Instrumentation and Optimization of Win32 Programs Using Etch

The Etch Group
Department of Computer Science and Engineering
University of Washington
etch-info@cs.washington.edu

Abstract

Etch is a program performance evaluation and optimization facility for use with Win32 applications running under Windows NT on Intel x86 platforms. The Etch system can be used effectively by both developers, who wish to understand the performance of their programs during the development cycle, and users, who wish to understand the performance of common applications executing in their environment. Etch also supports modification of executables to optimize their performance. Key to Etch is its use of binary rewriting technology; Etch operates directly on executable binaries and dynamically-linked libraries, so it requires neither source code, nor symbol tables, nor special linking procedures. This paper describes Etch, the services and interfaces that it provides, and our experiences using it for both measurement and optimization.

1 Introduction

During the last decade, the Intel x86 instruction set has become a mainstay of the computing industry. Arguably, Intel processors have executed more instructions than all other computers ever built. Despite the widespread use of Intel processors and applications, however, there are remarkably few tools available to assist the programmer and user in understanding or exploiting the interaction between applications, the processor, and the memory system on x86-based platforms. As overall performance becomes increasingly tied to memory system performance, such tools will be increasingly important in understanding and fixing program performance bottlenecks.

This paper describes Etch, a binary rewriting system for applications that run on Intel x86 processors. We developed Etch to aid programmers, users, and researchers in understanding the behavior of arbitrary applications running on Intel architectures. Because Etch supports general transformations on executables, binaries can also be rewritten to optimize their performance, *without* access to the original source code.

1.1 Background

In general, measurement instrumentation has been applied at two levels: at the source level (by the compiler) and at the executable binary level. The tools that work directly on executable binaries have the advantage of not requiring source code; as well, they are capable of running on any "shrink-wrapped application" compiled by any compiler.

There are two approaches to instrumenting executables. The first approach relies on emulation of the executable; an emulator explicitly fetches, analyzes, and executes each instruction in turn, possible capturing measurement data or producing traces as it runs. This technique has been used used, for example, in Intel's VTUNE measurement system [1] and by DEC's VAX instruction analysis tool [14]. The advantage of emulation is its simplicity and the fact that the program's control structure is implicitly discovered as the program runs. The disadvantage, however, is that emulation may slow the execution by 2 to 3 orders of magnitude, possibly perturbing the performance of the program to the point of distorting the results being measured. In addition, an execution time explosion of this scale may render a tool unusable, simply because it takes too long to measure any significant application.

The alternative approach relies on static modification of the binary to insert measurement code, so that the program traces or measures itself as it runs [6, 13, 4]; such modification has been used as well for applications such as binary code migration [8] and software fault isolation [12]. This general technique, called "in-line tracing" or "binary rewriting," has low inherent overhead; the overhead depends on the amount of code added "in-line" to perform a particular measurement. The disadvantage of this approach is that the tool must be capable of statically analyzing an arbitrary binary and determining its control flow. While in-line tracing tools have existed for some time for RISC processors (e.g., Pixie [9] for the MIPS, ATOM [10] for the DEC Alpha, or EEL [5] for the Sun SPARC), none have been available for Intel processors, which support a substantially more complex binary representation.*

1.2 The model

Etch is a system framework in which measurement and optimization tools of nearly any type can be integrated, similar to systems such as ATOM [3] and EEL [5]. Etch permits measurement tools to instrument an executable x86 binary so that the application program, when executed, produces the required measurement data as it runs. For example, one tool may instrument the program to produce a trace of all memory reads and writes, while another may instrument the program to record the number of conditional branch instructions that succeed or fail. Furthermore, because Etch allows complex modifications to the executable binary, Etch tools can also rewrite the program in order to improve its performance. For example, an Etch tool may reorder instructions to optimize for the pipeline structure of a particular processor implementation, or it may relocate procedures to improve memory system locality and behavior.

