Using Program Metadata to Support SDT in Object-Oriented Applications

Daniel Williams
University of Virginia
dan_williams@cs.virginia.edu

Jason D. Hiser
University of Virginia
hiser@cs.virginia.edu

Jack W. Davidson
University of Virginia
jwd@cs.virginia.edu

ABSTRACT

Software dynamic translation (SDT) is a powerful technology that enables software malleability and adaptivity at the instruction level by providing facilities for run-time monitoring and code modification. SDT has been used as the basis for many valuable tools, including dynamic optimizers, profilers, security policy enforcement, and binary translation to name a few. However, modern object-oriented programming techniques and their implementations (e.g., virtual functions, exceptions, dynamic code, etc.) pose unique challenges to high performing SDT systems. In this paper, we present Metaman, a generalized program metadata manager that stores and manages program information so that it can be efficiently accessed by emerging SDT systems to improve overall runtime performance of a managed executable. Using the information collected by Metaman, the run-time performance of an existing SDT system was improved by 22% making its execution speed only 3% slower than native (i.e., non-managed) execution.

Keywords

Performance, Software Dynamic Trasnlation, Object-Oriented Systems

1. INTRODUCTION

Software Dynamic Translation (SDT) systems have been used as the basis for many valuable tools, including dynamic optimizers, profilers, security policy enforcers, dynamic bug patchers, and binary translators, among others [2, 4, 8, 9, 12]. However, for these tools to be most effective, they must be able to execute code generated from any source, and incur low overhead with commonly generated code. Compiled object-oriented languages such as C++ generate sophisticated machine code for language features such as virtual functions and exception handling. Many of these features result in frequent use of indirect branches.

Previous research has shown that indirect branch handling is a major source of overhead for SDT systems [7, 6]. Targets of indirect branches must be checked at runtime to ensure that the branch target code has been translated and that control is transferred to the translated code. This check must be performed every time an indirect branch is executed. The standard techniques for handling indirect branches attempt to minimize the impact of the indirect branch handling code by creating highly customized inline assembly designed to be as small and fast as possible, and cause as few secondary cache effects as possible. This work presents two optimizations for compiled object-oriented languages such as C++. We use program metadata gathered at compile time to improve the SDT handling of virtual function tables (VFT), as well as call-return handling in the presence of exceptions. If the SDT system has compiler-level information about these constructs (e.g., layout of the VFT), it can generate a more efficient translation of the indirect branch by translating the VFT in its entirety.

To collect the metadata needed for these optimizations, we have developed a metadata manager, *Metaman*, which collects and maintains program metadata throughout the program's development cycle. Many tools collect a wide variety of metadata about the program's structure and expected behavior. However, this information is typically discarded once the tool finishes. Metaman allows individual tools to submit and retrieve program metadata, saving redundant analysis and making important program metadata available to any tool that needs it. Metaman collects metadata at various points in the compilation process, stores the information within an XML database, which can then be queried by subsequent tools. For object-oriented languages running under an SDT, Metaman collects the symbolic location of VFT tables and the locations of compiler-emitted return instructions to enable the SDT's OO-specific optimizations.

Our system has been built and tested on the GNU toolchain with C and C++. We tested the effectiveness of the SDT/C++ optimizations on SPEC CPU2006 benchmarks. Using the SPEC benchmarks running natively as a baseline, we were able to improve the performance of the SDT 22% with our optimizations, yielding performance within 3% of native speeds.

The primary contributions of this work are as follows:

1. We present the design of the Metaman metadata manager for handling arbitrary program metadata, whereas previous systems have stored only application- and system-specific data.

2. We present an optimization of indirect branch handling for virtual function tables in an SDT system. The optimization reduced overhead of a microbenchmark by 51%, and the `Xalan` benchmark by 15%

ICOOOLPS'09, July 2009, Genoa, Italy

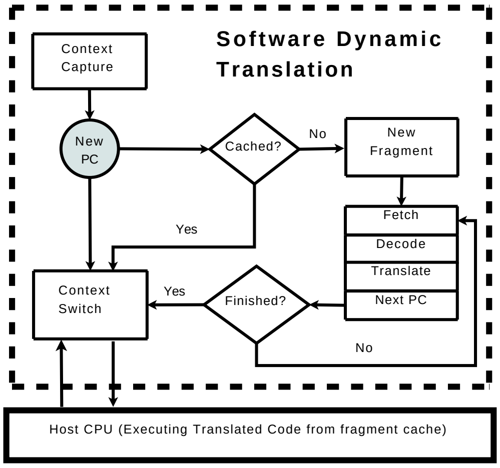

Figure 1: Overview of Software Dynamic Translation (SDT)

3. We give a design and implementation of an SDT optimization to improve return handling, including mechanism to repair the application's stack after an exception occurs, generalizing the transparency of the stack-modifying return translation mechanism.

