrument code at the source level. One of the best known is MPtrace written by Eggers et al. which runs on Sequent i386-based shared memory multiprocessor systems [2]. The generated trace is only a subset of the full execution trace to allow trace storage and to minimize time dilation during program execution. The latter is important for multiprocessor simulation which is the goal of MPtrace. While the preprocessor instruments the assembly output of the compiler it also creates a "roadmap" file for the later reconstruction of the trace.

A novel twist on source code modification is to perform the code instrumentation within the compiler. One advantage this approach has over assembly code modification is that the compiler can identify high-level code constructs and can use this knowledge to reduce the amount of instrumentation to construct a partial trace. Larus developed this approach for tool called AE [8]. A modified GNU C compiler generates instrumented code which upon execution will output a skeleton trace. The compiler also creates a schema file which is later used to reconstruct the full trace.

2.2 Object Code Modification

Rather than instrumenting the assembly source it is possible to wait and modify the compiled object instead. This can be done by a sophisticated linker which includes a module rewriter. During the linking process each object is passed to the rewriter which performs the necessary code modifications and handles code and data relocation. The latter is primarily a task of noting location changes in the object's relocation dictionary and symbol table. The modified objects are then passed back to the linker proper and are combined into one executable as usual. Recompilation of the source code is unnecessary. The presence of the relocation data and symbol table make relocation straightforward. Postponing modifica-

tion until the executable stage when this information is missing makes relocation much more difficult and sometimes impossible.

There are several tools which perform link-time modification. Mahler is a back-end code generator and linker for Titan, a DECWRL experimental workstation [19]. The module rewrite linker can perform intermodule register allocation, basic block counting and address trace generation, and instruction pipeline scheduling. Code and data relocation is done as described above. Another tool, Epoxie, relies on incremental linking which produces an executable containing a combined relocation dictionary and symbol table [18]. Its advantages over Mahler are that the standard linker can be used and data sections remain fixed so data relocation is not necessary. Epoxie produces address traces and block statistics.

Since binary code is manipulated, a link-time modification tool must have the use of a disassembler and will be more difficult to build than a source code annotator. From the user's point of view, many of the compilation complexities have been hidden but the user still must know how to perform the link step. Furthermore, an executable binary alone still cannot be instrumented. Often if the user does not have access to the source code neither will he or she have all the object modules.

2.3 Executable Modification

The third time at which code can be modified is after the objects have been compiled and linked into one executable. This is referred to as late code modification [18]. Its virtue is its ease of use. It is now trivial for the user to collect trace and other runtime information. In principle, any binary can be instrumented with such a tool without the user knowing anything about the assembly code constructs or compilation process. Also, the user need not have access to any source or library code. Unfortunately, such a tool is much more difficult to build than the ones described earlier. Disassembly is required to instrument instructions along with the ability to relocate user and library code. By trying to alter the code so late in the compilation process much information about the program structure needed for relocation is lost and sometimes instrumentation is impossible. At the very least, additional runtime overhead is likely to be incurred to overcome this loss.

One of the earliest tools of this type is Pixie which runs on MIPS R3000-based systems [9]. It instruments an R3000 binary and produces a runable `.pixie` file. When the new binary is executed trace and statistics files are also generated. These files can be examined and various execution statistics generated by `pixstats` or fed into a cache simulator [15]. Pixie is compiler-independent and will instrument most but not all programs. One drawback of Pixie is that it suffers from significant runtime overhead. The reasons for this are described later. Another tool, Nixie, was written to reduce this overhead [18]. It is functionally equivalent to Pixie on a restricted set of programs. Nixie makes various assumptions about code structure to improve performance and reduce modified code size. Sometimes these assumptions are incorrect and the modified program fails to run. In addition, many of assumptions are based on code sequences generated by a particular compiler thus making Nixie compiler dependent.

Goblin is a tool similar to pixie built for the IBM RS/6000 architecture [16]. It instruments user code only and has been used for instruction mix, register liveness, and basic block statistical studies. Finally, Sun has a similar family of tracing tools for Sparc executables called Spixtools[17].

Despite the dominance of the ix86 family of architectures in the marketplace, a similar tool for it has not been widely available. We have created IDtrace to instrument ix86 programs at the executable level to allow low-cost and fast analysis of various architectural trade-offs. It can produce a variety of types of traces including profile, memory reference, and full execution traces. It is very simple to use: no source, special hardware, or knowledge about libraries, linkers, or compiler constructs is required. IDtrace can instrument a stripped binary, no symbol table is needed. The disadvantage is that IDtrace cannot instrument all programs. Because of information loss in the translation process it must make certain assumptions about the code structure to successfully create a new binary. Hand coded assembly might contain

unrecognized control constructs which would produce erroneous instrumentation. Potential problems are described in Section 5.0. Currently IDtrace runs on Unix SysV R4 ELF binaries created using the Intel/AT&T and USL CCS C compilers. An earlier version ran on Sequent's DYNIX/ptx systems.

In summary there are three times at which code instrumentation can be performed. Binary instrumentation requires only the final executable, will instrument all library code automatically, and is simple to use. However, it is the most difficult tool to build and may not be able to instrument all programs. On the other hand, an assembly code instrumentation tool is relatively easy to build but more difficult to use. The user must maintain instrumented versions of all source files and possibly obtain instrumented versions of object files. Finally, object code instrumentation involves intermediate complexity to build and all programs can be instrumented since the system's linker can be used for relocation. The source code perhaps is not needed but it is unusual to have the object files but not the source files.

3.0 Method Of Use

This section describes how to use IDtrace. Included are the steps necessary to generate traces, a description of the available options and postprocessing tools, and a listing of the SysV instructions required to pipe the trace directly to a postprocessor. Figure 1 shows the relations between the various generated files, traces, and tools.

3.1 Trace Generation

1. Create a statically linked executable of the test program.

This is done using the -dn flag with the cc or icc compiler. IDtrace will not correctly instrument a dynamically linked executable. Thus

```
cc -o bench -dn bench
```

will create the proper executable.

2. Run IDtrace

Idt takes the binary as input and returns a new instrumented version. Various options can be given to produce different types of traces.

```
idt [options] bench
```

When idt completes two new files are created, **`bench.idt`** and **`bench.blk`**. The former is the new executable and the latter is an array of basic block structures with one structure corresponding to one basic block in the program. Each structure contains the basic block number, address, number of instructions, and a list of the instructions in the basic block. The .blk file is used by vcount or other postprocessors to generated dynamic profile data. The exact format of the .blk file can be found in A.2 on page 21.