Etch splits the process of instrumenting and tracing an executable into two phases: an *instrumentation phase* and an *analysis phase*. Similarly, each tool is split into two components, an *instrumentation module* and an *analysis module*. To perform a measurement experiment, the user selects the tool that she wishes to use and the executable that she wishes to analyze. During the instrumentation phase, Etch processes the executable binary in order to "discover" the components of the program, e.g., instructions, basic blocks, and procedures. As Etch discovers each component, it calls into the tool's instrumentation module, telling the module what it discovered. At each such callback, the instrumentation module has the opportunity to instruct Etch to examine and possibly modify the executable, e.g., to insert measurement instructions before or after an instruction, basic block, or procedure. These inserted instructions may include calls to procedures in the analysis module, which will be loaded with the executable at run time (the analysis phase). Thus, when the program runs, it will execute the additional inserted code, including calls into the tool analysis routines that record or process crucial measurement information. Finally, when the program completes, the analysis module is given an opportunity to run whatever routines are required to process and display the data collected during execution.

2 Analyzing Intel binaries

A key challenge in the implementation of Etch is the parsing and evaluation of Intel executables without benefit of source code or debugging information. On most RISC processors, all instructions are fixed format and length, which makes them trivial to parse. On Intel's CISC instruction set, however, there are various difficulties. First, x86 instructions have different formats and lengths. As a result, instructions can start on arbitrary boundaries, making it

*While in-line tracing tools have been built in the past for producing address traces of x86 applications (e.g., TRAPEDS [11] and MPtrace [2]), these tools operated on compiler-generated assembly language, rather than on binary executables.

difficult to identify where instructions begin or end. Second, procedure and basic block entry and exit points are not clearly defined. Third, program text can (and often does) contain embedded data; the data must be identified and preserved if the program is to function correctly.

In Etch, we deal with these challenges through a general technique, which we call *conservative code discovery.* Using conservative code discovery, Etch interprets a program text segment byte by byte, using a collection of rules and patterns to identify instruction boundaries and to distinguish code from data. Bytes within the text segment that cannot be precisely identified as either text or data are assumed to be data and are left unchanged. This ensures that the program will continue to run correctly.

Because the process is conservative, it is possible that Etch will not discover all instructions. Since the purpose of instrumentation is to gather information about a running program, failure to discover a code fragment may reduce the quality of the information, but it does not affect the program's correctness. In practice, we've found that the algorithms and heuristics used by Etch are able to correctly identify and classify over 99% of an executable as either code or data.

At a high level, Etch's code discovery operates in the following way. First, Etch reads the executable and parses the entry point and relocation record definitions specified in the file. Starting at these entry points, Etch begins decoding instructions, identifying the instruction types and looking in particular for control transfers such as conditional branches, jumps, and procedure calls. The objective is to build a description of the binary identifying all of the code blocks. At conditional transfer points, Etch examines both paths if possible; whether this is possible depends on whether the branch target can be statically determined. Similarly, for branch tables (e.g., used to implement `switch` or `case` statements), Etch continues scanning for code through all of the table pointers. Eventually, Etch will have identified all of the code that it can conservatively classify, and will consider the remainder of the binary to be data. Note that the distinction is sometimes complex; as one example, the jump tables mentioned above are not instructions, they are data tables embedded in the binary. However, such structures must be analyzed and identified in order to maximize code discovery. In analyzing the binary, Etch uses its knowledge of certain instruction and data sequence metaphors, as well as heuristics, in order to classify the program components it finds and to distinguish code from data.

Dynamically-linked libraries (DLLs) are also processed by Etch. A program loads a DLL in one of two ways: either the DLL is specified explicitly in the program's import table, or the DLL is specified at runtime by a library or program component. In the first case, Etch automatically discovers the DLL by analyzing the program's import table; in the second case, an Etch tool discovers the DLLs as they are loaded. In both cases, though, Etch copies the original DLL, transforms it, and ensures that the new copy will be used when the instrumented application runs.

Etch is intended for Win32 binaries, and as such has certain limitations. It does does not work on programs that are 16-bit binary executables, use self-modifying code, jump into the middle of an instruction, use complex segment register manipulation, or contain unusual hand-coded assembly sequences. Fortunately, modern 32-bit compilers do not generate code with these properties.

