Analysis of Virtual Method Invocation for Binary Translation

Jens Tröger
Faculty of Information Technology
Queensland University of Technology
Brisbane, QLD 4001
Australia
j.troeger@qut.edu.au

Cristina Cifuentes
Sun Microsystems Laboratories
15 Network Drive
Menlo Park, CA 94025
U.S.A.
cristina.cifuentes@sun.com

Abstract

The University of Queensland Binary Translator (UQBT) is a static binary translation framework that allows for the translation of binary, executable programs, from one architecture to another one. Using different specification files, the UQBT can be easily tailored for either a new source or target architecture. UQBT employs sophisticated analyses in order to abstract the binary code from the features of the source architecture.

In static binary translation, it is not always possible to decode and translate all paths of a program statically, as instructions like computed jumps and indexed calls rely on runtime values. An interpreter is commonly used at runtime to translate such pieces of code, at the expense of runtime.

In this paper we present a technique to statically recover the code of virtual method dispatchers, which enhances the abstraction from the binary code. We also show how the statically recovered information is then used by a dynamic environment to analyse virtual tables in order to increase the code coverage for the binary translator. The technique has been tested on SPARC(R) and Pentium machines using binaries of different C++ compilers, and preliminary results are presented.

1 Introduction

Static binary translation is the process of translating the object code of a program from a source architecture to a target architecture. The term static here means that the program is analysed and translated without executing the program code.

The UQBT approach to binary translation is to apply transformations to the machine code in order to transform it into a high-level intermediate representation. This representation abstracts from source machine details such as assembly instructions and calling conventions. The higher the level of abstraction, the easier it is to generate tailored native code for the target architecture, as source machine dependent peculiarities and idioms are no longer present. UQBT employs sophisticated analyses in order to abstract the code from the features of the source architecture. Analyses for switch statement and switch table recovery, removal of delayed branches and condition codes, translation of stack-based floating point to register-based code, and recovery of (low-level) parameters passed to functions is performed. UQBT translators for the SPARC, x86 and other architectures have traditionally been tested with conventional procedural programs[4].

Object-oriented languages provide the programmer with a lot of convenience, for example through features such as class inheritance, overloading/overriding of methods and virtual method invocation. These features make the design and implementation of software more comfortable, but they require advanced compilation techniques. Code generation for object-oriented languages is in some cases not trivial. For example, a virtual method call is implemented as an entire sequence of machine instructions that computes the destination of the call depending on the runtime type of an object pointer. This sequence of instructions may or may not have other inter-leaving instructions that perform some other task, or may even reuse the outcome of previous computations. For the purposes of binary translation, understanding and recovery of this type of code is challenging. We are not aware of any translator that incorporates analyses to recover virtual method calls.

In this paper we present a technique that is able to statically recover a virtual method call from a sequence of instructions. This analysis allows us to replace a procedural CALL instruction by a more specialized VINVOKE (virtual invoke) instruction, and thereby improving the level of abstraction from the native binary code. The statically recovered information supports type analysis to that effect.
that a given value can be identified as a pointer rather than a simple integer value. Further, the recovered information indicates the object model used by the compiler, and we use this information to perform analysis of virtual tables at runtime.

The paper is organized in the following way. Section 2 briefly introduces the UQBT framework and its intermediate language for representing binaries in an abstract and portable form. Section 3 recapitulates the concepts of class inheritance and virtual method calls. Section 4 shows our approach of abstracting instruction sequences that implement a virtual method call into a single virtual call instruction, and section 5 describes the usage of the recovered information in a dynamic environment. Section 6 shows our initial experimental results, and last, Section 7 gives a brief overview of related work.

2 Static Analysis with the UQBT Framework

The UQBT framework [3, 4] is a static, retargetable binary translation framework developed at the University of Queensland, Australia. Figure 1 illustrates the architecture of the UQBT framework, that translates binaries for a source machine \( M_S \) onto binaries for a target machine \( M_T \). In the framework, retargetability is supported by means of specification languages for syntax (SLED) and semantics (SSL) of machine instructions, as well as for calling conventions and procedural information (PAL); APIs for the binary file format (BFF) and to handle control flow (CTL); and plugable modules for machine specific analysis. Further, two intermediate representation are used: a machine-dependent register transfer representation (RTL) and a high-level, machine-independent representation (HRTL).