3. Execute the new binary

Once **`bench.idt`** has been created it can be executed exactly like **`bench.`** Two more files will be created: **`bench.trc`** and **`bench.cnt.`**

The .trc file is the output trace and can be saved or piped directly to a simulator. The .cnt file holds the dynamic execution count of each basic block and is described in A.2 on page 21.

3.2 IDtrace Options

The options are:

- -p Profile trace (default)
- -b Branch trace
- -c Cache trace output with line size equal to default value (currently 16 bytes)
- -l <num> Cache trace output with line size equal to <num> bytes
- -e Execution trace
- -B <num> Maximum number of blocks
- -h This option menu
- -H <fname> Use hint input file
- -r Distinguish between regular and block (repeat) accesses
- -s No block count overflow check for short trace
- -t Binary fully instrumented but no trace file produced
- -v Verbose mode during binary instrumentation

The -s option will produce a somewhat faster executable because block count overflow tests are not added to the code. Using this option with traces of over 4 billion instructions could cause an overflow of the basic block counts and thus give incorrect results. The -r option is used for block access analysis and will cause different tags to be used for data references from a rep prefixed instruction. A special repeat_-end entry will also be output after the last reference of a rep instruction. This option can only be used in conjunction with the -c or -l options and is automatic with the -e option. IDtrace will report when the -B option is necessary. Since memory must be allocated for basic block counts prior to analyzing the code, IDtrace will estimate the space needed by looking at the file's text size. If it underestimates the number IDtrace will report that the user must specify a value greater than its estimation and idt must be rerun using this option. Compiler dependency problems can be alleviated with the -H option. A hint file can be created to give IDtrace relocation information which it cannot derive itself. The specifics of this file can be found in Section 5.2. The -t flag specifies that the binary should be fully instrumented but the trace buffer output is disabled. This allows the measurement of the speed of the instrumented code without the collection of a potentially large trace. Finally, the verbose mode is used primarily for debugging IDtrace. It presents the addresses of various functions in the original and new binary, the size and location of tables and buffers, etc. The various traces and their formats are described in A.1 on page 18.

3.3 Postprocessing and Data File Verification

Several basic postprocessing tools accompany IDtrace. They are given as examples on how to interpret the various trace and information files. One tool is **`vcount`** which uses the .blk and the .cnt file to list the dynamic instruction count, instruction mix information, and some basic block information. Tools

which use the .trace file are **vtrace** which allows viewing the .trc file and **rtrace** which accepts a trace generated using the -r option and outputs the number and sizes of block accesses. Type

vcount bench, vtrace bench, or **rtrace bench**

to run the postprocessing tools.

Another tool, **idt2kt**, converts an execution trace into KT format. It uses the .trc and the .blk files along with the original binary to create a .kt file. The format of this file is outlined in Appendix B on page 23. Idtinfo is a utility which helps to maintain the correct versions of the .idt, .cnt, and .blk files. Typing

idtinfo bench

will look for the various created files, output the options used when created the .idt file, and check the magic numbers of the data files to verify that they were generated by the current .idt file and the current original binary.

3.4 Piping the Results

The trace file can be piped directly into a simulator using the mknod(2) command. The trace file will not be saved for future use. To pipe the output trace of the binary bench directly to the postprocessor **vtrace** type the following:

>mknod bench.otr p - This creates bench.otr as a pipe file

>bench.idt & - Run bench.idt in the background

>vtrace bench - Run processing program that uses bench.otr

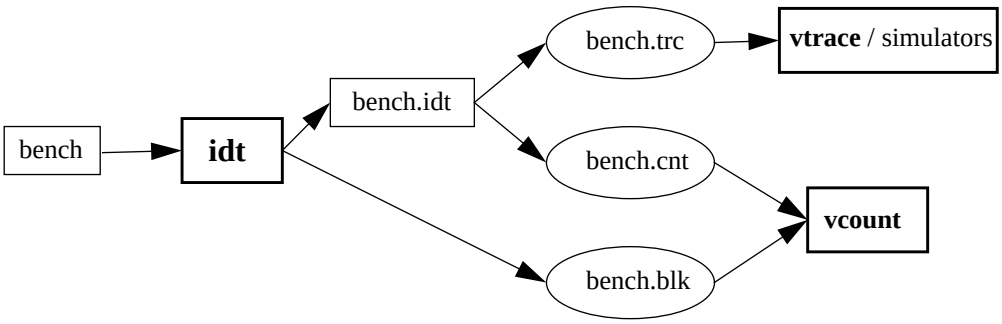

FIGURE 1. IDtrace programs and files - Rectangles are executables, ovals are data files produced by IDtrace, boldface names are IDtrace tools.

4.0 IDtrace Design Issues

This section describes the operation of a typical late code modification tool, how the target instruction set architecture (ISA) can affect the complexity of the tool, and the particular problems encountered in creating IDtrace.

A late code modification tool must disassemble the text section of the binary, insert binary code before each basic block and data memory referencing instruction, relocate all control instructions to account for text expansion, and combine the text and data sections along with some working tables and buffers into a new executable file. Only the text section is modified; any section containing data cannot be changed or moved in memory since it is impossible at this stage to relocate data references. The added code before a memory reference calculates the effective address and sends the address and type of reference to a reference buffer. The code added at the beginning of a basic block outputs profile data and instruction references and flushes the reference buffer if necessary. Creating the new executable can be tricky depending upon the OS loader's faithfulness to the executable file specifications. IDtrace is forced to combine all modified sections along with the original data into one expanded data section due to many loaders' inability to interpret all but the simplest executable structure.

