

Approaches for Universal Static Binary Translation

Marc Angelone
Masters Thesis

Technical Report DU-CS-06-02
Department of Computer Science
Drexel University
Philadelphia, PA 19104
March, 2006

Approaches for Universal Static Binary Translation

Marc Angelone

Master's Thesis

Dept. of Computer Science

Drexel University

mja33@drexel.edu

March 18, 2006

Abstract

Binary translation is the process of converting machine code created for one architecture to semantically equivalent machine code for another architecture. Static binary translation in particular is one of many ways to achieve Architecture-Independent Computing (AIC). AIC aims to provide the ability to execute code on any machine regardless of the original target. Unlike other solutions for AIC, such as portable virtual machines, emulators, or interpreters, static binary translation presents the possibility to create a framework that can translate code between any two machine code languages and provide results at near native speeds.

In this thesis we present the Binary Translation System (BTS), a new static binary translator that can retarget machine code for arbitrary computing architectures. This thesis presents the extensible BTS framework that can be used to create translators for any common instruction set architecture. We also present our architecture-independent code representation and manipulations that we used to face some of the common problems with binary translation.

Our prototype BTS implements a RISC-like, architecture-independent code representation, a PowerPC decoder, and a SPARC encoder to demonstrate the feasibility and problems of static binary translation. The PowerPC decoder in particular, exemplifies many decompilation problems that must be dealt with when converting machine code to an architecture-independent code representation. Our test results show that our prototype can achieve comparable performance and compatibility to other AIC solutions. Our methods and approaches presented in this thesis may be of interest to binary translation writers, reverse engineers, decompiler/compiler designers, and engineers wanting to do binary program manipulation or instrumentation.

1 Introduction

Constantly developers are making considerations for the hardware and run-time they are developing for, and in some cases these considerations may render the source code to be hardware dependent. Classic examples of this are: byte-swapping between big and little endianness, memory alignment for data structures, or in-line assembly. There also exists similar problems with the use of software. Intellectual property protection or limited product lifetime can bar users to source code or updates, effectively preventing any cross-architecture compatibility. Additionally, legacy software that needs updating to newer platforms many times cannot be, either because the code is too complex, missing or incompatible with modern compilers.

What do all these issues have in common? They all stem from the fact that competition and advances in computer hardware can make big differences in the types of interfaces that are available to software. These issues and the current state of computer technology then beg for a solution. A solution that will allow the developers to make specialized decisions without adversely affecting the users choice of hardware. Such a transformation solution can be simplified as enabling Architecture-Independent Computing (AIC)[11]. AIC would allow a program to effectively run correctly on any hardware/run-time environment.

AIC and software portability is still an on-going effort in the computer science field, with many potential solutions that range from software engineering methodologies to new programming languages. Another common solution to the software portability problem is special run-time environments that provide emulation or dynamic translation, converting software for hardware discrepancies on-the-fly. Another common approach is virtual machine code that likewise runs in a special run-time environment, but is optimized for such use. Both solutions work, but in practice can essentially hinder their own usefulness due to the overhead of converting software on-the-fly.

Dynamic translation is an on-the-fly form of what we call binary translation. Binary translation is the art of converting executable binary machine code to other forms of executable binary machine code[18]. Another form of binary translation is static binary translation. Unlike the dynamic form, static binary translation converts the machine code before run-time, in theory sacrificing storage space for speed. Binary translation systems may also be made to be generalizable or universal, i.e. support translations between any form of executable machine code.

After exploring related work, it is evident that there has not been much interest for solving AIC using universal static binary translation. With the apparent benefits of universal static binary translation, such as near-native performance for any architecture, this then begs the question of why universal static binary translation has not been researched more. Is it an infeasible technology, or perhaps dynamic translation is more justified with the mantra "faster hardware will solve all of our problems"? Whatever the general consensus is, a universal static binary translation approach should be explored more before it is silently dismissed.