2. SDT BACKGROUND

To demonstrate the utility of persistent storage of program metadata, we have used Metaman to improve the run time of Strata, an existing SDT system. This section gives basic background knowledge about the operation of SDT systems, and about possible indirect branch handling mechanisms within them. Other publications provide a more in-depth look at SDT and indirect branches [2, 3, 4, 6, 7, 13, 16].

2.1 SDT Overview

SDT is a class of tools that provide important functionality to system developers. SDT systems fetch, examine and potentially modify every machine instruction before it is executed. As shown in Figure 1, when the SDT encounters a new instruction address (i.e., PC), it begins translating that instruction, along with subsequent instructions until a termination point is reached – usually a branch or return. Once the termination condition is reached, the SDT places code to transfer control back to the SDT system at the end of the translated code blocks, called a *trampoline*. Then the translated code is executed in the native environment. Most SDT systems utilize a code cache (or a *fragment cache*) to save and reuse already translated blocks of code (*fragments*). By enabling a code cache, and directly linking code blocks in the cache, most of the cost of translation can be amortized across the run of the program. To alter the functionality of the SDT, all that is required is to change the translation step for the relevant instructions. For example, to prevent execution of code on the stack (a common attack method), an additional step can be added to translation to also check if the SDT is attempting to translate code on the stack. If it is, the SDT can halt execution [15].

For our experiments we are using the Strata SDT framework [17]. Strata is designed for low-overhead translation and execution, as well as flexibility. The flexible design of Strata allows the default translation process to be overridden to implement new SDT-based tools. Strata has been used for a wide range of purposes, including instruction set randomization [8], profiling [12], memory safety [15], and software patching. Because of this diversity, Strata is an ideal tool to test the advantages of supplying program metadata to SDT systems.

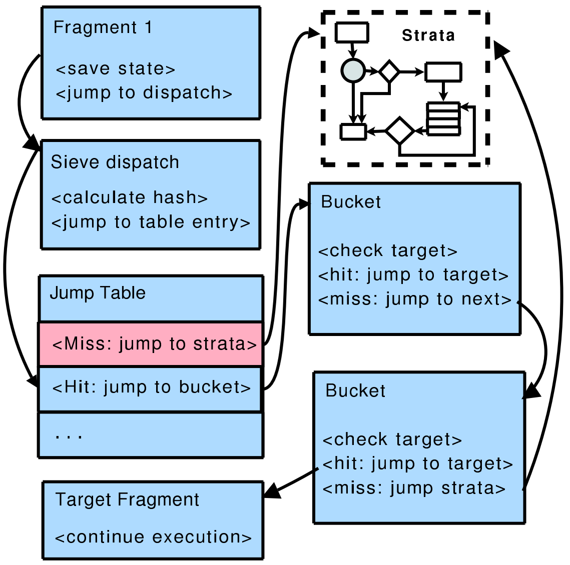

Figure 2: High-level operation of the sieve

2.2 Indirect Branches

Indirect branches and indirect calls require special handling by SDT systems. Since indirect branch targets can change during execution, they cannot be linked at translation time to their targets in the fragment cache as is done with direct branch instructions. Typically SDT systems use an inline hash table lookup to handle indirect branch targets either in data or in code. On x86, the code hashtable, called a sieve, has the advantage of having an implementation which does not require saving the `EFLAGS` register [19]. The high-level operation of the sieve is shown in Figure 2. The sieve stores the target application address, and then uses it to compute a hash value. It then jumps to an entry in the sieve table, which immediately jumps to either a "bucket" or back to Strata. On the x86 architecture, a call to the address `[eax+8]` results in the following sieve (Intel syntax):

```
0x2058: push   0x3e38 ; return address
0x205d: push   DWORD PTR [eax+8] ; tmp loc
0x2060: jmp    0xd05a
        ; . . .
0xd05a: push   ecx            ; save state
0xd05b: mov    ecx,DWORD PTR [esp+4]
0xd05f: lea    ecx,[ecx*4]  ; shift left
0xd066: movzx  ecx,cx         ; and 0xffff
        ; shift and add base
0xd069: lea    ecx,[ecx+ecx+0x500c]
0xd070: jmp    ecx
```