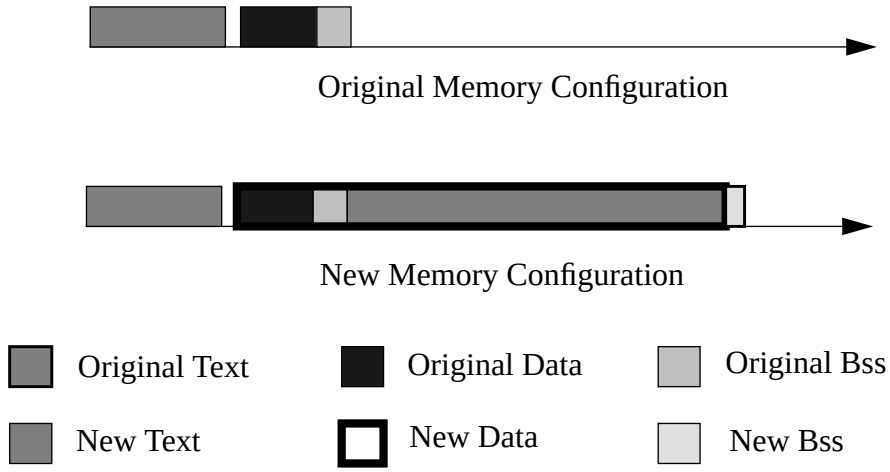

FIGURE 2. Original and new binary file configuration.

There are some inherent architectural features which simplify the above procedures. Others pose difficulties and add complexity.To illustrate these, below, we have compared and contrasted some characteristics of the ix86 ISA and the R3000 ISA[1] which affect memory reference instrumentation:

- Instruction Set — The i486 has approximately 296 different instructions, 182 of which can reference memory. Many of these instructions can reference two addresses or perform both a read and write. The R3000 has around 90 instructions including floating point and only 14 can reference memory [5][7]. No instruction can reference more than one data address. Since code must be added after each referencing instruction the ix86 instrumented code will be longer and slower.

- Indeterminate Reference Instructions — Some instructions produce an indeterminate number of references. One example is the i486 rep instruction prefix which can cause one string instruction to repeatedly access memory until a condition is true, the number of iterations can only be determined at runtime. To extract the memory references, the instruction must be emulated by a loop constructed during instrumentation.

- Fixed Instruction Length — The combination of data in the text section and variable instruction length can make it impossible to correctly disassemble the text section, because disassembly can become unaligned with correct instruction borders. Common sources of data in the text section are constant data and jump tables. In contrast to the R3000, the i486 has variable instruction lengths.

1. Recall that R3000 executables can be instrumented with Pixie.

- Addressing Modes — There are seven different modes in the i486 while the R3000 has only two. More addressing modes as well as instruction types add to the complexity of the instrumentation tool both to recognize a data referencing instruction and to output the proper instrumentation code.

- Indirect Addressing — Most architectures support this but use it only occasionally to index into a jump table or for an indirect call. Such control instructions are hard to relocation and usually add additional runtime overhead to resolve. The R3000 uses data indirection to perform a procedure return and so this overhead can be substantial. This is discussed further below.

- Register Set — The i486 has 8 general purpose registers while the R3000 and many RISC-style processors have 32. Instrumentation code needs several global variables to hold such things as buffer pointers. RISC instrumentation tools store these variable in little used registers while IDtrace must save register values, load the variables into registers, and then restore the register's original contents for every added code sequence. This causes a significant code size and execution speed penalty for IDtrace. In addition, the ix86 ISA includes segment registers which add to the complexity of effective address calculations. Fortunately, these registers are not used on most Unix machines.

Along with ISA characteristics, compiler constructs can also cause problems for late code modification during code relocation. Two examples are jump tables and indirect calls.

- Jump Tables — A jump table is a list of absolute addresses indexed by an indirect jump that is usually generated by a long switch or case statement. They can be in either the text or data section and the instrumentation process must find and update them since they are lists of absolute addresses which no longer point to their proper locations within the instrumented text. IDtrace analyzes the compiler generated preamble instructions before an indirect jump instruction to get the location and size of a jump table. Once a table is found each absolute address is relocated and the indirect jump will execute as intended. IDtrace is dependent upon the compiled code to have recognizable preamble instructions and thus may not correctly instrument all binaries.

- Indirect Calls — The target of most procedure calls is given by a static relative offset. Sometimes, however, an indirect call is used and the target is computed at runtime. The instruction[1]

```
call *1044(%eax,4)
```

  is an example of an indirect call. The target is computed from register values thus cannot be relocated at instrumentation time. In other words, if the target were originally $1000 and after code modification it has moved to $1200, it is difficult to change the instruction operands so that the new target of $1200 will be computed.

  The solution is to build an address translation table which will be resident when the new code is run. The table associates the original beginning addresses of all procedures with their corresponding new addresses. All indirect call instructions are replaced by code which computes the original call target and passes the target to a call-handling utility routine. This routine does a table lookup using the original target to find the new target and then transfers control to that address. Upon return, control passes back to the instruction after the indirect call. This method requires runtime overhead for every indirect call and additional memory allocation to store the table. Furthermore, hand coded (as opposed to compiler generated) routines may contain indirect calls to targets which are not recognized as function beginnings. Since an indirect call is the procedure return method on the R3000, this overhead is substantial in code instrumented by Pixie. In contrast, Nixie, the more efficient but compiler dependent version of Pixie, removes the return overhead by assuming register r31 always

1. This is i486 assembler indirect (*), indexed addressing using register `%eax` multiplied by 4 with a displacement of 1044.

holds the return address, which it normally does. The relocated return address is put in r31 during the call so that an unmodified indirect call can be used to return to the correct location.

5.0 Instrumentation Limitations

Since IDtrace must perform complete code relocation of a binary with no relocation information it is possible that IDtrace will be unable to correctly instrument all programs. Outlined below are the known limitations of IDtrace.

- Compiler dependency - To handle relocation IDtrace must be somewhat compiler dependent. Currently IDtrace recognizes code generated by the Intel/AT&T compiler and the Intel Proton compiler version Jul92. The areas of dependence and how they can be manually overcome are discussed below in Section 5.1.

- Data in text segment - Some compilers in-line read-only data such as constants or jump tables in the code segment. The data will be disassembled as instructions which in itself is not a problem but, because of non-uniform instruction length, it is highly likely that disassembly will not be aligned with the beginning of real instructions after the in-lined data. Thus spurious instructions will be produced ensuring that the resulting code will not run. The above compilers put such data in a read-only data section.

- Unusual indirect calls - All indirect calls must point to the recognizable beginning of a function. Currently a function beginning is recognized by a preceding ret, nop, or jmp instruction. The supported compilers conform to this rule but others may not. An indirect call to an unrecognized address will cause an error message and execution termination.