3 Using Etch

We have designed Etch so that it is (1) easy for the programmer to produce new instrumentation and analysis tools, and (2) easy for the user to utilize these tools. The simplicity of these tasks is aided by three standard interfaces (APIs) that Etch provides:

1. The *instrumentation interface* supports the construction of measurement tool instrumentation modules, as previously described in Section 1.2. This API defines the calls from Etch to the tool, as it processes a binary, and from the tool to Etch, as it examines and transforms the binary. The instrumentation interface exports the abstractions of program, module, procedure, basic block, and instruction. As Etch discovers each type of program component, it calls a routine provided by the tool writer. The routine then has the option of modifying the binary, e.g., by inserting instructions before and/or after the component in the executable. Figure 1 shows Etch's main loop and the calls made to the instrumentation module before and after each component.

```
InstrumentInit()
InstrumentModule(Before)
For each procedure in the program:
   InstrumentProcedure(Before)
   For each basic block in the Procedure:
      InstrumentBasicBlock(Before)
      For each instruction in the Basic Block:
         InstrumentInstruction(Before)
         InstrumentInstruction(After)
      InstrumentBasicBlock(After)
   InstrumentProcedure(After)
InstrumentProgram(Before)
InstrumentProgram(After)
InstrumentCleanup()
```

Figure 1: The main loop of Etch.

2. The *runtime interface* defines the calls available to the analysis module during program execution. This interface enables the analysis routines to discover dynamic qualities about the program in execution, such as the effective address of a memory-oriented instruction.

3. The *graphical user interface*, called Visual Etch, allows users to conduct experiments on programs. Visual Etch provides a window- and menu-based system through which users can control experiments, e.g., specifying the application to be instrumented and the instrumentation tool to be applied to that application. Visual Etch then takes care of locating and analyzing the executable, discovering any DLLs that it may use, applying the instrumentation routine against the binary and any associated DLLs, and displaying the results. The GUI provides a set of general mechanisms for collecting, coordinating, and displaying information from a particular experiment, and was designed to make it easy to use and plug in new tools. Figure 2 shows a screenshot of the Visual Etch main control window.

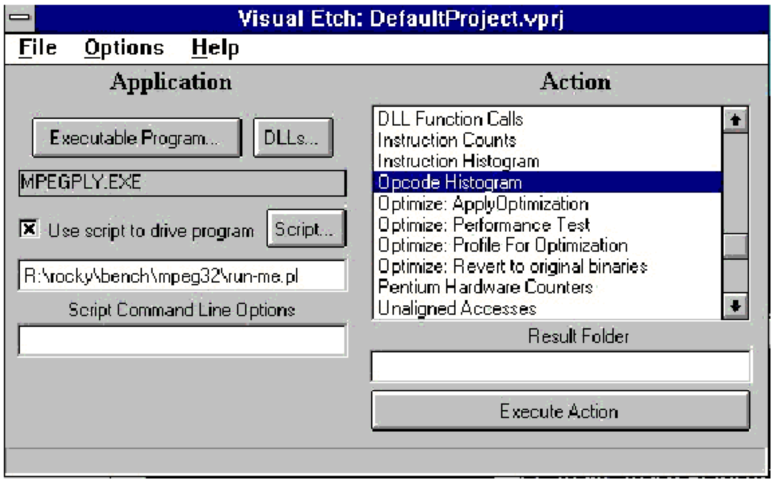

Figure 2: The Visual Etch GUI makes it easy to conduct experiments on Win32 executables.

As an example, Figure 3 and 4 show the code required to build a simple tool: in this case, a measurement tool to count the number of unaligned memory accesses. Figure 3 shows the instrumentation module. The module contains procedures, whose interfaces are defined by the Etch API, through which Etch notifies the instrumentation module of program components. For example, Etch calls the procedure `InstrumentInstruction` both before and after each instruction that it locates in the binary (the parameter `when` indicates whether this is the “Before” call or the “After” call). In the example presented, the instrumentation tool inserts calls to the analysis module routines `LoadReference` and `StoreReference` before the appropriate load and store instructions, respectively. The instrumentation module also inserts a call to the analysis module routine `ProgramAfter` so that it can process the data when the program completes.

