Detours: Binary Interception of Win32 Functions

Galen Hunt and Doug Brubacher
Microsoft Research
One Microsoft Way
Redmond, WA 98052
detours@microsoft.com
http://research.microsoft.com/sn/detours

Abstract

Innovative systems research hinges on the ability to easily instrument and extend existing operating system and application functionality. With access to appropriate source code, it is often trivial to insert new instrumentation or extensions by rebuilding the OS or application. However, in today's world of commercial software, researchers seldom have access to all relevant source code.

We present Detours, a library for instrumenting arbitrary Win32 functions on x86 machines. Detours intercepts Win32 functions by re-writing target *function images. The Detours package also contains utilities to attach arbitrary DLLs and data segments (called* payloads*) to any Win32 binary.*

While prior researchers have used binary rewriting to insert debugging and profiling instrumentation, to our knowledge, Detours is the first package on any platform to logically preserve the un-instrumented target function (callable through a trampoline) as a subroutine for use by the instrumentation. Our unique trampoline design is crucial for extending existing binary software.

We describe our experiences using Detours to create an automatic distributed partitioning system, to instrument and analyze the DCOM protocol stack, and to create a thunking layer for a COM-based OS API. Micro-benchmarks demonstrate the efficiency of the Detours library.

1. Introduction

Innovative systems research hinges on the ability to easily instrument and extend existing operating system and application functionality whether in an application, a library, or the operating system DLLs. Typical reasons to intercept functions are to add functionality, modify returned results, or insert instrumentation for debugging or profiling. With access to appropriate source code, it is often trivial to insert new instrumentation or extensions by rebuilding the OS or application. However, in today's world of commercial development and binary-only releases, researchers seldom have access to all relevant source code.

Detours is a library for intercepting arbitrary Win32 binary functions on *x86* machines. Interception code is applied dynamically at runtime. Detours replaces the first few instructions of the *target function* with an unconditional jump to the user-provided *detour function*. Instructions from the target function are preserved in a *trampoline function*. The trampoline function consists of the instructions removed from the target function and an unconditional branch to the remainder of the target function. The detour function can either replace the target function or extend its semantics by invoking the target function as a subroutine through the trampoline.

Detours are inserted at execution time. The code of the target function is modified in memory, not on disk, thus facilitating interception of binary functions at a very fine granularity. For example, the procedures in a DLL can be detoured in one execution of an application, while the original procedures are not detoured in another execution running at the same time. Unlike DLL re-linking or static redirection, the interception techniques used in the Detours library are guaranteed to work regardless of the method used by application or system code to locate the target function.

While others have used binary rewriting for debugging and to inline instrumentation, Detours is a general-purpose package. To our knowledge, Detours is the first package on any platform to logically preserve the un-instrumented target function as a subroutine callable through the trampoline. Prior systems logically prepended the instrumentation to the target, but did not make the original target's functionality available as a general subroutine. Our unique trampoline design is crucial for extending existing binary software.

In addition to basic detour functionality, Detours also includes functions to edit the DLL import table of any binary, to attach arbitrary data segments to existing binaries, and to inject a DLL into either a new or an existing process. Once injected into a process, the instrumentation DLL can detour any Win32 function, whether in the application or the system libraries.

The following section describes how Detours works. Section 0 outlines the usage of the Detours library. Section 4 describes alternative function-interception techniques and presents a micro-benchmark evaluation of Detours. Section 5 details the usage of Detours to produce distributed applications from local applications, to quantify DCOM overheads, to create a thunking layer for a new COM-based Win32 API, and to implement first chance exception handling. We compare Detours with related work in Section 6 and summarize our contributions in Section 7.

2. Implementation

Detours provides three important sets of functionality: the ability to intercept arbitrary Win32 binary functions on *x86* machines, the ability to edit the import tables of binary files, and the ability to attach arbitrary data segments to binary files. We will describe the implementation of each of these functionalities.