In this thesis, we do explore universal static binary translation and introduce our approach simply known as the Binary Translation System (BTS). We mainly focus on its programming framework for universally encoding, decoding, and manipulating user-level machine instructions. We will illustrate our techniques by explaining a prototype system that will decode PowerPC[43] instructions and encode SPARC V8[44] or V9 instructions. This thesis shows the feasibility of universal static binary translation

by its parallels to other existing technologies and the results that we achieve. This thesis also shows that like writing compilers or other translators, static binary translation is as much of an art as it is a science. Our results support the claim that universal static binary translation is feasible and has the potential of providing a good solution for AIC.

The rest of this thesis is organized in the following way. In section 2 the background is explained in more detail to show the motivation and need for the BTS project. In section 3 an overview of the machine code representation and translation process will be presented. Section 4 explains in detail how each of the modules in the prototype work, mainly consisting of the PowerPC decoder, SPARC encoder, and the core representation manipulations. Section 5 will give a brief introduction to programming with the BTS, and other issues to keep in mind while creating binary translators. Section 6 discusses evaluation of performance metrics, and presents qualitative comparisons to other existing systems that enable software compatibility. Section 7 will then wrap up our conclusion to our experiment and give a prediction for the potential future applications of universal static binary translation.

To eliminate any future ambiguity please take note of the following terms. **Source Program** will refer to the program or code that is being translated and/or manipulated. **Intermediate Representation** or what will become known as the **Intermediate Representation Tree (IR Tree)** is an architecture-neutral way of describing a source program. **Decoding** is the process of loading a binary executable into memory. **Smooth** will refer to our process of transforming a decoded representation into an Intermediate Representation Tree. **Decompilation** is the complete process of decoding and placing the source program into an intermediate representation. **Unsmooth** will refer to the reverse process of smoothing, where an Intermediate Representation Tree is transformed into the target's architecture-dependent representation. **Encoding** is then used to describe the process of putting the intermediate representation into assembly or machine code form. Also as previously mentioned, **universal** or **generalizable** refers to translations that are applicable to any arbitrary machine code language.

While we hope this thesis evokes much interest in the possibility of binary translation technologies, we will not attempt to make any presumptions on the legal or ethical implications on any of its potential applications.

2 Background Material

2.1 Requirements

In order to achieve general code portability across instruction set architectures (ISA), there needs to be some translation mechanism to convert the semantics of the source program to the target architecture before it can be successfully executed. This translator mechanism requires several measures to ensure a successful translation.

- *The first and most important requirement is correctness.* Each instruction must be understood and translated in its entirety, this includes all side effects. The source program may only be reduced by an optimization engine that leaves the logic of the program as a whole intact. For example, in order to achieve complete compatibility for the Windows Alpha platform, a translator known as FX32!^[1] provided a mechanism for dynamically and statically translating X86 code to the Alpha

architecture. Such an undertaking had to be complete and correct such that any application could be run without intervention by the user.

- *The second requirement is completeness.* While this requirement typically goes hand in hand with the correctness requirement, it is important nonetheless to require that the translation be able to work on any arbitrary source program and not make any harmful assumptions about compiler patterns, such as ignoring a rare instruction or side effect. As stated previously, the entire semantics of the source program must be translated unless altered by an optimization, in which case the logic of the source program as a whole should still remain intact. The entire set of possible inputs must all have mappings to the intermediate representation.
- *The third requirement is that the system must provide universal adaptation for any instruction set architecture, and enable the representation manipulations that will make translations possible.* Having a consistent representation in between the stages of decompilation and retargeting allows the engine to become universally adaptable. If designed correctly, the representation should also enable useful manipulations and optimizations for any translation process. One program analysis tool, known as StarTool[2], also takes this approach to enable the analysis of multiple programming languages. "An inexpensive way of achieving adaptability is to introduce an intermediate abstraction — an adapter component or adaptation layer — between an existing language representation and the program analysis." (Hayes pg 1) Usually, these requirements for achieving universal translation are in the capabilities of the intermediate representation.
- *The fourth requirement is that if deployed to an end user then the translation should be completely automated and transparent.* FX32![1] again was a great example of this. Not only did it provide the transparent dynamic translation, it also did static translation whenever possible for better performance, all completely without any user intervention. Another more recent example is the dynamic translation engine that Apple has for its newer X86 based computers to support the older PowerPC targeted applications[49].