Here, `0x500c` is the the base of jump table. Upon initialization this table is filled with `jmp` instructions that transfer control to a trampoline then back to Strata. Then the table is updated with a jump to a bucket with (now-translated) target. The next time that target goes through the sieve table, it jumps to the bucket:

```
          ; target of ecx
0x50a0: jmp     0x2100
          ; first bucket
0x2100: mov     ecx,DWORD PTR [esp+4]
          ; add the complement
0x2104: lea     ecx,[ecx+0xc250]
0x210a: jecxz   0x2111  ; check target
          ; miss - jump to next bucket
0x210c: jmp     0xed089
          ; hit case
0x2111: pop     ecx ; restore state
0x2112: lea     esp,[esp+4]
0x2116: jmp     0x20db  ; target frag
```

The bucket is responsible for checking the target value against the key, and in the case of a miss it goes to the next bucket, or returns to Strata if it is the last bucket. In the case of a hit the conditional branch at `0x210a` is taken, the state is restored, and control is transferred to the target fragment [7, 19]. This technique for indirect branch handling is able to handle any type of indirect branch, however, every time the indirect branch is executed, this code must run to ensure Strata has translated the target. In the case where the target is in the first bucket (which is expected for a reasonably size table), the single indirect branch translates to at least 16 instructions.

3. METADATA SERVICES

The SDT performance problems associated with indirect branches caused by object oriented languages are an example of a greater need for program metadata across the software development toolchain. The software development toolchain has become increasingly complex. Compilers often make many optimization passes, collecting a variety of information about the program's structure. For example, static analysis collects data to determine what is and is not allowable behavior in the program. However, access to this metadata is not evenly distributed across the toolchain. As a program is translated from a high-level language into an executable binary, it goes through a variety of representations, some of which result in information loss from the previous representation. Typically once the final executable is created, almost no metadata about the program is available.

The lack of information about how the program is structured and how it was created represents a semantic gap between the intent of programmers and the running program. For example, in the case of object-oriented languages, the information about a class's virtual function table is reduced to a small array of memory accessed by individual indirect branch instructions. Metadata which informs about the high-level meaning of these structures can be very valuable to the running program.

3.1 Metaman Structure

Effective management of metadata within a traditional software development toolchain is not a simple matter. Traditional compilers follow the Unix pattern of using many small programs with well-defined interfaces in order to accomplish a large task. Separate tools are invoked to preprocess, compile, assemble, and link a program. Modern Integrated Development Environments (IDEs) give the appearance of having a more monolithic approach; however, most still use external programs, and logically isolated functionality ("plug-ins") to perform individual compilation tasks. Therefore, in order to associate toolchain metadata to specific program artifacts (intermediate code files, executable, etc.), the data must be correlated across each stage of the compilation. Metaman achieves this goal by integrating with the build system and collecting metadata at each stage of compilation, organizing and persisting it as XML, and presenting it to other tools within the toolchain.

To allow communication between layers of the software development toolchain, Metaman acts as an intermediary to asynchronously store and organize useful metadata. Figure 3 shows Metaman's relation to the software development toolchain and runtime system.

Metaman integrates directly with the program's build system. Through the build system it tracks the actions of each tool. Metaman is able to leverage the build system's dependency tracking to relate the input and output of individual tools, and associate the metadata collected from that tool with the output file. Once the linker is invoked, the metadata server aggregates data from each of the object files to create an XML file of metadata for the entire program. The linker can also provide important information, mapping symbols to addresses.

There is potentially a very large amount of metadata that users might want to store with a metadata server, including information about high-level data structures, data and control flow analysis done by the compiler, and symbol information collected by the linker. Further, the metadata server must be flexible enough to handle new metadata as more information is collected by and provided to components of the software development toolchain. To achieve this flexibility, XML is used to store the metadata. XML parsers are able to ignore unknown tags, which means that a parser can effectively identify and ignore additions to the types of metadata collected.

Figure 4 shows a sample XML entry for a program's function information. It is roughly modeled after the DWARF debugging information that is typically attached when programs are compiled with `-g`. The top-level organization of the XML file breaks the file into a number of `object` elements (lines 3-13 of Figure 4), which map to originating object files. The `object` element contains information typically found in debugging information, such as function names and addresses (line 4), and parameters (lines 5 and 6). It also contains VFT information need for the optimizations described in Section 4. The `vft` element (line 12) contains the location and size of the virtual function table, which can be used by Strata. The `function` element can then include references to the VFT that occur in the function.