2.1. Interception of Binary Functions

The Detours library facilitates the interception of function calls. Interception code is applied dynamically at runtime. Detours replaces the first few instructions of the *target function* with an unconditional jump to the user-provided *detour function*. Instructions from the target function are preserved in a *trampoline function*. The trampoline consists of the instructions removed from the target function and an unconditional branch to the remainder of the target function.

When execution reaches the target function, control jumps directly to the user-supplied detour function. The detour function performs whatever interception *preprocessing* is appropriate. The detour function can return control to the *source* function or it can call the trampoline function, which invokes the target function without interception. When the target function completes, it returns control to the detour function. The detour function performs appropriate *postprocessing* and returns control to the source function. Figure 1 shows the logical flow of control for function invocation with and without interception.

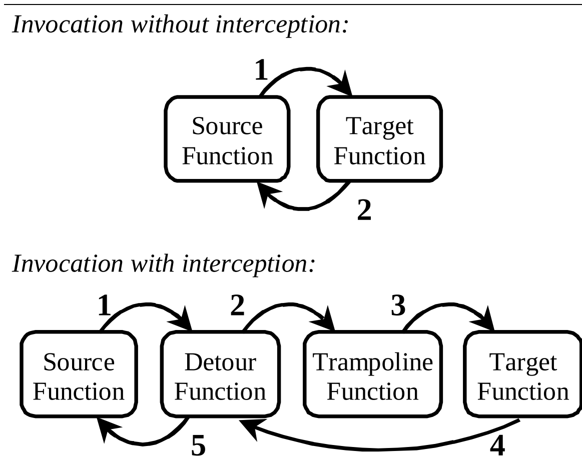

Figure 1. Invocation with and without interception.

The Detours library intercepts target functions by rewriting their in-process binary image. For each target function, Detours actual rewrites two functions: the target function and the matching trampoline function. The trampoline function can be allocated either dynamically or statically. A statically allocated trampoline always invokes the target function without the detour. Prior to insertion of a detour, the static trampoline contains a single jump to the target. After insertion, the trampoline contains the initial instructions from the target function and a jump to the remainder of the target function.

Statically allocated trampolines are extremely useful for instrumentation programmers. For example, in Coign [7], invoking the `Coign_CoCreateInstance` trampoline is equivalent to invoking the original `CoCreateInstance` function without instrumentation. Coign internal functions can call `Count_CoCreateInstance` at any time to create a new component instance without concern for whether or not the original function has been rerouted with a detour.

```
;; Target Function
        …
TargetFunction:
  push  ebp
  mov   ebp,esp
  push  ebx
  push  esi
  push  edi
        …

;; Trampoline
        …
TrampolineFunction:
  jmp  TargetFunction
        …
```

```
;; Target Function
        …
TargetFunction:
  jmp   DetourFunction


TargetFunction+5:
  push  edi
        …

;; Trampoline
        …
TrampolineFunction:
  push  ebp
  mov   ebp,esp
  push  ebx
  push  esi
  jmp   TargetFunction+5
        …
```

Figure 2. Trampoline and target functions, before and after insertion of the detour (left and right).

Figure 2 shows the insertion of a detour. To detour a target function, Detours first allocates memory for the dynamic trampoline function (if no static trampoline is provided) and then enables write access to both the target and the trampoline. Starting with the first instruction, Detours copies instructions from the target to the trampoline until at least 5 bytes have been copied (enough for an unconditional jump instruction). If the target function is fewer than 5 bytes, Detours aborts and returns an error code. To copy instructions, Detours uses a simple table-driven disassembler. Detours adds a jump instruction from the end of the trampoline to the first non-copied instruction of the target function. Detours writes an unconditional jump instruction to the detour function as the first instruction of the target function. To finish, Detours restores the original page permissions on both the target and trampoline functions and flushes the CPU instruction cache with a call to `FlushInstructionCache`.