Several approaches have been taken to address these requirements for AIC, all of which have proven to have good success. However, we have also discovered that each one typically does not satisfy all of the stated requirements. These related works in many different subfields is provided in the following subsections.

2.2 Programming Languages

High-level programming languages are a common choice for establishing code portability across architectures. In many cases, this involves recompiling source code to different targets. A popular example of this is the GCC[27] compiler that supports many languages and can compile or cross-compile to many different instruction set architectures (ISA).

However, compilers cannot use machine code as an input language. Therefore, in the many instances where the source code is not available, a compiler cannot even provide a possible solution for AIC. This is why retargetable source code does not meet any of our requirements. Portable source code cannot do translations between arbitrary machine code languages.

One solution for establishing compatibility is the concept of interpreted programming languages. An interpreted language is one where the source program is run inside a special run-time environment that executes a program on-the-fly using only the program's source code. The concept is solid and proven but requires the programs be distributed as source. This approach likewise does not meet any of our requirements because it cannot do any translations and forces the code to be exposed to the user.

There is also a dependence on frequently changing third-party software, namely the specialized run-time environment. However, the main drawback to interpreted programming languages is that in practice there can be significant overhead in parsing and executing a source program compared to compiling it. Even though we do not present speed performance as a formal requirement it is important that programs be able to run at a subjectively reasonable speeds, otherwise the purpose of enabling software compatibility can be defeated.

An example of a popular interpreted programming language is Perl[28]. Used frequently for web page scripting, a perl script will typically run on any machine with a perl run-time environment. Other examples include Python and Ruby.

For targeting specific run-time environments with portable code, cross-platform libraries such as WxWidgets[42] rely heavily upon compile-time flags to determine what library and system calls are available. Likewise, the standard C library and other similar cross-architecture will use compile-time information to target the appropriate architecture, (e.g. byte-swapping with network programming `htonl`, `ntohl`). Furthermore, using template and code transformation it is possible to achieve better cross-platform compatibility by specializing the program's code for its run-time environment at compile-time[34]. This technique would not only make general code faster, but also would help alleviate the headaches with inline assembly, specialized functions, or macros.

2.3 Compilers

Another solution is to create a translation service at the source code level or to enable more robust re-targetable compilations. A very earlier and simple example of this is the Amsterdam Compiler Kit[32]. The ACK uses the exact same conceptual model as a universal binary translator, where any arbitrary programming language is first targeted to a what they refer to as a Universal Computer Oriented Language(UNCOL). The UNCOL is a common intermediate language between all source-level representations and machine code representations. They referred to their UNCOL as Encoding Machine Language (EM).

Most modern compilers also use the concept of an intermediate representation or UNCOL, similar to the EM language, to allow cross-compilation of various programming languages. GCC[27] is one example of this, using register transfer lists to target C, C++, FORTRAN, and other languages to a number of different architectures.

Obviously, these example compilers do not meet all of our requirements, and are not capable of translating any binary code. They do however provide an intermediate representation and backend capable of meeting part of our third requirement. In fact, if these compilers created a frontend for machine code languages they would likely fulfill all of our requirements and become a capable binary translator. Therefore, the notable feature of compilers is that having a consistent intermediate representation is an important part in re-targeting from many languages to many machine code languages.