Metaman collects the VFT information via an analysis pass over the assembly code. Identifying VFTs is a compiler-specific operation, subject to the object model used by the compiler. We used the GNU C++ compiler, `g++`. It uses a name mangling scheme that assigns `_ZTV` to virtual function tables. Instructions referencing the virtual funtion tables are labeled, as shown here:

```
.global _vft_info_mb_6_xxx
_vft_info_mb_6_xxx:
```

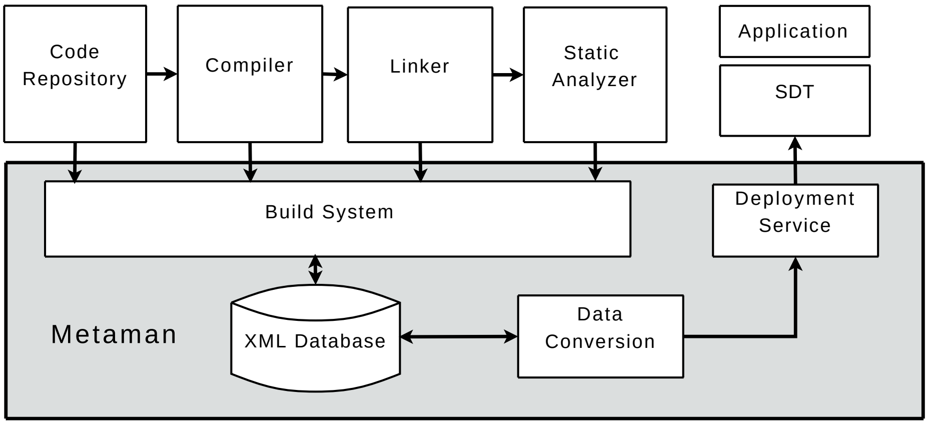

Figure 3: Overview of Metaman, the metadata manager.

```
<?xml version="1.0" encoding="UTF-8"?>
<metaman>
  <object name="mb.cpp" id="0x5d">
    <function name="main" id="0x72" address="0x0804836e" end-address="0x080483ce">
      <param id="0x27d" name="argc" type-ref="0x253" loc="DW_OP_breg3 0x0"/>
      <param id="0x28c" name="argv" type-ref="0x2ea" loc="DW_OP_breg3 0x4" />
      <return address="0x080483a0">
    </function>
    <function name="_ZN3CarC2EPcS0_" >
      <vft-ref name="_vft_info_mb_6_xxx" address="0x08073d44" vft-id="0x3c0" />
    </function>
    <vft id="0x3c0" name="_ZTV3Car" size="0x14" address="0x080d0c60">
  </object>
</metaman>
```

Figure 4: Sample XML for symbol table and function information.

```
        movl     $_ZTV3Car+8, (%ebx)
```

Metaman stores the labels, and uses them to resolve the final address when the object is linked.

3.2 Data Conversion

Metaman stores its information as XML, however, XML is not always the ideal format for all tools. In particular, for performance-sensitive applications like SDT including a heavy-weight XML parser is inappropriate. To address this issue, Metaman includes data conversion support to emit the needed metadata into a compact binary format. For example, Strata expects the VFT information to be a list of binary addresses of the location of the table, followed by a 32-bit integer encoding the size of the table. After the table locations are listed, the references are then encoded as binary addresses. The binary format is a subset of the XML data, but customized and reorganized to serve the needs of the particular tool. This technique allows the data to be read into SDT data structures from a separate binary file, or even included as an additional section in the ELF binary. Ultimately we hope to allow users to write xpath queries with scanf-like formatting to automatically generate the desired binary.

4. VFT AND RETURN HANDLING

Section 2.2 gave an overview of the sieve mechanism for handling indirect branches within an SDT. The sieve is a very general approach that is designed to handle indirect branches that can target any program address. The cost of the sieve, however, is significant; a single indirect branch is translated into 16 or more instructions. This section presents improved handling for returns and virtual function tables using metadata collected by Metaman. Given additional high-level metadata about the structures that produced the indirect branches, Strata can re-create translated versions of these structures, eliminating the overhead of the general indirect branch handler.

4.1 Metadata based Return Handling

The most common type of indirect branch to occur in most programs is a return instruction. Return instructions are designed to return to the instruction directly after the instruction that called the function. Due to the modular nature of modern object-oriented programs, which increasingly contain a large number of small functions, efficient return handling is vitally important.