The UQBT framework generates target code using a variety of backends. The initial implementation of UQBT relied on a C compiler to be used as an optimizer and code generator for the target machine. In this approach, the intermediate language of UQBT, HRTL, was translated into low-level C code, in essence using the C compiler as a macro assembler. Later versions of UQBT have made use of backends that use public domain or proprietary optimizers, integrating at the RTL\(^1\) level.

At present, UQBT performs an analysis to recover switch statements that were implemented as jump tables [5], but does not perform any analysis on indirect register calls. Therefore, UQBT relies on a runtime interpreter to translate any pieces of code that were not translated statically. This is a standard technique used in static binary translators as it is not always possible to distinguish code from data.

HRTL and the Intermediate Representation

The HRTL (High-level Register Transfer Language) intermediate language is used by the UQBT framework to represent a binary program in an abstract and portable way. HRTL provides locations in the form of virtual registers, variables and memory. The variables are variables with basic types recovered through the analysis. HRTL statements resemble those of a procedural, third-generation language:

- assignment of expression values to locations,
- conditional jump to a location,
- unconditional jump to a location,
- switch/n-way jump to n locations,
- call to a procedure (location), with parameters associated to the call site, and
- return from a procedure.

UQBT creates a Control Flow Graph (CFG) for each procedure of the source binary that it is able to find. The nodes of the CFG are representations of the basic blocks in the source binary, and contain a sequence of HRTL instructions that hold the semantics of the basic block. Call graphs are also built, as per standard compilers.

Figure 2 shows C++, assembly and HRTL language code for a virtual method call dispatcher from a SPARC(R) architecture program. The HRTL basic block contains abstract registers \( r[.\] \) and abstract memory references \( m[.\] \), as well as the function \( \text{sgnex}() \), that sign-extends a given value (its third parameter) from 16 to 32 bits. Note that the SPARC architecture has delayed instruction semantics for every branching instruction, i.e. the \( \text{add} \) instruction following the \( \text{call} \) instruction is executed before the target address at the location pointed to by register \%r2 is reached. In the HRTL code, this delayed semantics has been transformed away into a standard sequential approach.

3 Class Inheritance and Virtual Method Calls

This section reviews the concepts of class inheritance, the way compilers generate code for virtual method calls, and it illustrates the different implementations of objects using diagrams. This section is intended to help the reader

\(^1\)Register Transfer Lists, a common intermediate representation of code
C++ source code:
```
    obj->foo();
```

SPARC architecture assembly code:
```
ld [%l0+8], %o1
ldsh [%o1+8], %o0
ld [%o1+12], %o2
call %o2
add %l0, %o0, %o0
```

HRTL code:
```
Computed call BB (0x488228):
0001089c *32* r[8] := sgnex(16,32,m[r[9] + 8])
000108a0 *32* r[10] := m[r[9] + 12]
000108a4 CALL r[10]
```

Figure 2: SPARC architecture assembly and HRTL representation of a method dispatcher for foo()
For replicated multiple inheritance, the instances of B and C each contain their own instance of class A. On the other hand, shared multiple inheritance is specified using the C++ keyword virtual:

```cpp
class A { ... };
class B: public virtual A { ... };
class C: public virtual A { ... };
class D: public B, public C { ... };
```

Instances of class B and class C share the same instance of class A. With an object of type D, this sharing is implemented as a pointer to the common A instance.

**Method overloading** is another concept of object-oriented languages. For example, if a class B specifies a method with the same signature as a method in a super class A, then a call to this method depends on the compile time type of the object pointer:

```cpp
B* b = new B();
b->foo(); // calls B::foo()
((A*) b)->foo(); // calls A::foo()
```

Sometimes it is necessary to call the method B::foo(), no matter of what compile time type the object pointer is. This is done by specifying foo() as a virtual method, and the compiler generates code that looks up the correct method when the method is invoked at runtime. This special piece of lookup code is called a method dispatcher. If a class defines virtual methods, then instances of that class (and instances of all subclasses) contain a pointer to a virtual table for this particular class. This table holds the addresses of the correct virtual methods for the class and delta values that are used to correct the first parameter of the method call. This first parameter is implicitly added by the compiler and is a reference to the object that the method is invoked on. In the example above, the object pointer b is cast to type A*, but the virtual method B::foo() is called that expects a pointer to an object of type B rather than type A. Therefore, the pointer has to be corrected by adding a negative value (the delta value), that corrects the object pointer from A* to B* (refer to Figure 3.1).