2.4 Run-Time Environments

Also related are modern operating systems that provide cross-architecture compatibility by storing user-level applications using virtual machine code. Examples of such operating systems include VORTOS[22], DixOS[3], and Intent2 RTOS[21]. Characteristically, they use a virtual instruction set architecture (ISA) to dynamically target operating system and user-level modules to the native host architecture. The virtual ISA approach in operating systems is arguably often chosen not only for compatibility but also security reasons.

Other run-time environments such as Java[26], are potentially good candidates for achieving compatibility. Java and languages like it use compiled virtual machine code to dynamically translate and cache native code using just in time (JIT) compilation on any machine, provided that the Java Virtual Machine is installed. The Java language also provides a large rich API, scheduling, memory management, and other things that give it many of the characteristics of a full operating system.

In terms of virtual machine code-based languages, Java is not unique in this approach. Pseudo-Code or PCode[33] was a virtual machine code instruction set that enabled languages, namely Pascal, to be portable once compiled. PCode could be run on a variety of real machines that supported a virtual PCode run-time environment. Similarly, newer languages such as C-Sharp use the .NET¹ common language runtime (CLR) environment to operate much like Java with managed virtual machine code and a robust API. Unlike other run-time environments however, .NET is not very platform independent since it relies heavily on the Windows OS.

The largest disadvantage to Java and some other virtual machine code systems is that in practice they tend to be slow even on modern day computers. This is probably due to the dynamic translation and largely in part to the heavy-weight virtual machine API/environment that it provides. Another disadvantage in practice is that its API often changes, deprecating, removing, or adding functionality, frequently breaking its portability. Obviously none of these solutions are a complete binary translator, but they do meet the some of the more important requirements. Specifically, they all provide a virtual machine code language that can easily meet part of the third requirement. Additionally, they not only meet the fourth requirement, but make it a point to do so as one of the main features.

2.5 Dynamic Translation and Emulation

Other similar approaches involve dynamic translation strategies that provide interpreting code on a much lower level. The difference between code interpreters(2.3, 2.4) and dynamic translators is that interpreters have translated machine code implicitly built into the interpreter as the behavior is created by the interpreter. A dynamic translator however, dynamically builds the native machine code on-the-fly to later be cached, loaded, or run indirectly.

Examples of dynamic binary translators include bintrans[19] and Walkabout[20]. It should also be noted that Apple has also made use of dynamic translators in the past and present to support their first switch from 68000² to PowerPC[49] to Intel processors. Characteristically, dynamic translators usually implement some caching system for storing previously translated code, and sometimes optimized pieces of translated code. Caching is usually required to get acceptable performance out of a dynamic translator.

¹<http://msdn.microsoft.com/netframework/default.aspx>

²http://en.wikipedia.org/wiki/Mac.68k_emulator

An extension to dynamic translation is emulation, where the emulator runtime not only provides a dynamic translation service but also must support much of the emulated machine's other responsibilities to support appropriate system level instructions or calls for important things such as IO.

One example of an emulator is Bochs[23], which is an open-source X86 emulator. Another example of an emulator is the FX32![1] system that provided Alpha support to Intel applications compiled natively for an Intel Windows NT system. FX32! was also much more than an emulator, supporting profile driven static translation as well as specialized linkers and loaders for connecting X86 system calls to native alpha system libraries.

Alternatively, dynamic translation can also be used as a fundamental feature for hardware implementation. For example, it is possible for a processor's instruction set to be different than the actual programmer's instruction set architecture. The IBM System/360 was the first family of computers to popularize this using microcode. Microcode allowed hardware to vary while keeping the programmer's instructions consistent. This was accomplished by converting instructions into smaller execution units called microcode that could be different from implementation to implementation. The S/360 also had the capability to emulate older ISAs for backwards compatibility.

Another example where dynamic translation is a critical part for the operation of hardware is in some Very Long Instruction Word Processors (VLIW). VLIW commands are special processor commands that unlike typical instructions combine pre-determined parallelism with many instructions. VLIW processors essentially get rid of the traditional dispatcher and rely on the compiler or dynamic translator that makes the VLIW commands to combine instruction commands appropriately.