2.2. Payloads and DLL Import Editing

While a number of tools exist for editing binary files [10, 12, 13, 17], most systems research doesn't require such heavy-handed access to binary files. Instead, it is often sufficient to add an extra DLL or data segment to an application or system binary file. In addition to detour functions, the Detours library also contains fully reversible support for attaching arbitrary data segments, called *payloads*, to Win32 binary files and for editing DLL import tables.

Figure 3 shows the basic structure of a Win32 Portable Executable (PE) binary file. The PE format for Win32 binaries is an extension of COFF (the Common Object File Format). A Win32 binary consists of a DOS compatible header, a PE header, a text section containing program code, a data section containing initialized data, an import table listing any imported DLLS and functions, an export table listing functions exported by the code, and debug symbols. With the exception of the two headers, each of the other sections of the file is optional and may not exist in a given binary.

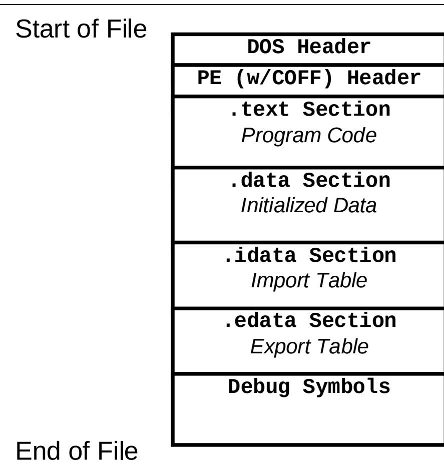

Figure 3. Format of a Win32 PE binary file.

To modify a Win32 binary, Detours creates a new `.detours` section between the export table and the debug symbols. Note that debug symbols must always reside last in a Win32 binary. The new section contains a detours header record and a copy of the original PE header. If modifying the import table, Detours creates the new import table, appends it to the copied PE header, then modifies the original PE header to point to the new import table. Finally, Detours writes any user payloads at the end of the `.detours` section and appends the debug symbols to finish the file. Detours can reverse modifications to the Win32 binary by restoring the original PE header from the `.detours` section and removing the `.detours` section. Figure 4 shows the format of a Detours-modified Win32 binary.

Creating a new import table serves two purposes. First, it preserves the original import table in case the programmer needs to reverse all modifications to the Win32 file. Second, the new import table can contain renamed import DLLs and functions or entirely new DLLs and functions. For example, Coign [7] uses Detours to insert an initial entry for `coignrte.dll` into each instrumented application. As the first entry in the applications import table, `coignrte.dll` always is the first DLL to run in the application's address space.

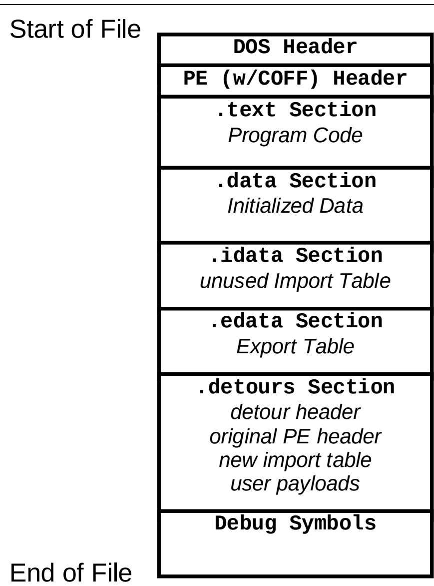

Figure 4. Format of a Detours-modified binary file.

Detours provides functions for editing import tables, adding payloads, enumerating payloads, removing payloads, and rebinding binary files. Detours also provides routines for enumerating the binary files mapped into an address space and locating payloads within those mapped binaries. Each payload is identified by a 128-bit globally unique identifier (GUID). Coign uses Detours to attach per-application configuration data to application binaries.