When the program is run, the inserted instructions perform calls to the analysis module routines shown in Figure 4. Here we see that the routines `LoadReference` and `StoreReference` simply update counters of unaligned loads and stores. When the program completes, it makes the inserted call to ProgramAfter, which just prints the values of those counters.

```
/*
 * Instrumentation routine for unaligned access detection.
 */

void InstrumentProgram(WhenType when)
{
     if (when == After)                 /* dump results */
        InsertCall("ProgramAfter", 0, 0, 0);
}

void InstrumentInstruction(WhenType when, InstPtr inst, int procNum)
{
    int argc;
    Address_t pc;
    void * array[3];
    ArgType argt[3];

    if (when == Before) {               /* before each instruction */
        argc = 3;
        pc = InstGetPC(inst);
        // 3 arguments get passed to the runtime routines.
        array[0] = 0;     argt[0] = ArgEffAddr;
        array[1] = 0;     argt[1] = ArgEffAddrLen;
        array[2] = (void*)pc; argt[2] = ArgImmed;

        /* for each load or store execute by this instruction
         * call the appropriate routine.
         */
        InsertCallLoadRefs("LoadReference",argc,array,argt);
        InsertCallStoreRefs("StoreReference",argc,array,argt);
    }
}
```

Figure 3: Instrumentation routine for counting unaligned accesses in an executable.

```
/*
 * Analysis routine for unaligned access detection.
 */
int loads = 0;      int unaligned_loads = 0;
int stores = 0;     int unaligned_stores = 0;

void ProgramAfter()
{
    printf("%d loads, %d stores\n", loads, stores);
    printf("%d unaligned loads, %d unaligned stores\n",
                unaligned_loads, unaligned_stores);

void LoadReference(unsigned long address, int len, unsigned long pc)
{
    loads++;
    if (address % len) unaligned_loads++;
}

void StoreReference(unsigned long address, int len, unsigned long pc)
{
    stores++;
    if (address % len) unaligned_stores++;
}
```

Figure 4: Runtime routine for counting unaligned accesses in an executable.

This example shows the simplicity of adding new tools to Etch using the standard interface. Once constructed, a tool like this can be added to the standard tool menu provided by the GUI, and then specified by the user to be applied to any executable. This simple tool (the unaligned access checker) was integrated into the GUI in under 5 minutes, roughly the time required to write the tool in the first place.

4 Rewriting binaries under Win32

In order to rewrite binaries, Etch must perform a number of transformations on the data structures and the instructions in an executable. For example, an Etch tool can insert new instructions between any two existing instructions, remove instructions, or replace instructions; or, it can arbitrarily rearrange the order of basic blocks and procedures inside the text section. To perform these transformations, Etch must be able to manipulate the data structures of an executable, such as the image and section headers, the import and export tables, the relocation tables, and the debug section, if available. When code is inserted, removed, replaced, or moved, Etch must maintain program correctness by updating program control flow. For example, if the target of a jump or call moves, than all jumps or calls to the target must be updated. Likewise, if basic blocks are rearranged, then Etch must ensure that the control flow among the basic blocks remains the same; e.g., if one basic block flows directly into another, and a transformation causes them to be separated, then a jump must be added to the first so that it continues to flow into the second. Finally, Etch must be able to create an entirely new executable once it has finished applying the transformations of the tool being used to the original executable.

5 Some example tools and measurements

Using Etch, we have built a collection of measurement and optimization tools. Among the tools we've built are the following:

- A cache simulator that simulates a memory system with a two-level cache. The simulator consists of separate first-level instruction and data caches (virtually indexed) with a unified second-level write-back cache (virtually or physically indexed). Cache size, associativity, and write policy are configurable. If debugging information is available, the information on misses can be attributed to individual procedures.

- A graphical cache simulator that provides a graphical animation of the above cache simulator. Only the first-level caches are shown; the screen shows two regions representing the instruction cache and data cache. For each cache access, the simulator highlights the row representing the cache line in which the access occurred. A "tick" noise is generated if the access was a cache miss. "Stop" and "Start" buttons can be used to suspend and resume animation.

- An unaligned access counter that monitors a program for expensive unaligned memory accesses; it reports the total number of loads/stores and the number of unaligned loads/stores.

- A trace generator that creates a trace of instruction and data references. We have collected a number of traces of common x86 applications and made them available on the web (see http://www.cs.washington.edu/homes/bershad/Etch/index.html for a link to the traces).

- An instruction profiler that monitors instruction behavior; the output of this tool is fed to an optimizer tool that reorders instructions to improve memory system performance.

- An instruction opcode histogram profiler.

- A call graph profiler that counts the number of cycles in each procedure executed (as with Unix gprof utility). The profiler works on multithreaded applications, and is capable of recording any information on a per-procedure basis that could be supplied using the Intel P6 hardware performance counters.