The Crusoe is one such processor that uses VLIWs[11]. The Crusoe uses a piece of software called Code Morphing Software to dynamically translate X86 instructions into its own VLIWs. Another VLIW architecture is the IA-64. The IA-64 mainly relies on compilers to create the VLIWs. It is however also capable of emulating IA-32 instructions, but since it does not have a complex dispatcher it is actually quite slow at this task.³ Similar to the Crusoe, the Daisy[14] project was a much more capable dynamic translator that targeted any ISA to the VLIW S/390. Daisy has all the capabilities of a universal dynamic translator with the exception of having a single VLIW target.

Dynamic translators largely have the distinct advantage of being transparent to the user. However, like interpreters the biggest drawback still remains to be the execution speed and the overhead of using multiple instructions to generate a single translated instruction. In practice, this may defeat the purpose if the translator does not cache the translated code, or the cache utilization is insufficient. A properly implemented dynamic translator or emulator has the potential to meet all of our established requirements, and could very well provide a generalizable binary translation system. However, the trend of emulators and dynamic translators is that they tend to be extremely specialized for their applications, therefore many of them do not meet the third requirement, and could not provide a generalizable service.

2.6 Decompilers

Decompilers also fall close to the realm of binary translation since they provide much of the same functionality that a binary translator does. That same functionality being an intermediary form, namely

³Intel scraps once-crucial Itanium feature: http://news.com.com/Intel+scraps+once-crucial+Itanium+feature/2100-1006_3-6028817.html?tag=nefd.lede

decompiled source code, may be easily retargeted using another compiler.

There are several decompilers worth mentioning. DCC[25], Boomerang[24], and REC[29] all will decompile various machine code formats into C source code.

In theory a decompiler could work well enough to provide retargetable source code, but in practice decompilers tend to concentrate on extracting and approximating as best they can what the original code looked like, rather than producing compilable output. Approximating being the key word here, because the compilation process that made the source program gets rid of a lot of structural information. The processor does not care about the alignment of data structures or switch statements etc., it just needs to know how to execute them.

Even though we commonly refer to universal static binary translation as a practice of decompilation, there remains the fundamental difference of informative vs. accurate decompilation. Informative being the goal of a decompiler, this does not inherently mean it will satisfy all of our requirements. Commonly in practice, all the requirements are not met; for the examples we mentioned the requirements of correctness are not met. While they do provide insight into the structure of the original source code, they do not always provide a source code extraction that can be recompiled. Another generalizable requirement not met by decompiles is the fourth of transparency. While they do have the possibility of being transparent if the other requirements were met, at present a decompiled program may require manual modification for it to recompile properly.

2.7 Static Translation

Other systems for porting applications do so completely offline, where the entire retargeting process is done before the translated application can even be run. This approach has the obvious advantage of eliminating the run-time overhead of the actual translation process. However, there are some tasks that static translators do not handle so well, such as translating the targets of indirect branches. Overall, static translation offers a good alternative to the dynamic translation solution where obvious performance slow downs would be present. Many specialized static translators have shown that it is possible and can be quite useful, but typically generate less efficient code than a compiler would.

Static binary translators typically work almost identically to a combination of compilers and decompilers. Compilers will usually work in a series of intermediate representations to manipulate the source code to be simplified down to machine code. Decompilers will attempt to reverse the compilation process and also use a series of intermediate representations to analyze and manipulate the program to approximate the structure of the original source code. Static translators mimic a decompiler, and then a compiler once the intermediate representation is extracted. The static translator focuses on reaching this point and unlike the decompiler is concerned with accurately representing all of the semantics present in the program instead of guessing what the source looked like. Once the independent representation is reached, this is as good as source code and the compiler part of the system can kick in and can complete the translation by targeting the independent representation to any architecture (e.g. much like it can with C). A simple illustration of this is presented in Figure 1.