Virtual method invocations on objects follow the same pattern, but they slightly differ in their implementation depending on the compile time type of the object pointer and the object model used by the compiler. Generally speaking, a virtual method call works according to the following steps (see the example code in Figure 2):

1. If the method call is performed on a type cast object pointer, then the correct sub-object is obtained first, i.e. the view to the object is modified,
2. given the correct view to an object, the address of the virtual table is fetched from the object, and
3. given the virtual table, the address of the virtual method is fetched from the table, as well as the delta value to correct the view to the object for the call.

### 4 Static Recovery of Virtual Method Calls

The techniques we use in our approach are similar to the techniques used by Cifuentes and Van Emmerik [5] to recover table-based switch statements. In their work, the aim was to recover the possible target addresses of a switch statement at an unconditional jump site. In our work, we are interested in recovering a virtual method invocation from an instruction sequence that may or may not

---

2 Assuming that foo() is declared to be virtual
implement a method dispatcher. Note that we do not identify possible destination addresses with this technique, as they can only be determined at runtime when a method is invoked on an actual object. Nevertheless, the recovered information not only improves the level of abstraction, but also supports type analysis. Further, indication about the layout of objects and virtual tables is obtained from our analysis, and is exploited at runtime by our HRTL interpreter in order to analyse virtual tables.

Our technique works as follows. Given an arbitrary basic block that ends on a called computation instruction, backward slicing[15] is applied to the basic block. Thus, those instructions that compute the target of the call instruction are extracted into a slice. Given this slice, copy propagation combined with simplification generates the call expression. Copy propagation starts with the call expression of the CALL instruction at the end of the slice (this is \( r[10] \) in Figure 2). It then replaces all occurring locations (only \( r[10] \) in the example) with those expressions that are assigned to the locations in the previous assignment instruction (here \( r[10] \) is replaced by \( m[r[9] + 12] \)). The resulting intermediate expression is then simplified, and the process is repeated until the beginning of the slice is reached. The simplification of expressions is trivial – constant folding is applied and the expressions are rearranged in such a way that constants go to the right hand side of operations, and locations to the left hand side. Further more, every expression is matched against a small set of common idioms:

- a location, that is assigned a value XOR'ed with itself, can instead be assigned the value zero,
- bit-wise AND'ing a value with \(-1^3\) can be replaced by the value itself, and
- the expression \((zfill(16,32,EXPR) * 65536) \gg A 16\) can be replaced by \(sgnex(16,32,EXPR)\) (this turned out to be a typical idiom on SPARC architectures)\(^4\).

With the same technique, an expression for the first parameter is derived from the basic block. Even though we currently analyse only single basic blocks, our technique still identifies most of the virtual method calls, as illustrated in Section 6.

\(^3\) Since \(-1\) in two's complement is a string of 1s.

\(^4\) \(zfill()\) here takes a 16-bit value and creates a 32-bit value in which the upper 16 bits are filled with zeros, \(\gg A\) denotes an arithmetic bit-wise right-shift, and \(sgnex()\) sign-extends the 16-bit value to a 32-bit value.

### 4.1 Normal Forms

Using slicing techniques, copy propagation and simplification, a call expression and a first-parameter expression can be derived from a basic block. These two expressions must match a particular pattern and meet certain conditions in order to be recognized as a virtual method call.

The compile-time type of an object pointer and the object model dictate the instruction sequence of the dispatcher code. Even though the output of different compilers on different platforms was analysed, we found similarities that led to the derivation of three sets of normal forms.