In cases where instrumentation need be inserted into an application without modifying binary files, Detours provides functions to inject a DLL into either a new or an existing process. To inject a DLL, Detours writes a `LoadLibrary` call into the target process with the `VirtualAllocEx` and `WriteProcessMemory` APIs then invokes the call with the `CreateRemoteThread` API.

3. Using Detours

The code fragment in Figure 5 illustrates the usage of the Detours library. User code must include the `detours.h` header file and link with the `detours.lib` library.

```
#include <windows.h>
#include <detours.h>

VOID (*DynamicTrampoline)(VOID) = NULL;

DETOUR_TRAMPOLINE(
  VOID WINAPI SleepTrampoline(DWORD),
  Sleep
);

VOID WINAPI SleepDetour(DWORD dw)
{
  return SleepTrampoline(dw);
}

VOID DynamicDetour(VOID)
{
  return DynamicTrampoline();
}

void main(void)
{
  VOID (*DynamicTarget)(VOID) = SomeFunction;

  DynamicTrampoline
    =(FUNCPTR)DetourFunction(
       (PBYTE)DynamicTarget,
       (PBYTE)DynamicDetour);

  DetourFunctionWithTrampoline(
    (PBYTE)SleepTrampoline,
    (PBYTE)SleepDetour);

  // Execute the remainder of program.

  DetourRemoveTrampoline(SleepTrampoline);
  DetourRemoveTrampoline(DynamicTrampoline);
}
```

Figure 5. Sample Instrumentation Program.

Trampolines may be created either statically or dynamically. To intercept a target function with a static trampoline, the application must create the trampoline with the `DETOUR_TRAMPOLINE` macro. `DETOUR_TRAMPOLINE` takes two arguments: the prototype for the static trampoline and the name of the target function.

Note that for proper interception the prototype, target, trampoline, and detour functions must all have exactly the same call signature including number of arguments and calling convention. It is the responsibility of the detour function to copy arguments when invoking the target function through the trampoline. This is intuitive as the target function is just a subroutine callable by the detour function.

Using the same calling convention insures that registers will be properly preserved and that the stack will be properly aligned between detour and target functions.

Interception of the target function is enabled by invoking the `DetourFunctionWithTrampoline` function with two arguments: the trampoline and the pointer to the detour function. The target function is not given as an argument because it is already encoded in the trampoline.

A dynamic trampoline is created by calling `DetourFunction` with two arguments: a pointer to the target function and a pointer to the detour function. `DetourFunction` allocates a new trampoline and inserts the appropriate interception code in the target function.

Static trampolines are extremely easy to use when the target function is available as a link symbol. When the target function is not available for linking, a dynamic trampoline can be used. Often a function pointer to the target function can be acquired from a second function. For those times, when a pointer to the target function is not readily available, `DetourFindFunction` can find the pointer to a function when it is either exported from a known DLL, or if debugging symbols are available for the target function's binary[1].

`DetourFindFunction` accepts two arguments, the name of the binary and the name of the function. `DetourFindFunction` returns either a valid pointer to the function or `NULL` if the symbol for the function could not be found. `DetourFindFunction` first attempts to locate the function using the Win32 `LoadLibrary` and `GetProcAddress` APIs. If the function is not found in the export table of the DLL, `DetourFindFunction` uses the `ImageHlp` library to search available debugging symbols. The function pointer returned by `DetourFindFunction` can be given to `DetourFunction` to create a dynamic trampoline.

Interception of a target function can be removed by invoking the `DetourRemoveTrampoline` function.

Note that because the functions in the Detours library modify code in the application address space, it is the programmer's responsibility to ensure that no other threads are executing in the address space while a detour is inserted or removed. An easy way to insure single-threaded execution is to call functions in the Detours library from a `DllMain` routine.

4. Evaluation

Several alternative techniques exist for intercepting function calls. Alternative interception techniques include:

Call replacement in application source code. Calls to the target function are replaced with calls to the detour function by modifying application source code. The major drawback of this technique is that it requires access to source code.