Aside from other very specialized translation systems, the University of Queensland's Binary Translator UQBT[8][4][9] project can be considered the only group to have done relevant work in designing a generalized static binary translation system. UQBT provides a multi-tiered architecture to providing

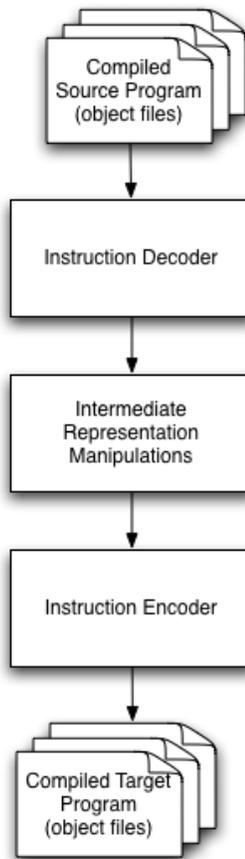


Figure 1: Simplification of the static translation process.

a universal translation system. The UQBT relies on the New Jersey Machine Code Toolkit[30] and other custom made specification language modules for achieving generality. Each specification language module plays an important part in specifying the functionality for reading, manipulating, or outputting binary files. Some of the following specification languages exemplify the motivation for this design: Specification Language for Encoding and Decoding (SLED) and Semantic Specification Language (SSL)[6] for describing the syntax and semantics of machine instructions, Procedural Abstraction Language (PAL)[7] for describing the ABI. The UQBT passes the source program through a few representations in the process before reaching its final target. A basic simplification can be given as follows: first the original source machine code is translated to a machine-dependent high level register transfer list language (HRTL), which is in turn translated to a machine-independent HRTL, before the process is reversed to get the resulting machine code.

In practice, static translators can offer significant performance gains compared to dynamic translation. If the translator is implemented to be universal it would meet all of our established requirements. In fact, the UQBT does meet all of our requirements, it can or has the capability to support correct and complete representation of instructions, a flexible intermediate representation, and could become fully automated.

2.8 Analysis Tools

Analysis tools are similar to binary translation tools in the fact that they also use some sort of intermediate representation. Some analysis tools operate on compiled programs, others on source, but their engines all rely on a consistent known representation.

One relevant example is the Bunch[39] tool created at Drexel University. Bunch is an analysis tool that performs clustering to determine the interdependencies and modularity of classes and files. Unfortunately, this proof of concept is not an all-inclusive solution, since it has the same downfall of other analysis tools of being language-specific to Java, as its consistent form.

Another work we would like to point out is Alloy[40][41]. Alloy is a simple modeling and analysis tool for testing execution and invariants in a system. It also provides analysis features for testing conformance and test cases. This tool is extremely useful in many active areas of creating system designs, however it may lack practicality by requiring a strict modeling language.

Similar tools have been commercialized such as the ones made by AbsInt⁴, but they usually suffer from the same disadvantages of the other tools we discuss, in this case forcing the user to use a formalized specification language. In other cases of commercial tools the analysis is just as restrictive, as it will only support Java or some other specific language.

Dynamic analysis is also a very important area of research since much of the program semantics cannot be derived exhaustively through static analysis. Instead, tracing program execution can reveal statistically critical sections of code and also interdependencies that may have been overlooked in the traditional static analysis. This reveals a key weakness in static analysis, as it is an intractably difficult problem and there are not enough resources to do exhaustive searches through the code all of the time.

⁴<http://www.absint.com/>

2.9 Modification and Instrumentation Tools

Another advantage for having a consistent representation is to have the ability to modify the semantics or instrument the behavior of programs. Once programs are decompiled or available in a abstracted representation, the user often has other intentions than to just see what it looks like decompiled. Higher-level modification over and beyond just refactoring and integrating can be highly desirable. This is especially true with a language such as TXL[37] that was specifically designed for manipulating trees and as a result has been used many times for program transformations. A notable use of such transformations has been source code level analysis in several research projects to actually find, isolate, and correct problematic C buffer overflow exploits[35][38]. Common applications with parse trees, such as compilers, may someday go beyond just optimization manipulations, but actually increase the general security of programs on a regular basis.