The first set contains three normal forms that describe the retrieval of the correct virtual table from a given object pointer:

**VTBL 1:** \(vtbl = vtoffset[\text{obj}]\)

The object pointer is not cast, and the address of the virtual table is fetched from a fixed offset. For example a simple call like \(\text{obj}->\text{foo()}\) creates this form of dispatcher code.

**VTBL 2:** \(vtbl = vtoffset[\text{obj} + \text{view}]\) For a cast object pointer the view constant is added to the object pointer first to modify the view to the object according to the cast. Then the address of the vtable is fetched. In Figure 3 this is illustrated by casting the object pointer to \(A^*\) in the first and the second picture. A method call \(((A^*) \text{obj})->\text{foo()}\) for Figure 3.1 or \(((A^*) ((C^*) \text{obj}))->\text{foo()}\) for Figure 3.2 relate to this normal form. The cast to \(C^*\) and then to \(A^*\) are merged into one single view constant, since the compiler is able to determine both view values at compile time.

**VTBL 3:** \(vtbl = vtoffset[[\text{obj} + \text{view}]\]

This normal form relates to an object that has type definition makes use of shared multiple inheritance and where the cast of the object pointer happens to be a cast to a shared instance. Again, before the address of the virtual table is fetched from the object the view to the object needs to be modified. The view to the shared instance is obtained by dereferencing the pointer. Figure 3.3 depicts the additional indirection to the shared instance and \(((A^*) ((B^*) \text{obj}))->\text{foo()}\) serves as an illustration.

When the virtual table is obtained from an object, the address of the virtual method is fetched from the table as well as the delta value to correct the view to the object for the call site. The second set contains two normal forms to obtain the address of the virtual method from the virtual table:
VMTH 1: offset[vtbl] The address of the target method can be obtained from the virtual table from a fixed offset. The same way the delta value is picked from a different offset.

VMTH 2: offset[vtbl + row] In this case the correct row in the table is selected first, and then the address of the virtual method and the delta value are retrieved from their assigned columns.

To compute the value of the first parameter (the this pointer) that is implicitly passed to any method call, we found two possible normal forms for the third set:

PARM 1: obj' This passes the modified view to an object as a first parameter. The destination address of the call that is stored in the virtual table is not the address of the virtual method, but trampoline code, that corrects the view to the object for the virtual method and then jumps to the actual method address.

PARM 2: obj + f(obj) Here, the first parameter is corrected by adding the appropriate delta value from the virtual table to the object pointer. The function f takes an object pointer and obtains the correct delta value from the virtual table using a combination of the normal forms mentioned above.

4.2 Expression Matching

The call expression and the first-parameter expression that are derived from a basic block must meet the following conditions in order to be “matched” as a virtual method call:

- The call expression must match a combination of the normal forms to retrieve the virtual table from an object (either VTBL1, VTBL2 or VTBL3), and to obtain the correct method address from the virtual table (VMTH1 or VMTH2),

- the first-parameter expression must match either one of the normal forms for the first parameter PARM1 or PARM2, and

- for both call and the first-parameter expressions obj must be stored at the same location.

Therefore, a computed call basic block is identified as the implementation of a virtual method call dispatcher, iff there exist two expressions (that are obtained using slicing, copy propagation and simplification) that match a valid combination of the normal forms mentioned above, and iff those two expressions use the same location for their respective object pointer.

4.3 Type Recovery

If a basic block implements a virtual method call dispatcher, then the information recovered during our analysis also implies the types of the values. Apart from the fact that the call value must be an Address type, the recovered type information is as follows (refer to the dispatcher code in figure 2):

- The first parameter value is a pointer to the object. Thus, the value in register r[8] (at 000108a4) is of type Address.

- The value in register r[16] is a pointer to an object, register r[9] then holds a pointer to the virtual table of the object. Therefore, both values are of type Address.

- The delta value is an integer number – thus register r[8] (at 0001089c) holds a value of type Integer.

The recovered type information is useful to improve the quality of the intermediate code, and can be used to support or to verify further type analysis on the binary or the intermediate representation.

4.4 An Example

We illustrate our technique by analysing the HRTL representation of the basic block in Figure 2. The basic block consists of several assignment instructions and the CALL instruction at the end of the basic block. The destination address of the CALL can be found at runtime in register r[10], and the only parameter location is variable r[8].