Call replacement in application binary code. Calls to the target function are replaced with calls to the detour function by modifying application binaries. While this technique does not require source code, replacement in the application binary does require the ability to identify all applicable call sites. This requires substantial symbolic information that is not generally available for binary software.

DLL redirection. If the target function resides in a DLL, the DLL import entries in the binary can be modified to point to a detour DLL. Redirection to the detour DLL can be achieved by either replacing the name of the original DLL in the import table before load time or replacing the function addresses in the indirect import jump table after load [2]. Unfortunately, redirecting to the detour DLL through the import table fails to intercept DLL internal calls and calls on pointers obtained from the `LoadLibrary` and

[1] Microsoft ships debugging symbols for the entire Windows NT operation system as part of the retail release. These symbols can be found in the `\support\symbols` directory on the OS distribution media.

GetProcAddress APIs early in an applications execution.

Breakpoint trapping. Rather than replace the DLL, the target function can be intercepted by inserting a debugging breakpoint into the target function. The debugging exception handler can then invoke the detour function. The major drawback to breakpoint trapping is that debugging exceptions suspend all application threads. In addition, the debug exception must be caught in a second operating-system process. Interception via break-point trapping has a high performance penalty.

Table 1 lists times for intercepting either an empty function or the CoCreateInstance API. Times are on a 200 MHz Pentium Pro. Rows list the time to invoke the functions without interception, with interception through call replacement, with interception through DLL redirection, with interception using the Detours library, or with interception through breakpoint trapping. As can be seen, function interception with Detours library has only minimal overhead (less than 400 ns in either case).

Interception Technique	**Intercepted Function**	
	Empty Function	CoCreate-Instance
Direct	0.113µs	14.836µs
Call Replacement	0.143µs	15.193µs
DLL Redirection	0.143µs	15.193µs
Detours Library	0.145µs	15.194µs
Breakpoint Trap	229.564µs	265.851µs

Table 1. Comparison of Interception Techniques.

5. Experience

The Detours package has been used extensively in Microsoft Research over the last two years to instrument and extend Win32 applications and the Windows NT operating system.

Detours was originally developed for the Coign Automatic Distributed Partition System [7]. Coign converts local desktop applications built from COM components into distributed client-server applications. During profiling, Coign uses Detours to intercept calls to COM instantiation functions such as CoCreateInstance. The detour functions invoke the original library functions through trampolines, then wrap output interface pointers in an additional instrumentation layer (for more details see [8]). The instrumentation layer measures inter-component communication to determine how application components should be partitioned across a network. During distributed executions, new Coign detour functions intercept calls to COM instantiation functions and re-route those calls to distributed machines. In essence, Coign extends the COM library to support intelligent remote invocation. Whereas DCOM supports remote invocation of a few COM instantiation functions, Coign supports remote invocation for approximately 50 COM functions through detour extensions. Coign uses Detours' DLL redirection functions to attach a runtime loader and the payload functions to attach profiling data to application binaries.

Our colleagues have used Detours to instrument the user-mode portion of the DCOM protocol stack including marshaling proxies, DCOM runtime, RPC runtime, WinSock runtime, and marshaling stubs [11]. The resultant detailed analysis was then used to drive a re-architecture of DCOM for fast user-mode networks. While they could have used source code modifications to produce a special profiling version of DCOM, the source-based instrumentation would have been version dependent and shared by all DCOM applications on the profiling machine. With binary instrumentation based on Detours, the profiling tool can be attached to any Windows NT 4 build of DCOM and only effects the process being profiled.