2.10 Summary

In summary, all of the related works mentioned fundamentally deal with translating languages. The one theme throughout all of them is that they each have an underlying engine that expects a consistent representation of the source program. Even more, most of the related works also create their own intermediate language between reading the source program and translating or acting on it.

It is important to note that generalizable static binary translators like the BTS and the UQBT provide enough facilities such that almost any one of the related works could be built on them. For example, to create a compiler, all that is needed is to create a new front end to convert a source program into the translator's intermediate language, and the back-end can retarget appropriately from there. Similarly, operating systems and runtime environments that use various machine languages to store programs, a static translator could translate code on-demand or during idle system time. If done properly, it appears that a universal static translator could be one of the best solutions for architecture-independent computing.

3 Overview of the Binary Translation System

We now introduce the Binary Translation System (BTS). The Binary Translation System is a structured programming framework that was created for the main purpose of statically translating machine code between arbitrary architectures. The framework was designed to be extensible enough to support any machine code format, as well as various object file formats, optimizations, or source code parsers.

This system varies from the previously mentioned background work in that its purpose was to explore the feasibility of universal static binary translation. It does not provide specification languages, dynamic translations, or any other auxiliary features. The BTS also was implemented with a focus on practicality, i.e. to be good enough for common architectures. The following subsections establish our goals and give a purely conceptual overview of our system, just short of any implementation details.

3.1 Goals and Objectives

The overall main objective of implementing the Binary Translation System is to provide a novel universal static binary translation system that can demonstrate the feasibility and merits of a structured programming framework approach. We do not claim to be solving AIC all at once, but with well defined goals we hope to argue that universal static binary translation is a step in the right direction. We predict that universal static translation could potentially rival any other approach, with results running at near native-compilation speeds. We also predict that an extensible general framework can be implemented by exploiting the commonality between specific translation goals (e.g. X86 to SPARC or SPARC to PowerPC), and the use of generic manipulations on the source program. We believe that the implementation of our sample prototype system, and refining the methods used that a generic framework will even begin to emerge that will be flexible enough to adapt to other architectures, and have performance good enough to support our predictions.

To reach our main objective however, there are many intermediate supporting goals that must first be reached in order to complete and evaluate the system as a whole.

These goals are:

1. Explore and implement the minimum requirements for a prototype universal binary translator.
2. Document the specific methods implemented and used in this prototype.
3. Establish a working generalizable framework with those methods that can be useful for other architectures.
4. Evaluate the system quantitatively for speed and performance.
5. Determine how this specific system compares to the quality of other systems that provide comparable solutions to software portability.
6. Hypothesize if the prototype system demonstrates potential to be universally applicable across all common architectures.

In order for these goals and objectives to be easily reached and evaluated we will be placing constraints on what is actually required of the prototype. The instructions that will be translated will be user-level instructions only, not kernel-level instructions because it is not a simple task to extract common abstractions across this particular class of instructions. Additionally, we will also focus on all the non-floating point instructions in order to make a more clear and concise case for the tests that exemplify our system. Lastly, we will also be limiting the API function calls to standard C library, and posix system calls that are present on many of our source and target environments. Statically linked libraries, if present, will be translated as normal code and not recognized for any other purposes.

3.2 Overview of the Translation Process

Recall from Figure 1, the translation process converts raw machine code into a representation that represents the operations present in the source program. This representation is then put through a series of manipulations (e.g. remove architecture-specific artifacts or optimize) before it is encoded into the target machine or assembly code language.