- Unexpected code termination - Unspecified results occur if the test code terminates without calling the C library exit routine. Most likely the .cnt file will be empty and the .trc file will be empty or incomplete. Again this will not be a problem with code generated by the above mentioned compilers.

- Fork system calls - A fork call will execute properly but the child process will not be instrumented. It is possible for the child process to be separately instrumented and the source code of the parent process to be modified to call the instrumented version of the child process. However, two separate trace files would then exist and the method of merging the two files is unclear. Furthermore, if the child process is called multiple times, only the trace of the last call will be saved.

As a final note, not all instructions are currently implemented primarily because they have yet to appear in any tested application code. The following instructions are unimplemented: `enter`, `frstor`, `fsave`, `fnsave`, `lgdt`, `lidt`, `lldt`, `sgdt`, `sidt`, `sldt`, `verr`, and `verw`. Test code which includes any of these instructions should still run correctly but the resulting trace data may be somewhat incorrect.

5.1 Compiler Dependencies

IDtrace relies on specific compiler conventions in many places. If the binary contains code which does not conform to these conventions it is likely that the instrumented version will not run. Some actions mentioned above such as data inlining and strange indirect calls cannot be overcome. However, other dependencies listed below can be overcome by manually examining the code and including extra information in the hint file:

- _start in crt0.o - IDtrace finds the addresses of main and exit by looking in predefined locations in _start for calls to these functions. Different positions for these calls can be given by including main_call or exit_call entries in the hint file. As a side note, Idtrace differentiates the Intel/AT&T and Proton compiled code by the beginning address of _start. This will be changed in the future.

- Start and stop instrumenting addresses - IDtrace will start instrumenting code at the predefined address directly after _start and computes the stop instrumentation address by looking for the .fini section. Start and stop addresses can be given directly by including trace_beg and trace_end entries in the hint file.

- _curbrk location - _curbrk is a C library variable which contains the address of the end of the data segment. It is modified during calls to sbrk which is called by malloc. The initial value of _curbrk is the correct value for the original binary but it must be increased to reflect the larger size of the instrumented binary. Idtrace must know the location of _curbrk to make this change and it finds it by pattern matching disassembled instructions with known sbrk instructions. If IDtrace cannot find _curbrk a warning message is produced. It is not always an error because _curbrk is not included in all programs. A curbrk_addr entry can be added to the hint file if IDtrace cannot find _curbrk but the user can.

- jump tables - Jump tables are produced by long switch statements. The usual code structure is a few preamble instructions checking the length of the jump table and then an indirect jump to the jump table located in the data segment. IDtrace must know the position and size of a jump table so that it can update the original table entries with their new relocated values. If the predefined preamble instructions are not what IDtrace expects, the table size cannot be ascertained and the jump table will not be updated. An indirect jump without the expected preamble instructions will cause a warning to be issued at which point the user can decide the jump table address and size and include a jtab entry in the hint file.

5.2 IDtrace Hint File (-H option)

The hint file can be created by the user to assist idt in instrumenting the binary file. It is an ASCII file and is composed of any number of the following lines:

Line Tag	Description
TRACE_BEG_TAG	Address at which to begin instrumentation
TRACE_END_TAG	Address at which to stop instrumentation
MAIN_CALL_TAG	Address of main call in start()
EXIT_CALL_TAG	Address of exit call in start()
JTAB_TAG	Table Beginning Address Table Size
CUR_BRK_TAG	Address of _curbkr

The values can be given in decimal or hexadecimal. The tag values can be found in idtdef.h.

6.0 Experiments with IDtrace

Idtrace can produce a variety of traces including profile, memory references, and full execution traces. In this section we illustrate some of its capabilities. 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, see Table 1. IDtrace can instrument a binary in roughly three times the time required

to compile the binary and it is about twelve times larger than the original. 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. For comparison, Pixie running on a DECstation 5000 has a filesize expansion of about four times and execution slowdown of about two times with no memory reference trace and ten times with a reference trace. The following subsections illustrate instruction mix and basic block statistics using a profile trace, a branch prediction study using the branch traces generated by IDtrace, and finally, cache studies using memory reference and full execution traces.

Program	Description	Type	i486 Instructions
cc1	Major forked process of GNU C compiler	Integer	65 million
compress	Unix compression utility	Integer	60 million
eqntott	Boolean equation to truth table translator	Integer	1650 million
espresso	Logic minimization tool	Integer	531 million
sc	spreadsheet program	Integer	980 million
xlisp	XLISP interpreter solving 8 queens problem	Integer	990 million
alvinn	Neural network training with backpropagation	FP	5042 million
ear	Inner ear model	FP	458 million

TABLE 1. The SPEC92 C benchmarks used in our experiments.

6.1 Instruction Mix

Table 2 illustrates instruction mix measurements using six integer benchmarks. Notice that the top 3 instruction types comprise 60% of the total instructions executed and the top 14 types make up 90%.

Opcode	% of Total	Opcode	% of Total
mov	28.3	pop	2.9
jcc	17.4	xor	2.9
cmp	14.4	and	2.2
push	4.9	add	2.0
movswl	4.1	jmp	1.7
inc	3.8	dec	1.2
test	3.5	call	1.1

TABLE 2. Instruction mix of integer benchmarks: cc1, compress, eqntott, espresso, sc, and xlisp. Total of 4.3 billion instructions.

Frequency measures of this type can be used to guide implementation decisions such as whether an instruction should be designed to execute in one cycle or emulated in microcode in multiple cycles.

6.2 Code Profiling

The spatial locality of executed code can be measured by using the basic block execution counts provided in the profile trace. The execution frequency of a block is multiplied by the number of instructions in the block to get the approximate time spent in each block. The largest times are then summed until

they equaled 90% of the total execution time. The results in Table 3 show that most of the execution

Program	Different Executed Basic Blocks	Number Generating 90% of Total Ex. Time	Percent
cc1	10216	1482	14.5%
xlisp	1204	119	9.9%
espresso	3752	343	9.1%
sc	3047	223	7.3%
compress	421	18	4.3%
eqntott	749	12	1.6%
ear	906	12	1.3%
alvinn	1065	9	0.8%

TABLE 3. Code locality - The percentage of the different executed basic blocks which contribute 90% of the total execution time.

time is spent in a small portion of the executed code, as one would expect. Although the first four roughly follow the 90/10 locality rule, floating point and highly iterative programs show even greater localization.