Normally, Strata converts a call to a pair of instructions, the first pushing the application address, and the second jumping to the target function. Then the return instruction is treated as a standard indirect branch, handled by the sieve. However, because that results in at least 16 instructions, the dynamic execution count of small functions can be greatly increased. Fortunately, the call-return semantics of structured programs is much more regular than the other indirect branch mechanisms. Typically, a call instruction pushes the return address on to the stack, and it is stored there, unaltered, until a return instruction jumps back to that address. One way to improve upon the general sieve approach for returns is to have Strata use the stack for returns. Using this approach, Strata translates a call by emitting code that pushes a fragment cache address rather than application address. Because the address is unaltered on the stack, when Strata reaches the return instruction, the return can be emitted to the fragment cache directly. When the return is executed, the fragment cache address is at the top of the stack, and Strata maintains control. This reduces the dynamic instruction count for handling returns from at least 16 down to 1.

This return optimization relies on the ABI, specifically that the return address is not altered when it is on the stack. However, that restriction is not enforced in hardware, so assembly language programmers sometimes modify return addresses to effect optimizations (e.g., custom sibling-call optimizations). Furthermore there are some libraries, such as C++ exception handling mechanisms, that read return addresses for various purposes, including “walking the stack” to identify the function frames present on the stack. Therefore, in order to insure that the semantics of the program are not changed, we can use this optimization on standard call-returns emitted by the compiler. Using Metaman’s analysis during the compilation phase, we can identify the calls and returns where this optimization can be applied transparently. In the majority of cases for compiled programs, the optimization can be used. In the cases where the calls and returns cannot be determined to be transparent, the original return address is placed on the stack, and the sieve is used for those return instructions, rather than a return instruction, which might result in altering the semantics of the program.

4.1.1 *Exception Handling*

An important special case is exception handling. Most object-oriented languages provide exception handling to allow the program to throw exceptions at arbitrary program points and have execution resume at a catch statement. When an exception is thrown, the run-time system must find the frame on the call stack that handles the exception. To handle exceptions in C++, the exception handler walks the stack, identifies return addresses, and uses those addresses to index into an exception table. If an exception handler is found, stack cleanup code is executed and control is transferred to the exception handler. If the exception table lookup fails, a default handler is invoked.

As discussed, when the return handling optimization is used, fragment cache addresses are placed on the stack instead of return addresses. Figure 5(a) shows the stack at the time of the exception. To identify when an exception occurs, Strata monitors calls to the function that walks the stack (`_Unwind_RaiseException`). When the application calls this function, Strata saves program state and then calls a wrapper function, `_strata_Unwind_RaiseException`. This function uses the `libunwind` library to identify the translated return addresses on the stack and replace them with the actual return addresses, as seen in Figure 5(b). The code cache is then flushed, and Strata begins translating the actual exception handling code, with the optimization temporarily disabled. The cache must be flushed so that while the exception handling code is run, no returns from already translated library code are executed. Strata then identifies the end of the exception handling code by identifying the code sequence emitted by `__builtin_eh_return` and checking the stack location to ensure that it is the end of the error handler, and not cleanup code. Once the end of the error handling is reached Strata flushes the cache again to remove any non-fast return library code. Finally it re-enables fast returns and recreates the fragments pointed to by return addresses below the current stack location, as show in Figure 5(c). Once the stack is correctly remapped, execution continues at the exception handler, under Strata’s control.

4.2 Virtual Function Call Handling

Indirect call instructions provide the basis for virtual functions in compiled object-oriented languages. They allow the program to invoke a function without statically knowing the address of the function. General indirect calls are very powerful constructs, used to implement calls through function pointers, as well as virtual function invocations in object-oriented languages. Statically identifying the target of an indirect call is undecidable; however, the targets of virtual function calls are much more restricted. Virtual function invocation is used to implement polymorphism, where the actual function being invoked can vary depending on the concrete type. Figure 6(a) shows the typical text and heap layout of an object being constructed. Selection of the actual function to invoke is done through a virtual function table (VFT); in this example there are two functions that can be virtually invoked: `func1` and `func2`. These objects are placed into the VFT, which is statically allocated for each concrete type. When the constructor is run on a newly allocated object, an implicit field (`vft`) is stored, which points to the VFT specific to the concrete type of the object. When a virtual function is invoked, the compiler emits code to look up the function address in the VFT, and invokes the function through an indirect call.

