Instrumentation and Optimization of WIN32/Intel Executables

Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong,
Hank Levy, Brian N. Bershad
University of Washington
and
Brad Chen
Harvard University
Contact: etch-info@cs.washington.edu

Introduction

Etch is an application program performance evaluation and optimization system, developed for Intel x86 platforms running the Windows/NT operating system. The system allows you annotate existing binaries with arbitrary instructions (for example, to trace, or perform coverage analysis), or to rewrite an existing binary so that it executes more efficiently. Etch works directly on x86 executables. It does not require program source code for either measurement or optimization.

Etch is targeted at two different user groups: developers, who wish to understand the performance of their programs during the development cycle, and users, who wish to understand and improve the performance of common applications executing in their environment.

Etch provides both groups with measurement tools to evaluate performance at several levels of detail, and optimization tools to automatically restructure programs to improve performance, where possible.

How Etch Works

Etch reads executable binaries (and, under Win32, DLLs) for an application, modifies the image, and writes a new one that has been enhanced for measurement or optimization. The transformations performed on the binary by Etch do not change program correctness, although a program transformed for performance measurement collection will run more slowly. Etch does not require changes to the operating system, but a modified Etch binary may utilize OS facilities, such as software timers, or even implementation-specific facilities, such as Intel Pentium performance counters.

Etch is similar in spirit to other binary rewriting tools, such as Digital’s ATOM for the DEC ALPHA, or the University of Wisconsin’s EEL system for the SPARC. Unlike these other tools, which run on RISC-based UNIX systems, Etch works on Win32 x86 executables. This environment creates several challenges for a binary rewriting tool that are not present in UNIX-based environments, including:

- **code discovery.** The x86 has a fairly free-formed instruction set, making it difficult to statically discover code within an executable image, which contains a mixture of code and data. In addition, there are few standards that define an executable’s internal format, beyond that of the Win32 PE header. Internally, a binary can contain code, data, and jump tables laid out in nearly any order, and the format can change from compiler to compiler. A binary rewriting tool must be able to accurately distinguish between code and data, rewriting code, and leaving data intact. Otherwise a program may fail to execute correctly.
• **module discovery.** A Win32 program is actually a combination of a top level executable and an unbounded number of dynamically linked libraries (DLL). These libraries may be statically described by an executable (or by one of its referenced DLLs), or they may invoked dynamically by a program that constructs the DLL name on the fly while it runs. In order to transform such a loosely structured executable, it is necessary to be able to apriori identify all the DLLs used by a program and to transform them before they are loaded. In contrast, most UNIX programs are represented a single, wholly linked executable.

• **environment management.** A Win32 program executes in a context, which includes its working directory and the program name by which it was invoked. Unlike most UNIX programs, many Win32 programs are quite sensitive to this context and may not work when the program is copied, or transformed and moved into some temporary workspace. Consequently, it is necessary to run a transformed executable within a protective "shell" that makes it appear as though the Win32 executable is running in its originally installed directory.

• **user interface.** While UNIX tools are primarily driven from the command line, such an interface is anathema to the Win32 usage model, which encourages rich user interfaces and turn key solutions. A truly useful binary rewriting program for Win32 must be accessible through an easy-to-use graphical user interface.

### Using Etch

There are three key concepts that must be understood to use Etch:

**Instrumentation**

Instrumentation transforms a binary according to an arbitrary criteria. For example, a program may be instrumented to count instructions, or to count the occurrence of each instruction, or to simulate a cache by tracking memory references.

**Data collection**

Once instrumented, an executable can be run. At that time, instrumentation routines collect data about the program.

**Data processing**

Once run, any data generated by an instrumented executable can be processed. Trace-based optimization is a typical data processing phase made possible by Etch.

**Instrumentation**

To instrument a program, Etch is invoked with the name of an executable and a DLL. The DLL provides a set of routines which are invoked for each instruction in the executable. Roughly, Etch operates as:

```plaintext
for each instruction in executable
    InstrumentBefore(instruction);
    InstrumentAfter(instruction);
end;
InstrumentBeforeMainProgram();
InstrumentAfterMainProgram();
```

The instrumentation tool provides implementations of these "Before" and "After" functions. The call back functions can in turn direct Etch to modify the executable with respect to the specific instruction. The directions in effect say "before (or after) this instruction runs, please call some specific function with some specific set of arguments." For example, to count instructions, the InstrumentBefore procedure would
direct Etch to insert code that incremented a counter at runtime. These inserted instructions do not change the correctness of the program. Before and After callbacks are also provided for basic blocks, procedures, and entire modules (DLLs and executables).

Once the entire executable has been scanned and instrumented, Etch writes a new version of the executable that can be run. Any functions referenced in the callback routines, as well as the Etch runtime library are included in the new executable.

**Data Collection**

The executable written by Etch can be run, and any instrumentation routines will run as a side effect of running the program. Instrumentation routines, as the program is running, can inspect the state of the program, for example, the contents of registers, or effective addresses. All addresses, whether text or data, are relative to the original binary, so the collection routines do not have to compensate for the fact that they are part of a modified executable.

**Data Processing**

When an Etched program terminates, its data collection routines can save information about the executable to disk. Later, post-processing utilities can examine the data. For example, a predicted execution time can be determined after the fact based on hypothetical processor, cache, and memory speeds. At a lower level, detailed information about a program’s performance can be obtained such as is shown below in the graph of instruction cache performance for a collection of popular Win32 programs. The graph shows the miss penalty of the first level instruction cache and a second level unified cache for the Perl interpreter, three commercial C++ compilers, and MS-WORD.

![Graph showing instruction cache performance](image)

**Optimization**

Etch also provides facilities for rewriting an executable in order to improve its performance. For example, the instrumentation phase, rather than adding new instructions, can direct Etch to write the executable out according to a different code layout optimized for cache and VM behavior.
The impact of optimization

The graph below shows the reduction in instruction cache misses and execution time (in cycles) for a collection of popular Win32 programs that have been optimized for code layout using Etch on a 90Mhz Pentium. Etch was first used to discover the programs’ locality while executing against a training set, and then rewritten in order to achieve a tighter cache and VM packing. Infrequently executed basic blocks were moved out of line, and frequently interacting basic blocks were laid out contiguously in the executable. The results were measured using inputs different than those used during training.

The User Interface

In addition to a programming interface, Etch also offers a graphical user interface for performing common instrumentation and optimization operations. The user interface can drive the measurement process: it runs Etch on the original binary to produce a new binary, modified to collect the necessary behavioral data; it executes the modified binary to produce the data; and it feeds the data to analysis tools that produce graphs or charts that help to pinpoint problems. Once a problem has been identified, the user may instruct Etch to perform a performance-optimization transformation. For example, Etch may rewrite the original binary to change the layout of data or code in order to improve cache or virtual memory performance.
Sample dialog box from the user interface
Sample results showing distribution of instruction opcodes

Requirements
Etch runs on Intel 486, Pentium and P6 processors with at least 24 MB of memory. Etch works on 32-bit (Win32) binaries. It has been used for programs built by MSVC, Borland, and Intel compilers.

To Learn More
If you’d like to see some traces we’ve generated from a few popular Windows programs on the X86, visit http://etch.eecs.harvard.edu/traces/index.html.