6.3 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 last bits of a branch instruction's address is used to index into the table. If the value of the counter in the table is non-negative the branch is predicted taken. Taken branches increment the counter up to at most a value of 1 and non-taken branches decrement the counter to at most -1. The simulator outputs the location of the branch and whether or not it was predicted correctly. A postprocessor counts the number of unique, executed conditional branches and the number of mispredictions for each branch. Table 4 shows that a few unique branches are executed many times. The last column gives the number and percentage of unique conditional branches which generate 90% of the total conditional branches executed. More importantly, the last column in Table 5 gives the number and percentage of unique branches which cause 90% of the mispredictions. It shows that an even smaller percentage of conditional branches (usually less than 5%) cause 90% of the 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 2-bit counter cannot contain enough information to predict.

Benchmark	Total Branches Executed	Unique Branches	Number Causing 90% of Total Branch Executions
eqntott	345763211	333	5 (2%)
ear	37291744	402	8 (2%)
compress	11749280	217	13 (6%)
sc	193868064	1427	104 (7%)

TABLE 4. Conditional Branch Distribution

Benchmark	Total Branches Executed	Unique Branches	Number Causing 90% of Total Branch Executions
xlisp	157075812	425	49 (12%)
espresso	98055126	1434	188 (13%)

TABLE 4. Conditional Branch Distribution

Benchmark	Unique Branches	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 5. Mispredicted Branch Distribution

6.4 Cache Simulations

Our first cache study was to investigate the properties of the i486 on-chip cache. To do this timing and memory-system simulators where built to model i486 behavior. The memory simulator models the cache, write buffers, and instruction prefetcher. The timing simulator takes an execution trace and derives the execution time for each instruction. It calls the memory system simulator to determine memory stalls due to cache misses or full write buffers. Floating point instructions are not accurately modeled at present so the experiments were only run on the integer benchmarks. While our results should not be used to derive any absolute performance figures such as CPI or effective memory access time since actual times will be slower due OS effects, it is possible to use these results to measure relative i486 performance with respect to different cache configurations. Furthermore, it sheds light on the i486 designer's cache parameter choices. In an effort to make the cache results more representative of multiprogramming operation, the cache is flushed every 50,000 instructions.

The i486 on-chip cache is an 8K byte, 4-way set associative, unified write through cache with a line size of 16 bytes. In addition, 4 double word[1] (dword) write buffers avoid lock-up on writes. The fastest i486 memory model was used: 2 cycles to read the first 4 byte dword, 1 cycle for subsequent burst dword reads, and 2 cycles to write a dword. Thus a cache line refill takes 5 cycles. If its buffer is not full, the prefetcher tries to prefetch the next line past the current execution point. The line can be copied from the cache or from memory if the memory bus is free from pending writes and cache misses caused by data reads. If the line is copied from memory it is termed a memory prefetch. It can potentially increase performance by eliminating an instruction cache miss. Both the cache and prefetch buffer are updated on a memory prefetch. For more complete timing and memory system information see [5] and [6].

Table 6 shows the cycles required to run 10 million instructions of each benchmark for different cache configurations which bracket the on-chip cache found on the i486 (shown in bold in the table). Table 7 shows the combined statistics for all six benchmarks. Unfortunately, the relatively small number of

1. We use Intel's convention of 16 bit words.

misses in the SPEC benchmarks produces only a small spread in the cycle counts but for these user pro-

	Cache Config.	cc1	compress	eqntott	espresso	sc	xlisp
Vary Line Size	8B, 4-way, 4wb	21.3	19.0	18.3	17.4	20.8	18.3
	16B, 4-way, 4wb	**21.2**	**18.8**	**18.1**	**17.2**	**19.9**	**18.1**
	32B, 4-way, 4wb	21.7	18.8	18.0	17.2	19.8	18.1
	64B, 4-way, 4wb	24.6	19.1	18.0	17.5	21.1	18.6
Vary Asso-ciativity	16B, 8-way, 4wb	21.0	18.8	18.0	17.0	19.7	17.7
	16B, 4-way, 4wb	**21.2**	**18.8**	**18.1**	**17.2**	**19.9**	**18.1**
	16B, 2-way, 4wb	21.6	19.0	18.1	17.3	20.2	18.3
	16B, 1-way, 4wb	23.5	19.2	18.5	17.7	21.0	19.3
Vary Write Buffers	**16B, 4-way, 4wb**	**21.2**	**18.8**	**18.1**	**17.2**	**19.9**	**18.1**
	16B, 4-way, 2wb	21.4	18.8	18.1	17.2	20.3	18.1

TABLE 6. Cycles (in millions) to execute 10 million instructions of each benchmark with cache configurations bracketing that onboard the i486. The cache was flushed every 50 instructions to approximate context switches. 8B, 4-way, 4wb denotes line size, associativity, and the number of write buffers. The other parameters, 8K total, unified, and a write-through policy are fixed.

grams, the i486 cache configuration is near optimal. The 8-way associativity is beneficial but probably not worth the cost or potential increase in critical path. In fact, 2-way associativity might have sufficed. The bus traffic can be viewed by comparing the prefetches and write buffers full columns. Fewer prefetches mean fewer free bus cycles. Similarly, an increase in number of times the write buffers are full means there were fewer cycles available to write the buffers. This is most evident in caches with long cache lines and many misses both of which tie up the memory bus. Finally, it is interesting to note that an increase in the number of prefetches doesn't translate into a reduction of instruction misses. In fact, the number of prefetches which actually prefetch a line from memory into both the cache and the prefetch buffer (prefetch misses) is fairly constant. This implies that little is gained by prefetching user

code. Branches in the code still cause instruction misses and sequential instruction lines are most often

	Cache Config.	Cycles (M)	Read Misses (K)	Instr. Misses (K)	Prefetch (K)	Memory Prefetch (K)	Write Buffers Full
Vary Line Size	8B, 4-way, 4wb	115	1719	2069	7710	12	2
	16B, 4-way, 4wb	**113**	**1220**	**1535**	**2836**	**15**	**27**
	32B, 4-way, 4wb	114	979	1155	1391	15	102
	64B, 4-way, 4wb	119	1045	1153	281	13	270
Vary Associa-tivity	16B, 8-way, 4wb	112	953	1361	3206	12	39
	16B, 4-way, 4wb	**113**	**1220**	**1535**	**2836**	**15**	**27**
	16B, 2-way, 4wb	115	1453	1535	2719	31	71
	16B, 1-way, 4wb	119	2341	2451	2162	46	122
Vary Write Buffers	**16B, 4-way, 4wb**	**113**	**1220**	**1535**	**2836**	**15**	**27**
	16B, 4-way, 2wb	114	1224	1545	2849	15	112

TABLE 7. Combined cache performance statistics of the six integer benchmarks: cc1, compress, eqntott, espresso, sc, and xlisp.

already in the cache.