The targets of the virtual function call are limited to those in the VFT. Using this fact, Strata can translate VFT entirely, and remap reference to the VFT to point to the translated VFT. To implement this optimization, Strata checks the PC to identify the instructions that load the VFT into the new object. Figure 6(b) shows process of translating the constructor under Strata. When the instruction to load the VFT into the object is encountered, Strata creates a new VFT, `fvft` which contains pointers to trampolines that will jump back to Strata and translate the target function. Strata then translates the `mov` instruction to load the newly created `fvft` instead of the original. When the indirect call is encountered, it is unnecessary to emit a sieve, but instead a simple push an jump combination can replace the call:

```
0x2058: push 0x3e38   ; return addr
        ; call becomes a jump
0x205d: jmp DWORD PTR [eax+8]
```

The VFT metadata collected by Metaman, described in Section 3 allow Strata to implement this performance enhancement. For programs that make heavy use of virtual functions, such as the highly object-oriented `Xalan`, reduc-

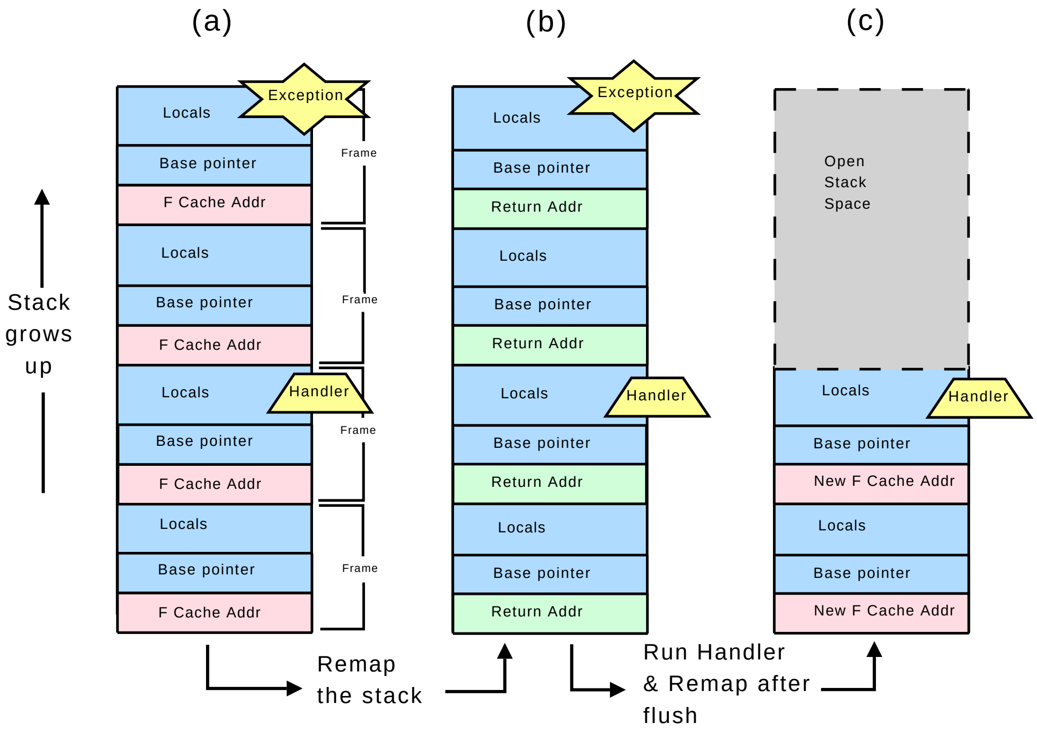

Figure 5: Stack view of exception handling

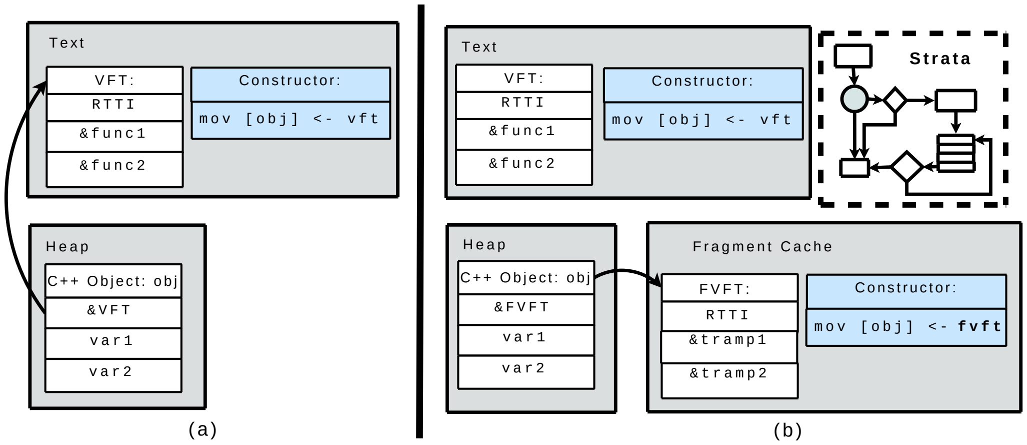

Figure 6: (a) Normal VFT initialization. (b) VFT initialization with Strata translation

ing the instruction count from at least 16 to 2 results in a significant performance gain.