In another extension exercise, Detours was used to create a thunking layer for COP (the Component-based Operating System Proxy) [14]. COP is a COM-based version of the Win32 API. COP aware applications access operating system functionality through COM interfaces, such as IWin32FileHandle. Because the COP interfaces are distributable with DCOM, a COP application can use OS resources, including file systems, keyboards, mice, displays, registries, etc., from any machine in a network. To provide support for legacy applications, COP uses detour functions to intercept all application calls to the Win32 APIs. Native application API calls are converted to calls on COP interfaces. At the bottom, the COP implementation communicates with the underlying operating system through trampoline functions. COP requires no modifications to application binaries. At load time, the COP DLL is injected into the application's address space with Detours' injection functions. Through its simple interception, Detours has facilitated this massive extension of the Win32 API.

Finally, to support Software Distributed Shared Memory (SDSM) systems, we have implemented a first chance exception filter for Win32 structured exception handling. The Win32 API contains an API, `SetUnhandledExceptionFilter`, through which an application can specify an exception filter to execute should no other filter handle an application exception. For applications such as SDSM systems, the programmer would like to insert a first-chance exception filter to remove page faults caused by the SDSM's manipulation of VM page permissions. Windows NT does not provide such a first-chance exception filter mechanism. A simple detour intercepts the exception entry point from kernel mode to user mode (`KiUserExceptionDispatcher`). With only a few lines of code, the detour function calls a user-provided first-chance exception filter and then forwards the exception, if unhandled, to the default exception mechanism through a trampoline.

6. Related Work

Detours are an extension of the general technique of code patching. To intercept execution, an unconditional branch or jump is inserted into the desired point of interception in the target function. Code overwritten by the unconditional branch is moved to a code patch. The code patch consists of either the instrumentation code or a call to the instrumentation code followed by the instructions moved to insert the unconditional branch and a jump to the first instruction in the target function after the unconditional branch. Logically, a code patch can be prepended to the beginning of a function, inserted at some arbitrary point in a function, or appended to the end of a function.

Whereas a code patch invokes instrumentation then continues the target function, our technique transfers control completely to the detour function which can invoke the original target function through the trampoline at its leisure. The trampoline gives instrumentation complete freedom to invoke the semantics of the original function as a callable subroutine at any time.

Techniques for code patching have existed since the dawn of digital computing [3-5, 9, 15]. Code patching has been applied to insert debugging or profiling code. In the distant past, code patching was generally considered to be a much more practical update method than re-compiling the entire application. In addition to debugging and profiling, Detours has also been used to resourcefully extend the functionality of existing systems [7, 14].

While recent systems have extended code patching to parallel applications [1] and system kernels [16], Detours is to our knowledge the only code patching system that preserves the semantics of the target function as a callable subroutine. The detour function replaces the target function, but can invoke its functionality at any point through the trampoline. Our unique trampoline design makes it trivial to extend the functionality of existing binary functions.

Recent research has produced a class of detailed binary rewriting tools including Atom [13], Etch [12], EEL [10], and Morph [17]. In general, these tools take as input an application binary and an instrumentation script. The instrumentation script passes over the binary inserting code between instructions, basic blocks, or functions. The output of the script is a new, instrumented binary. In a departure for earlier systems, DyninstAPI [6] can modify applications dynamically.

Detours' primary advantage over detailed binary rewriters is its size. Detours adds less than 18KB to an instrumentation package whereas detailed binary rewriters add at least a few hundred KB. The cost of Detours small size is an inability to insert code between instructions or basic blocks. Detailed binary rewriters can insert instrumentation around any instruction through sophisticated features such as free register discovery. Detours relies on adherence to calling conventions in order to preserve register values. While detailed binary rewriters support insertion of code before or after any basic instruction unit, they do not preserve the semantics of the uninstrumented target function as a callable subroutine.

7. Conclusions

The Detours library provides an import set of tools to the arsenal of the systems researcher. Detour functions are fast, flexible, and friendly. A detour of `CoCreateInstance` function has less than a 3% overhead, which is an order of magnitude smaller than the penalty for breakpoint trapping. The Detours library is very small. The runtime consists of less than 40KB of compiled code although typically less than 18KB of code is added to the users instrumentation.