The slice that computes the target address of the call in r[10] is

\[
\begin{align*}
\end{align*}
\]

and the slice for the first parameter is

\[
\begin{align*}
*32* r[8] & := sgnex(16,32,m[r[9] + 8]) \\
\end{align*}
\]

Copy propagation and simplification is applied to both slices – to the call slice to create the call expression, and to the first-parameter slice to create the first-parameter expression. The output of our tool that implements our technique follows:

```
call m[m[r[16] + 8] + 12]
parm sgnex(16,32,m[m[r[16] + 8] + 8]) + r[16]
```
Both expressions are now matched against the different normal forms. The call expression matches normal form VTBL 1 to retrieve the address of the virtual table from the object (vtbl = m[r[16] + 8]), and normal form VMTH 1 to obtain the address of the target method from the virtual table (m[vtbl + 12]). In this example, the first-parameter expression matches normal form PARM 2: r[16] + f(r[16]) and f obtains the virtual table from the object and sign-extends the delta value from 16 to 32 bits. Finally, both expressions use register r[16] that holds the object pointer at runtime.

Since all of our criteria for a virtual method dispatcher are met by the given basic block, the CALL instruction at the end of the basic block is replaced by a VINVOKE instruction that holds the recovered information:

\[
\text{callnf: 1 obj: r[16] sharedoffset: -1 vtoffset: 8 vmethoffset: 12}
\]

\[
\text{paramnf: 2 deltaoffset: 8}
\]

5 Analysis of Virtual Tables at Runtime

The static analysis as shown in the previous section recovers the implementation of virtual method dispatchers, and therewith improves the abstraction from the source binary code. To increase the code coverage by translating the targets of computed branch instructions, dynamic feedback is required. And because objects are created only at runtime, our analysis of virtual tables can only take place at runtime.

In general, the advantage of analysing virtual tables is the chance of discovering new code fragments, which would have been missed by a purely static approach. Obviously, this analysis is only useful if the code was compiled from an object-oriented language.

5.1 Implementation and Techniques

The statically recovered information can be used to support the analysis of virtual tables at runtime. An interpreter for UQBT’s intermediate language HRTL was formally defined and implemented amongst other things for this purpose [16]. Please note that our intermediate code interpreter is different from the fall-back interpreter for native code that is used to handle statically untranslated code at runtime.

Our HRTL interpreter implements a fetch and execute cycle, and interprets one HRTL instruction after the other. By doing so, it continuously changes the state of the underlying abstract machine and, thus, executes the program as if it would run on a real machine. Once the interpreter encounters a VINVOKE instruction, it triggers the following actions one after the other:

1. given a valid object pointer, the address of the virtual table is obtained, and the virtual table is analyzed,
2. for every entry in the virtual table that contains the address of a virtual method, the UQBT static translator is invoked to translate all reachable code into its intermediate form, and
3. the target method is then called and interpreted like any other procedure.

In the following, the first two steps will be explained in more detail. For more details on step three please refer to [16].

5.1.1 Obtaining the Virtual Table

It is known from our static analysis of virtual method dispatchers where the address of one virtual table is stored in the object. Obtaining the address of the virtual table is trivial. In our example in section 4, the address was stored at offset 8 within the object. Therefore given the content of the register that holds the address of an object at runtime, say r[16] = 0x08001200, the address of the virtual table for all instances of this class can be fetched from 0x08001208. Further analysis is performed with this particular table as shown in the following section.

5.1.2 Dynamic Analysis of Virtual Tables

During the examination of binary code that was generated by different compilers on different platforms, we came across three different layouts for virtual tables. Note that the virtual table analysis is performed without any type information and only at the binary level. In contrast, the rules to lay out virtual tables by a compiler which has all the type information at hand are much more complex and not very helpful for binary translation purposes [14].

From the binary point of view, a virtual table consists of several entries, grouped into rows. Figure 4 shows three different types of rows\(^5\). As can be seen, every row contains at least the delta value and the address of the virtual method. A special case is shown in Figure 4.3, where the address of the code for dynamic casting (RTTI\(^6\) in C++) is stored in each row, whereas the other table layouts used separate rows for the RTTI functions.