5. RESULTS

A prototype of the Metaman system has been implemented on x86/Linux, to test the effectiveness of the metadata collection approach, using the GNU software toolchain and the extensible build system SCons [11] to integrate Metaman into the build system. All the experiments described in this section were performed on a 2.8GHz dual Pentium 4 system running Debian Linux, with 1GB of RAM. Application code was compiled with `gcc` version 3.3.5, using the `-O2` optimization flag.

The SDT branch handling optimizations were tested on the C and C++ SPEC 2006 benchmark suite [5], as well as a microbenchmarks designed to expose the overhead virtual function calls. The microbenchmark consisted of a tight loop over a virtually invoked function. The VFT handling optimization reduced overhead of the benchmark from 1.76x to 1.25x.

Figure 7 shows the performance on the SPEC benchmark suite. The graphs compare performance of the SDT system by normalizing to native execution speed (i.e., no SDT system). Thus, bars below 1 indicate a speed up, while bars above 1 indicate a slowdown. The first six data sets are the C++ benchmarks. We also included the C benchmarks, which do not benefit from the VFT optimization, but do see improvements from the return handling optimization. The first bar shows the baseline performance of Strata running with the sieve for indirect branch handling. The second bar shows the performance of the return handling mechanism alone. The final bar shows the additional effects of the VFT optimization. `Xalan`, which is an XML parser and processor, with many virtual function invocations, shows a 15% improvement with the VFT optimization. The final group shows the arithmetic average of the SPEC benchmarks. With all optimizations enabled, the average execution speed is within 3% of native speed.

6. RELATED WORK

There is a significant amount of research into SDT systems and indirect branch handling. Most of this research focuses on pure run-time solutions. More recently, integrated solutions have been examined. JikesRVM is a JVM implementing runtime optimizations, many of which rely on the information rich Java bytecode [1]. Having such a robust run-time/JIT environment as is required by Java increases the access to program metadata. However, JikesRVM does not offer a general facility to store and retrieve metadata, and all tools making use of the data must strictly be written within the JikesRVM framework. Similarly, Montana and Oberon provide frameworks for building tools and sharing information within the software development toolchain [18, 10]. In both cases, plug-ins must be written specifically to the API of the system. Metaman differs from these systems by being toolchain neutral as much as possible, using XML as a generalized data exchange medium.

Xu et al. presented work that used metadata to improve register allocation within a dynamic binary translator [20]. The metadata that identified where the register allocator needed to spill registers was stored in an ELF section. The binary translator used this information to eliminate the spills on the target ISA (IA64), which had more architectural registers than the source ISA (IA32). The LENS project is designed to give system writers a method of naming and understanding information as it is compiled and optimized [14]. Our goal is to supply and map both user-level and binary-level information that is maintained and usable across the toolchain.

7. CONCLUSIONS

This paper has described a system for collecting and sharing program metadata among the components of a modern software development toolchain, specifically for supporting an object-oriented environment. The paper describes how Metaman, a generalizable system for collecting program metadata, was employed to improve the efficiency of Strata, a production-quality SDT system. The key insight is that Metaman provided information restricting the targets of indirect branches so Strata could retranslate those structures completely, thus avoiding the high-overhead hash-table based lookup mechanism. By collecting data about the use of virtual functions and call/returns in the presence of exceptions, the runtime overhead of Strata was reduced by 22% making mediated execution comparable to native execution (within 3%).

Future directions include expanding the scope of data collected by Metaman, and enabling more tools to be Metaman aware. As more tools are able to both query and submit metadata to Metaman, software improvements will be enabled beyond SDT to include tools throughout the software development toolchain.

8. REFERENCES

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, New York, NY, USA, 2000. ACM Press.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 1–12, New York, NY, USA, 2000. ACM Press.

[3] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* PhD thesis, Massachusetts Institute of Technology, 2004.

[4] K. Ebcioglu and E. R. Altman. Daisy: dynamic compilation for 100% architectural compatibility. In *ISCA '97: Proceedings of the 24th annual International Symposium on Computer Architecture*, 1997.