The availability of IDtrace also makes it possible to compare cache performance of two very different approaches to instruction set design typified by the i486 and the R3000. The i486 to maintain compatibility with earlier members of the ix86 family has variable length instructions and few CPU registers (8 vs. 32) resulting in higher code density but more memory data references. We found these differences to be less significant than one would expect, because the i486 stack acts as an extended register file that usually resides in the cache.

The R3000 traces are generated by Pixie on a DECstation 5000 and the cache simulator used is a modified version of the multicache simulation tool, Tynero [14]. Table 8 shows some preliminary data in

	Misses in thousands (% Ratio)			
	i486: 290M references		R3000: 134M references	
Line Size (bytes)	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 8. 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-way	1.0%	4.3%	0.5%	2.4%
8K 4-way	0.5%	3.7%	0.2%	2.1%
16K 2-way	0.4%	2.8%	0.2%	1.1%
16K 4-way	0.3%	2.4%	0.1%	1.2%

TABLE 9. Miss ratios for push, pop, and non-push/pop references running espresso on the i486.

which 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 of the stack. Once the top of the stack is resident in the cache few of these references will be misses. 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 9 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.

7.0 Conclusion

Instruction traces can be gathered by both hardware and software methods. Hardware monitoring gives complete system level information but its use is limited by the required equipment. Software methods can provide much of same information yet are much more accessible. Software instrumentation tools provide an easy way to gather traces and have been build for many architectures. IDtrace is the first executable-level instrumentation tool for ix86 architectures running Unix SysV. Its advantages are its ease of use, low cost, ability to instrument library code, freedom from source code, and fast trace gathering. The tool is currently being used to gather traces for cache performance, branch prediction, and instruction pipeline studies.

Appendix A Trace and File Formats

This section describes the format of the information contained in the various files created by IDtrace. The actual tag values given for the tag names are not guaranteed to be the current values. It is suggested that names be used from the header file addcode.h. The C structures below can be found in idtdef.h.

A.1 Trace Formats

IDtrace will produce four different kinds of traces for different applications depending upon the options given.

Profile Trace

Records only the dynamic execution count of each basic block in the .cnt file. This file in conjunction with the .blk file (and maybe the original binary) can be used to produce profile and runtime statistics. An empty .trc file is also produced.

Branch Trace

This trace is designed for branch prediction simulation. The .trc file is a stream of two entry types. One is a 5 byte entry specifying a basic block beginning. The first byte is a basic block tag, BBADTAG (0x70), and the other 4 bytes contain the basic block beginning address. The other type of entry is either 6 or 7 bytes long. It specifies a control instruction ending the basic block. The first byte of the entry is the tag, either OP1TAG or OP2TAG (0xa0 or 0xb0 respectively) for 1 or 2 byte opcodes. The next 1 or 2 bytes is the control instruction opcode and the last 4 bytes contain the address of the last byte of the control instruction. Although the last field may seem unintuitive, branch target buffers are often indexed by this number. As an example its use, suppose a control instruction entry for a conditional jump contains X in the address field and the immediately following basic block entry contains the address Y. If X + 1 = Y then the jump was not taken and execution continued with the instruction (basic block) succeeding the conditional jump.

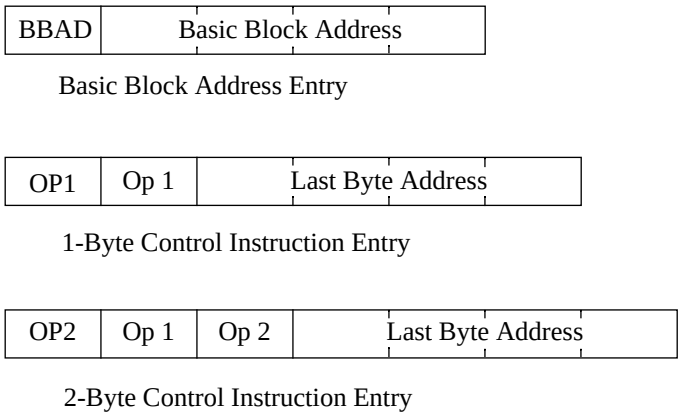

FIGURE 3. Branch Trace Entries

Cache Trace

The trace file is for use as input to a cache simulator. If the -r option is not given the trace is a stream of two entry types: cache line accesses for instructions and effective addresses for data references. Each entry is 5 bytes long. The first byte of the data reference entry is a tag containing information about the direction of the access (read or write) and the size of the reference (byte, word, lword, etc.) The tag is generated by adding either RDTAG (0x10) for a memory read or WRTAG (0x20) for a memory write reference with one of the size values from the table below.

Type	Value	Type	Value
byte	0x01	64 bits	0x04
word	0x02	80 bits	0x05
lword	0x03		

For example, 0x21 is a byte write and 0x13 is a lword read. Unaligned and aligned references are treated identically. It is assumed the cache simulator will specially handle unaligned references.

The other entry type of the cache trace is an instruction cache access. Since is wasteful in both space and time to output an entry for each executed instruction, one entry is output to the trace file for each cache line access instead. If two instructions in a basic block are located in the same cache line only one entry will be output. The default cache line size is currently16 bytes and can be changed using the -l option. So with the default value caches with lines size of 16 bytes or greater can be modeled. Increasing the trace line size will reduce the number of entries to be processed and thus will slightly reduce simulation time. The format for a cache access entry is a 1 byte tag of value CLTAG (0x60) followed by the 4 byte address of the beginning of the cache line.

Upon entering a basic block, the instruction cache line entries will be generated first followed by all data references. Thus the instruction and data references are not interleaved within a basic block but are in a coarser sense throughout the whole program. This will have no effect when modeling split caches and a negligible effect on a unified cache model.