Figures 5 and 6 show examples of data produced by Etch tools. Figure 5 is the output of the cache simulator that is driven by an Etch trace-producing tool, which logs the addresses of memory access instructions as it runs. The figure shows instruction cache behavior as a function of cache size and organization for the Microsoft Visual C compiler. Using Etch, it is also possible to plug in a cache simulator directly, driving it from address data as it is produced. This technique is used in our graphical cache simulator, which visually shows cache performance in "real time." Figure 6 shows an architectural evaluation, in this case an opcode histogram showing the distribution of opcodes executed during a run of the perl language interpreter. Such measurements are often useful to architects and hardware designers seeking to optimize hardware performance for a class of applications. The opcode histogram tool is simple: at instrumentation time, for every instruction discovered, the instrumentation routine examines the opcode, and then inserts an instruction to call a runtime routine with an integer indicating that opcode. The runtime routine simply updates the specified element of its opcode histogram array, and then prints the array when the program completes.

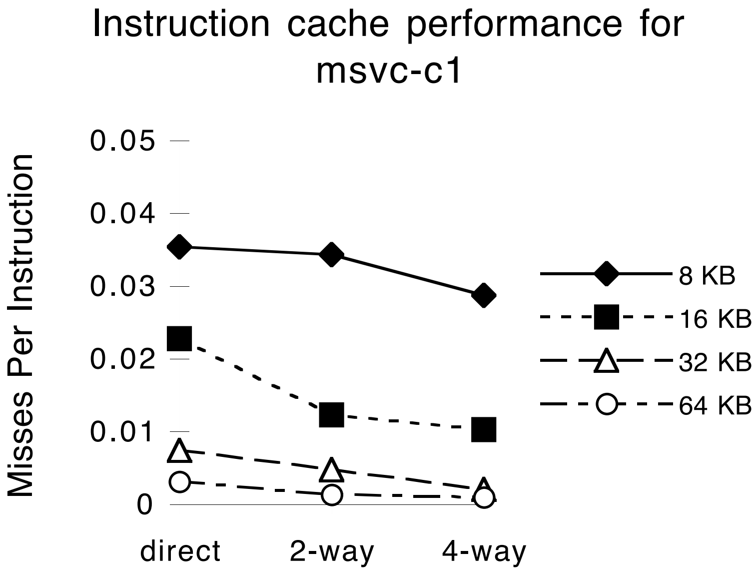

Figure 5: Simulated instruction cache behavior for Microsoft Visual C.

6 Program optimization

In addition to instrumentation, which allows users to observe performance behavior, Etch also makes program optimization straightforward. The optimization process involves four basic steps: instrumentation, data collection, prescription, and transformation. A candidate program is first instrumented to collect data about its execution. Then, the instrumented program is run against a training set, which consists of inputs that exercise the common, and therefore critical paths in the program. During training, data about program performance is

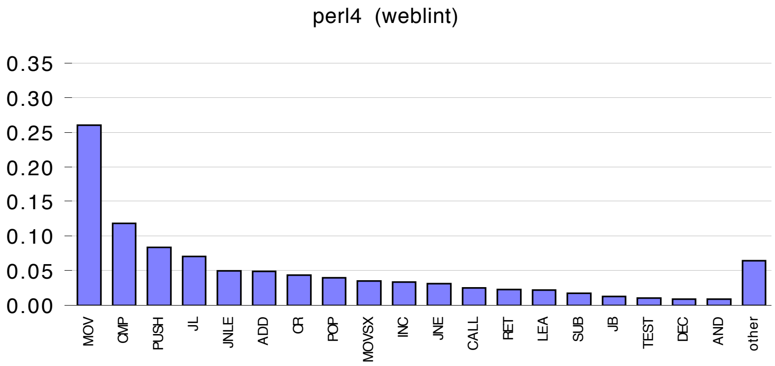

Figure 6: Opcode histogram for execution of the Perl interpreter.

collected by the analysis routines. Once the data has been collected, a prescription routine determines how the program should be modified so that it runs more efficiently. Finally, transformation uses the Etch rewriting engine to modify an executable according to the prescription.