Usually, a delta value is either zero or a negative integer, and a valid entry for a virtual method is an address in the virtual table for all instances of this class can be fetched from 0x08001208. Further analysis is performed with this particular table as shown in the following section.

5.1.2 Dynamic Analysis of Virtual Tables

During the examination of binary code that was generated by different compilers on different platforms, we came across three different layouts for virtual tables. Note that the virtual table analysis is performed without any type information and only at the binary level. In contrast, the rules to lay out virtual tables by a compiler which has all the type information at hand are much more complex and not very helpful for binary translation purposes [14].

From the binary point of view, a virtual table consists of several entries, grouped into rows. Figure 4 shows three different types of rows\(^5\). As can be seen, every row contains at least the delta value and the address of the virtual method. A special case is shown in Figure 4.3, where the address of the code for dynamic casting (RTTI\(^6\) in C++) is stored in each row, whereas the other table layouts used separate rows for the RTTI functions.

Usually, a delta value is either zero or a negative integer, and a valid entry for a virtual method is an address in the virtual table for all instances of this class can be fetched from 0x08001208. Further analysis is performed with this particular table as shown in the following section.

\(^5\) The dynamic analysis assumes a word size of 32 bits

\(^6\) Runtime Type Information, the C++ implementation for dynamic casting.
Naive Analysis

The static analysis of section 4 provides information at which offset the address of the virtual method is stored in the table. Thus, in the naive approach every 32-bit wide entry from the beginning of the virtual table up to the given offset is tested whether or not it is a valid virtual method entry. If so, then the UQBT static translator is called to translate the code at this address, including all statically reachable procedures.

The higher the offset of the virtual method entry, the better chances are to discover more code because more rows are scanned until the target entry is reached. The worst case here is that the offset is the first row in the virtual table, and therefore only one, i.e. the target virtual method is decoded.

Recovering the Type of Virtual Tables

In our second and more complex approach, the type of the virtual table is determined first. This is done by comparing the first 2 or 3 words against our three possible table layouts (refer to Figure 4). If none of the table types can be identified, then our type analysis falls back to the naive approach.

A virtual table is identified to be of one specific type if its first 2 or 3 words are either a valid 16- or 32-bit wide delta value, or the address of a virtual method. For example, in order to identify a table of type 1, the following conditions must hold:

- the first 32 bits must be either a 32-bit wide delta value, or the upper 16 bits contain a delta value and the lower 16 bits are set to zero,
- following the delta value, a 32 bit wide address in the code section must follow.

Once the type of the virtual table is determined, row after row is fetched and the addresses of the virtual methods are handed to the UQBT static translator. Every row is also checked for validity. If the first invalid row is encountered then the HRTL interpreter stops its analysis, invokes the target method and continues execution.

6 Preliminary Results

We have experimented with several C++ compilers for different machines (in paper and in practice) and we report on our preliminary results of implementing our technique as part of the UQBT framework. Binaries were generated by different versions of the following C++ compilers: gcc, cc and icc, on Pentium and SPARC architectures.

Figure 5 shows our results for the static recovery of virtual method dispatchers with a small set of object-oriented benchmarks. The table gives the ratio between the calls that are identified as a method dispatcher out of the number of computed calls in the intermediate representation\(^7\). The static size in bytes of the binaries is denoted in parenthesis. A “_” denotes that UQBT was not able to decode the binary successfully. The first three programs, single.cpp, shared.cpp and replicated.cpp, contain the definition of an abstract data type using the different types of inheritance. In these programs, an object is instantiated and calls to virtual methods using all possible cast object pointers are performed. The other three programs are standard object-oriented benchmark programs. Richards.cpp emulates a task dispatcher, DeltaBlue.cpp is an incremental constraint solver, and OOPack.cpp is a benchmark that tests the code quality of a C/C++ compiler.