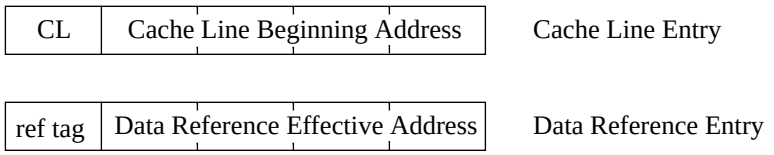

FIGURE 4. Cache Trace Entries

If the -r option is given repeat-end entries will be output and different tags will be used for data references generated by instructions with rep prefixes. An instruction with a rep prefix will generate at least one data reference entry for each iteration. The data reference format will be the same as above except that RRDTAG (0x30) is used for the repeat read tag and RWRTAG (0x40) for the repeat write tag. A repeat-end entry is output at the end of the data reference entries for a rep instruction. The first byte of this entry type is a tag with value RENDTAG (0x50) and the other 4 bytes are zero. The use of different tags and the repeat-end allow a postprocessor to analyze the locations and sizes of block references.

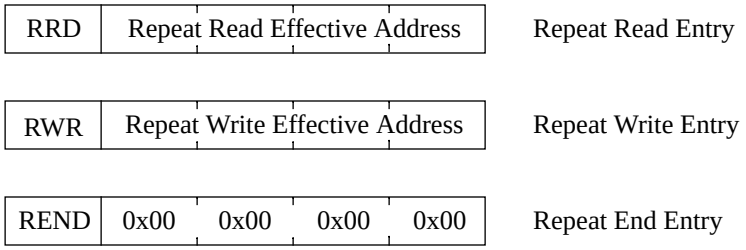

FIGURE 5. Repeat Entries

Execution Trace

By comparing the original binary code with the execution trace it is possible to recreate the exact behavior of the program. The information and format of this trace is geared toward creating a kernel trace (KT format) for processor and pipeline simulators. A program exists called idt2kt which converts IDtraces's execution trace into the KT format. Information on this tool can be found in Section 3.3 on page 8. There are four types of entries: data references, repeat ends, basic block file positions, and targets. All of them begin with a one byte tag and then have a 4 byte information field, look at Figure 6 on page 21. Unlike in the cache trace the data reference entries produced for an instruction do not always correspond to the memory references performed by the instruction. Exceptions are listed below. Instructions with a

rep prefix generate multiple data reference entries using the RRDTAG and RWRTAG tag values as does cache tracing with the -r option. Basic block beginnings generate a basic block entry with tag value BBFPTAG (0x80). The four byte information field contains the location of the block record in the .blk file. A target entry with tag value TARGTAG (0x90) is generated by control instructions such as call, lcall, ret, and jumps. The information field of the target entry contains the target address of the control instruction. There are a few other details:

- Call instructions first produce a target entry and then a data reference entry,
- Lcall instructions generated a target entry and then one data reference entry containing the first stack push address (%esp-4),
- In the case of indirect jumps or calls the target entry contains the indirection pointer, i.e., not the actual target but the location at which the target is found,
- Pusha, pushal, popa, and popad generated only data reference entry - the stack location of the first push (%esp-4) or pop (%esp),
- Instructions such as inc which read and write to the same effective address generate only one data reference entry with a tag value of RWTAG (0x00). The cache trace would produce separate read and write entries.

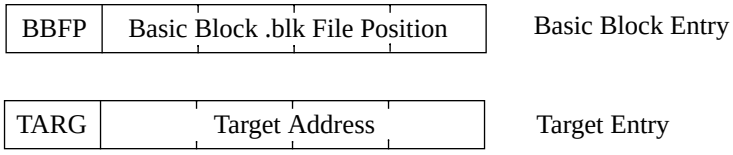

FIGURE 6. Execution Trace Entry Formats

A.2 File Formats

.idt File

ELF executable

.blk File

This file hold information pertaining to each basic block of the original binary. The first bytes are the header structure:

```
#define BLK_FHDR struct blkhdrstruc
struct blkhdrstruc {
   long blk_cnt;
   long oldt_offset;
   magic_t orig_magic_num;
   magic_t idt_magic_num;
};
```

Blk_cnt is the number of basic blocks and oldt_offset is the difference between the address of an instruction and its file position in the original binary. Its use is shown below. The rest of the file is a list of block structures - one for each basic block. Each block structure has the form of BLOCK-STRUC below.

```
#define I_INFO struct i_info
struct i_info {
   short idt_id;
   char instr_size;
   char entries;
};

#define BLOCKSTRUC struct blockstruc
struct blockstruc {
   ADDR address;
   char size;
   I_INFO instr_list[MAX_INSTR_PER_BLOCK];
};
```

The field address refers to the beginning of the basic block and size is the number of instructions in the basic block. The file position of the basic block can be found by subtracting oldt_offset from address. Instr_list is a list of 4-byte I_INFO structures, one for each instruction in the order in which they occur in the basic block. The first two bytes contain a unique identifier for the instruction which enables instruction recognition without opcode disassembly and the next byte, instr_size, is the number of instruction opcode bytes. The last byte, entries, is the number of trace file entries generated by the instruction. In the cache trace this would be the number of data references, in the execution trace it would also include target entries. An exception to the definition of entries is for instructions with a rep prefix since the number of data references produced is unknown prior to execution. A rep prefixed instruction which has one data reference per iteration (scas, lods, and stos) has the value -1 for entries while rep prefixed instructions with two references per iteration (movs and cmps) has a value of -2. So to associate a rep prefixed instruction with its data references, entries in the trace file are read in groups of the absolute value of the entries field until a REPEND entry is read thus signalling the end of the rep instruction.

.cnt File

This file holds the dynamic count of the number of times each basic block is executed. The first bytes are the header structure:

```
#define CNT_FHDR struct cnthdrstruc
struct cnthdrstruc {
   magic_t orig_magic_num;
   magic_t idt_magic_num;
};
```

The rest of the file is a stream of long (4-byte) integers. The number of integers will be a multiple of the block count. For smaller traces there will be exactly one number for one basic block. The first number after the header is the execution count of the first basic block, the second number is the count of the second basic block, etc. Multiple sets of counts are saved if one block count becomes too large ($> 2^{32}$-1). For example, if there are 1000 basic blocks, there could be 1000, 2000, 3000,

etc. integers in the file after the header. The sum of the 1st, 1001st, 2001st, 3001st, ... integers is the total execution count for the first block.