We are currently working on versions of Detours for Windows 98 and the Alpha processors. The Alpha port should be trivial due to the uniform size of instructions in the Alpha's RISC architecture.

Unlike DLL redirection, the Detours library intercepts both statically and dynamically bound invocations. Finally, the Detours library is much more flexible than DLL redirection or application code modification. Interception of any function can be selectively enabled or disabled for each process individually at execution time.

Our unique trampoline preserves the semantics of the original, uninstrumented target function for use as a subroutine of the detour function. Using detour functions and trampolines, it is trivial to produce compelling system extensions without access to system source code and without recompiling the underlying binary files. Detours makes possible a whole new generation of innovative systems research on the Windows NT platform.

Availability

The Detours library is freely available for research purposes. It can be found in either source form or as a compiled library at `http://research.microsoft.com/sn/detours`.

References

[1] Aral, Ziya, Illya Gertner, and Greg Schaffer. Efficient Debugging Primitives for Multiprocessors. *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 87-95. Boston, MA, April 1989.

[2] Balzer, Bob. Mediating Connectors, Release 2.0. University of Southern California, Information Sciences Institute, Los Angeles, CA, April 1999.

[3] Digital Equipment Corporation. *DDT Reference Manual*, 1972.

[4] Evans, Thomas G. and D. Lucille Darley. DEBUG -- An Extension to Current Online Debugging Techniques. *Communications of the ACM*, 8(5), pp. 321-326, May 1965.

[5] Gill, S. The Diagnosis of Mistakes in Programmes on the EDSAC. *Proceedings of the Royal Society, Series A*, 206, pp. 538-554, May 1951.

[6] Hollingsworth, Jeffrey K. and Bryan Buck. DyninstAPI Programmer's Guide, Release 1.2. Computer Science Department, University of Maryland, College Park, MD, September 1998.

[7] Hunt, Galen C. and Michael L. Scott. The Coign Automatic Distributed Partitioning System. *Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI '99)*, pp. 187-200. New Orleans, LA, February 1999. USENIX.

[8] Hunt, Galen C. and Michael L. Scott. Intercepting and Instrumenting COM Applications. *Proceedings of the Fifth Conference on Object-Oriented Technologies and Systems (COOTS'99)*, pp. *to Appear*. San Diego, CA, May 1999. USENIX.

[9] Kessler, Peter. Fast Breakpoints: Design and Implementation. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 78-84. White Plains, NY, June 1990.

[10] Larus, James R. and Eric Schnarr. EEL: Machine-Independent Executable Editing. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 291-300. La Jolla, CA, June 1995.

[11] Li, Li, Alessandro Forin, Galen Hunt, and Yi-Min Wang. High-Performance Distributed Objects over a System Area Network. *Proceedings of the Third USENIX NT Symposium*. Seattle, WA, July 1999.

[12] Romer, Ted, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and J. Bradley Chen. Instrumentation and Optimization of Win32/Intel Executables Using Etch. *Proceedings of the USENIX Windows NT Workshop 1997*, pp. 1-7. Seattle, WA, August 1997. USENIX.

[13] Srivastava, Amitabh and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 196-205. Orlando, FL, June 1994.

[14] Stets, Robert J., Galen C. Hunt, and Michael L. Scott. Component-based Operating System APIs: A Versioning and Distributed Resource Solution. *IEEE Computer*, 32(7), July 1999.

[15] Stockham, T.G. and J.B. Dennis. FLIT- Flexowriter Interrogation Tape: A Symbolic Utility Program for the TX-0. Department of Electical Engineering, MIT, Cambridge, MA, Memo 5001-23, July 1960.

[16] Tamches, Ariel and Barton P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, pp. 117-130. New Orleans, LA, February 1999. USENIX.

[17] Zhang, Xiaolan, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System Support for Automated Profiling and Optimization. *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*. Saint-Malo, France, October 1997.