As an example of this process, we have implemented an optimization that improves instruction cache, TLB, and virtual memory behavior using Pettis and Hanson's code layout strategy [7]. Roughly, the strategy improves a program's spatial locality by co-locating temporally related routines. This strategy has been shown to reduce pressure on all levels of the memory management system. The program is run on a training set and an Etch instrumentation tool collects data about the execution; in this case, the tool builds a weighted call graph, indicating which procedures are most frequently called. Transformation code then examines the weighted call graph and builds an optimized layout, where the most frequently called routines are clustered together to produce better locality for the cache and virtual memory systems. Finally, transformation code builds a new executable with the optimized layout using the Etch program binary modification calls provided by the instrumentation API. During the transformation, rarely used code segments within a frequently called routine are lifted and placed at the bottom of the executable image where they have minimal impact on the cache, TLB, and virtual memory system.

The effect of this particular optimization depends on the initial code layout and the dynamic behavior of the application, which may be input dependent. However, for several applications we have examined, this optimization reduced instruction cache misses by as much as 60%, leading to performance improvements of between 3% and 10%, as shown in Figure 7. Note that neither of the Microsoft applications benefit very much from code reordering; we believe that this is due to Microsoft's use of a code reordering tool prior to product delivery.

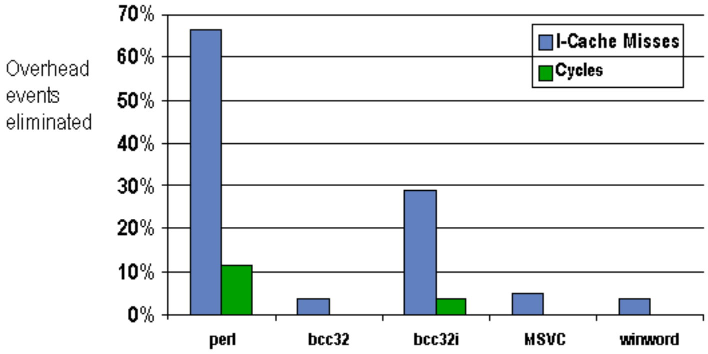

Figure 7: Results of optimization for several applications. Bcc32 and Bcc32i are two versions of the Borland C compiler.

7 Summary

Etch is a program instrumentation and optimization tool for 32-bit x86 applications. It is capable of analyzing and modifying program executables without need for source code or debugging information. Using binary rewriting techniques, Etch can create an instrumented version of an executable that produces measurement data as it runs. Or, Etch can modify an executable to improve its performance, e.g., by modifying the organization of code in order to optimize cache behavior. Key to Etch is its code discovery algorithm which identifies and differentiates code and data in the executable image. Our experience shows that Etch's conservative algorithms and heuristics discover close to 99% of the code for most executables produced by standard compilers.

More information about Etch can be found on the World Wide Web. Please see http://www.cs.washington.edu/homes/bershad/Etch/index.html. In particular, this page leads to a collection of long address traces for several popular Win32 applications.

References

[1] M. Atkins and R. Subramaniam. PC software performance tuning. *IEEE Computer*, 29(8), August 1996.

[2] S. Eggers, D. Keppel, E. Koldinger, and H. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proceedings of the ACM Conf. on Measurement and Modeling of Computer Systems*, May 1990.

[3] A. Eustace and A. Srivastava. ATOM: A flexible interface for building higher performance program analysus tools. In *Proceedings of the USENIX 1995 Technical Conference on Unix and Advanced Computing Systems*, January 1995.

[4] J. Larus. Efficient program tracing. *IEEE Computer*, 26(5), May 1993.

[5] J. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN 95 Conference on Programming Language Design and Implementation*, June 1995.

[6] J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. In *Proceedings of the 3rd ACM ASPLOS Conference*, April 1989.

[7] K. Pettis and R. Hansen. Profile guided code positioning. In *Proceedings of SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990.

[8] R. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *Digital Technical Journal*, 4(4), 1992.

[9] M. D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, November 1991.

[10] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Proceedings of the SIGPLAN 94 Conference on Programming Language Design and Implementation*, June 1994.

[11] C. Stunkel and W. K. Fuchs. TRAPEDS: Producing traces for multicomputers via execution driven simulation. In *Proc. of the ACM Conf. on Measurement and Modeling of Computer Systems*, May 1989.

[12] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, December 1993.

[13] D. W. Wall. Systems for late code modification. Technical Report 92/3, Digital Equipment Corporation Western Research Laboratory, May 1992.

[14] C. Wiecek. A case study of VAX-11 instruction set usage for compiler execution. In *Proceedings of the ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1982.