.trc File

The trace file does not have any header information and its structure depends upon the trace option given to idt. The trace file will be empty for a profile trace.

Appendix B KT Format

There are three trace driver data records. PID records contain information about the process for which the following instructions and memory references pertain. MMU records contain information regarding the virtual memory to physical memory mapping for the current address space of the process. Instruction trace records contain information regarding the effective address and contents of the current instruction. IDtrace will only output instruction records. The others are described for completeness.

The internal format for each of the records follows. Note that the ‘type’ of each record is defined by byte 4 of the record. The following shows the meanings of type field values:

Type	Values
Instruction	0x00 - 3f
Reserved	0x40 - 7f
PID	0x80
MMU	0x81
Reserved	0x82 - ff

PID Records

The format:

Mnemonic	Size	Description
pid	4 bytes	Current Process ID
type	1 byte	TR_PID from sys/tr.h
PAD	3 bytes	Pad bytes for record alignment

MMU Records

The format:

Mnemonic	Size	Description
pfn	4 bytes	Page Frame Number
type	1 byte	TR_MAP from sys/tr.h
va	3 bytes	Virtual Address of page

The four bytes containing ‘type’ and ‘va’ above are considered as a four byte integer representing the virtual address of the page plus the value of TR_MAP.

It may be necessary to keep up to TR_TLBR MMU records in the buffer in order to successfully decode the physical addresses involved in an instruction.

Instruction Records

The format:

Mnemonic	Size	Description
eip	4 bytes	Instruction Pointer
type	1 byte	00eennnn
inst	N bytes	Instruction opcode bytes
PAD	(-n-1) % 4 bytes	Pad bytes for record alignment
ea	E*4 bytes	E effective addresses (virtual)

An instruction can have 0, 1 or 2 effective address entries. N = nnnn is the number of bytes of instruction opcode bytes and E = ee is the number of 4 byte ef fective addresses. Some instructions put values other than memory effective addresses in the ea field.

- Jump and conditional jump instructions have the target address in ea field.
- Indirect jumps have the effective address of location of the indirection pointer in the one ea entry.
- Return and leave instructions have the stack pointer and base pointer respectively in the ea field since they are memory references of those instructions.
- Instructions with the rep prefix generate a separate entry for each iteration of the instruction. For instance, a rep movsb instruction which executes five times would generate five instruction records. The eip, type, inst, and PAD fields would all be identical. The ea field of the first record would contain %esi and %edi (in that order), the ea field of the next record would have %esi+1 and %edi+1, and so on until the ea field of the last record contains %esi+4 and %edi+4.
- The instruction type not the actual number of memory references performed by the instruction determines the number of effective addresses in the ea field. In other words one ea entry may result in more than one memory reference. For instance, instructions such as inc which perform two references to the same address have a single ea entry.

  Below is a table of all the cases of two ea entries:

Name	Opcode	Ea1 Value	Ea2 Value
Call Ev	ff /2	stack	ind. addr
Push Ev	ff /6	stack	target
Call Jv	e8	target	stack
movs/cmps	a4 - a7	%esi	%edi
Call Ap	9a	target	stack
Call Ep	ff /3	target	stack

References

[1] A. Borg, R. Kessler, and D. Wall, "Generation and Analysis of Very Long Address Traces," *Proc. of 17th Annual International Symposium on Computer Architecture*, Seattle, WA, 1990, pp. 270-281.

[2] S. Eggers, D. Keppel, E. Koldinger, and H. Levy, "Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor," *Proc. of 1990 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, May 1990, pp. 37-47.

[3] K. Flanagan, K. Grimsrud, J. Archibald, B. Nelson, "BACH: BYU Address Collection Hardware," Technical Report TR-A150-92.1, Dept. Of Electrical and Computer Engineering, Brigham Young University, Provo, UT, Jan. 1992.

[4] M. Holliday, "Techniques for Cache and Memory Simulation Using Address Reference Traces," International Journal in Computer Simulation, Vol. 1, pp. 129-151, 1991.

[5] Intel Corp., *i486 Microprocessor Programmer's Reference Manual*, 1990.

[6] Intel Corp., *i486 Microprocessor Hardware Reference Manual*, 1990.

[7] Kane, Gerry, *MIPS R2000 RISC Architecture*, Prentice Hall, Englewood, Cliffs, NJ, 1987.

[8] J. Larus, "Abstract Execution: A Technique for Efficiently Tracing Programs," *Software - Practice and Experience*, Vol. 20, pp. 1241-1258, December 1990.

[9] MIPS Computer Systems, Inc., *Language Programmer's Guide*, 1986.

[10] D. Nagle, R. Uhlig, T. Stanley, T. Mudge, S. Sechrest and R. Brown, "Design Tradeoffs for Software-Managed TLBs," *Proc. of the 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 27-38.

[11] O.A. Olukotun, T.N. Mudge, and R.B. Brown, "Implementing a cache for a high-performance GaAs microprocessor," *Proc. of the 18th Annual International Symposium on Computer Architecture*, May 1991, pp. 138–147

[12] J. Patel, "How to Simulate 100 Billion Address References Cheaply?," *ISCA '90 Workshop on Processor Tracing Methodologies*, Seattle, WA, May 1990.

[13] J. Pierce, "IDtrace: A Trace Generation Tool for the ix86 Instruction Set," Technical report, Intel Corp., Hillsboro, OR, Sept. 1992.

[14] J. Quinlan, and K. Lai, "Tynero: A Multiple Cache Simulator," Technical Report, Intel Corp., Hillsboro, OR, May 1991.

[15] M. Smith, "Tracing with Pixie," Technical Report, Center for Integrated Systems, Stanford University.

[16] C. Stephens, B. Cogswell, J. Heinlein, G. Palmer, and J. Shen, "Instruction Level Profiling and Evaluation of the IBM RS/6000," *Proc. of 18th Annual International Symposium on Computer Architecture*, Toronto, Canada, 1991, pp. 180-189.

[17] Sun Microsystems Laboratories, Inc., "Introduction to SpixTools," Technical Report, Mountain View, CA, April 1992.

[18] D. Wall, "Systems for Late Code Modification," Digital Western Research Laboratory, Research Report, June 1991.

[19] D. Wall, "Link-Time Code Modification," Digital Western Research Laboratory, Research Report 89/17, Sept. 1989.