There are various reasons that our current approach does not recover all virtual method dispatchers. The most common reason is that intermediate results (for example the address of a virtual table) are temporarily stored and reused in immediately subsequent method invocations on the same object. In these cases, the condition that the call expression and the first-parameter expression must use the same

\(^7\)The intermediate representation of the Richards benchmark for SPARC architectures was created by speculatively decoding code, whereas the Pentium version was not, therefore more code is statically decoded.
location is not met. Some compilers in higher optimization mode are able to determine the position in the virtual table where the address of the virtual method is stored. In those cases, no dispatcher code is generated at all.

Figure 6 shows the increase of the code coverage using our dynamic virtual table analysis compared to pure static binary translation. For the time being, those results are sparse due to several reasons. First of all, UQBT sometimes fails to translate larger OO programs or code that does not conform to standard calling conventions, and therefore our interpreter is not incorporated at all. The star indicates that the HRTL interpreter bailed out because the UQBT static translator produced an erroneous translation\(^8\). Secondly, the interpreter is in an experimental development stage and currently has difficulties to setup a proper program environment and to execute native calls with side effects correctly. As can be seen, the dynamic analysis of virtual tables increases the code coverage. The runtime overhead that is added to the HRTL interpreter obviously depends on two things:

- it is linear in the number of code entries found in the virtual table, and
- it is linear in the code size of the target procedure, including all statically reachable code.

7 Related Work and Summary

Static binary translators that have been reported in the literature have not addressed the issue of analysing virtual method dispatchers. In all static translation frameworks, emulation of these features is done at runtime. This is the standard technique used by translators such as Digital’s VEST and mx [11], Tandem’s translator [1] and the Dixie toolset [8].

In the UQBT framework, Cifuentes and Van Emmerik [5] described a method to analyse native machine code in order to recognize table-based switch statements. Although similar techniques (slicing and copy propagation) are used in our approach to analyze the code, and no simplification is performed, the aim of the analyses is different. Whereas Cifuentes and Van Emmerik focused on the recovery of branch tables for switch statements in order to discover yet untranslated code, our approach aims to improve the level of abstraction and to obtain information for further analysis in order to increase the code coverage using runtime feedback. In contrast to our approach, the authors analyze native code rather than its intermediate representation.

Dixie [8] is not just a static binary translation tool, but an entire toolset. It consists of the Dixie compiler that translates binary code (Alpha, Convex, PPC, x86, ...) into Dixie ISA code, Jango that instruments Dixie code for evaluation and profiling, Speedy to optimize the code and, finally, the Dixie Virtual Machine (DVM), that actually runs Dixie code. The Dixie virtual ISA is a 64-bit load-store architecture, that offers a single register file with 32768 64-bit registers and 128-bits wide instruction words, support for vector execution to facilitate vector instructions and the usual machine data types of different sizes. Although Dixie is a static translator, no extra analysis on the intermediate representation is performed.

In this paper we presented a technique to recover and abstract from the implementation of virtual method dis-

\(^8\)So-called \textit{trampoline code} does not conform to the calling conventions of the e.g. Pentium architecture, and therefore UQBT cannot identify the parameters properly. Thus, the HRTL interpreter raises an exception due to wrong variable usage.
patches. The static analysis is performed on the intermediate representation of a source binary. As a result, an ordinary CALL instruction could be replaced by a more sophisticated VINVOKE instruction. The statically recovered information is then employed at runtime to enable the analysis of virtual tables. The primary goals are to increase the code coverage for the UQBT binary translator, and to enhance the abstraction of the intermediate representation from the native OO source binary code.

References


<table>
<thead>
<tr>
<th></th>
<th>static analysis</th>
<th>dynamic vtab analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>code translated</td>
<td>procs</td>
</tr>
<tr>
<td>single.cpp -O0:</td>
<td>211 bytes</td>
<td>3</td>
</tr>
<tr>
<td>-O2:</td>
<td>75</td>
<td>1</td>
</tr>
<tr>
<td>shared.cpp -O0:</td>
<td>714 bytes</td>
<td>6</td>
</tr>
<tr>
<td>-O2:</td>
<td>226</td>
<td>1</td>
</tr>
<tr>
<td>replicated.cpp -O0:</td>
<td>437 bytes</td>
<td>5</td>
</tr>
<tr>
<td>-O2:</td>
<td>183</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 6: Code coverage for gcc (i386/Linux) compiled binaries