[5] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[6] J. Hiser, D. Williams, A. Filipi, B. Childers, and J. Davidson. Evaluating fragment construction policies for SDT systems. In *VEE'06: Second International Conference on Virtual Execution Environments*, pages 122–131, New York, NY, USA, 2006. ACM Press.

[7] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating indirect

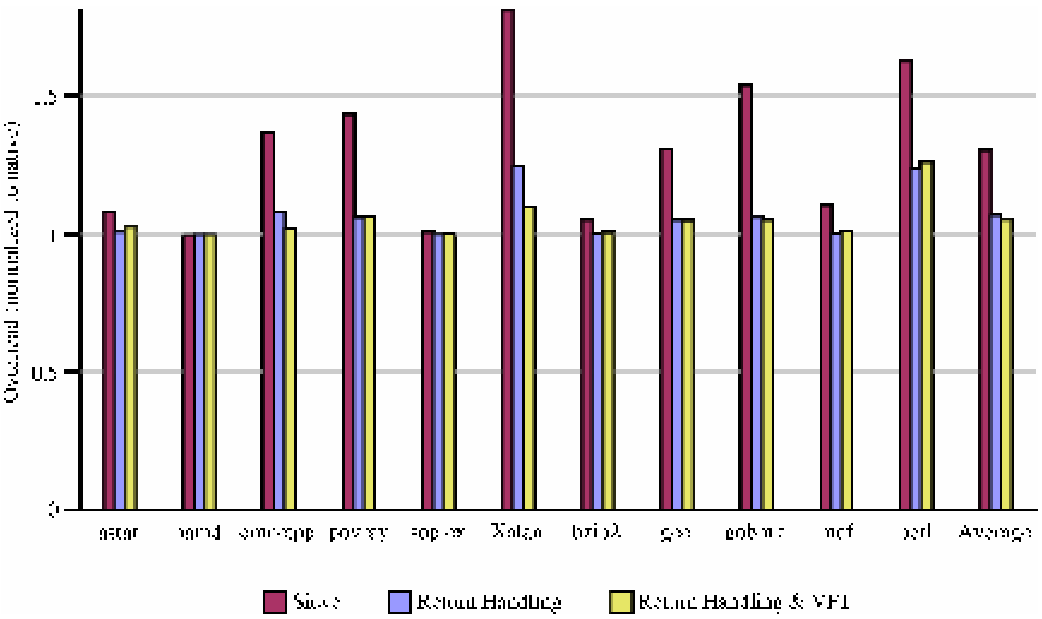

Figure 7: SPEC2006 performance for virtual function tables

branch handling mechanisms in software dynamic translation systems. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 61–73, Washington, DC, USA, 2007. IEEE Computer Society.

[8] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *VEE'06: Second International Conference on Virtual Execution Environments*, pages 2–12, New York, NY, USA, 2006. ACM Press.

[9] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[10] T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Trans. Program. Lang. Syst.*, 25(4):500–548, 2003.

[11] S. Knight. SCons design and implementation. In *Tenth International Python Conference*, 2002.

[12] N. Kumar, J. Misurda, B. R. Childers, and M. L. Soffa. Instrumentation in software dynamic translators for self-managed systems. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 90–94, New York, NY, USA, 2004. ACM Press.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2005.

[14] M. O. McCracken. The design and implementation of the lens program information framework. Technical report, UCSD CSE, 2006.

[15] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*, pages 209–218, 2002.

[16] K. Scott, N. Kumar, B. R. Childers, J. W. Davidson, and M. L. Soffa. Overhead reduction techniques for software dynamic translation. In *IPDPS Next Generation Software Program - NSFNGS - PI Workshop*, 2004.

[17] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, 2003.

[18] D. Soroker, M. Karasick, J. Barton, and D. Streeter. Extension mechanisms in montana. In *ICCSSE '97: Proceedings of the 8th Israeli Conference on Computer-Based Systems and Software Engineering*, page 119, Washington, DC, USA, 1997. IEEE Computer Society.

[19] S. Sridhar, J. S. Shapiro, E. Northup, and P. P. Bungale. Hdtrans: an open source, low-level dynamic instrumentation system. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 175–185, New York, NY, USA, 2006. ACM.

[20] C. Xu, J. Li, T. Bao, Y. Wang, and B. Huang. Metadata driven memory optimizations in dynamic binary translator. In *VEE '07: Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 148–157, New York, NY, USA, 2007. ACM.