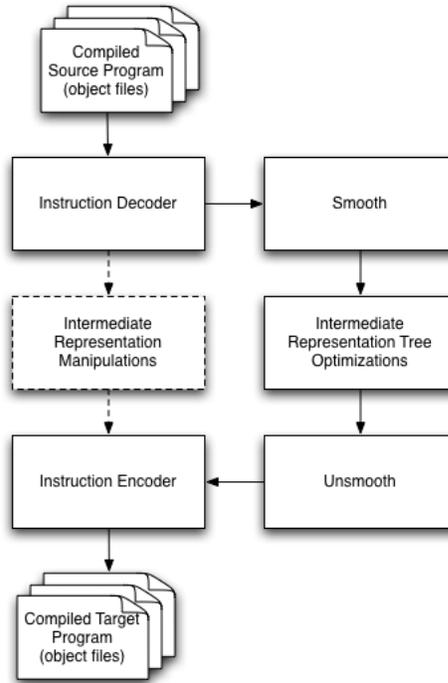


Figure 2: Overview of the static translation process.

In Figure 2, the series of manipulations is illustrated in a bit more detail. What is actually happening, is the representation is first manipulated to remove any architectural-dependent properties (e.g. function epilogues and prologues can be made into a single function). This process we refer to as the smoothing process. Once this is complete, the representation is completely architecture-neutral. In this state there are many possible manipulations that can be applied to the source program. The only architecture-neutral manipulation of interest to us is a simple optimization manipulation. As previously mentioned, once the decompilation phase is complete and the architecture-neutral representation tree is reached (what we will refer to as the Intermediate Representation Tree or IR Tree), the compilation phase of the system can complete the translation. We can easily justify this since a C parse tree is analogous to our Intermediate Representation Tree. In fact, for many of our examples we use C to represent our intermediate representation tree because they are so analogous.

Aside from the series of summarized actions and manipulations the remaining system details are fairly intuitive. Figure 3 summarizes conceptually the minimal requirements for a static translation system that we have identified. The very first step for the entire static translation process is to decode the object files where the source program is contained. For each object file, the magic number is fetched and the appropriate object file decoder then reads each section into memory. This will typically include code, data, and symbol tables.

For each code section found, the system uses specific processor decoders to sequentially decode each instruction. After each instruction is decoded into an architecture-dependent tree, the relative branches are resolved, and the external symbols matched to locations in the code. The decoded instructions then need to be analyzed as a whole to extract the structure of the program such as functions and function

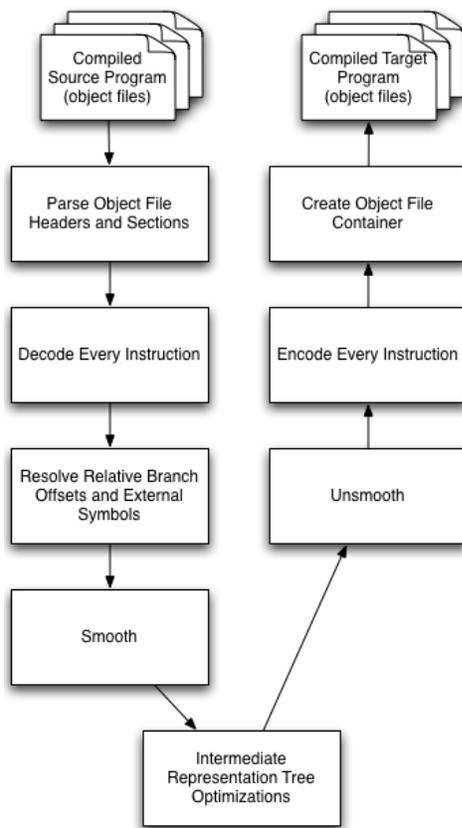


Figure 3: Detailed overview of the static translation process.

calls (the smooth process).

Once the previous steps have been completed, the system will have its Intermediate Representation Tree. It is at this stage where all general manipulations and optimizations take place.

The remaining stages are then very similar to a typical compiler, where the representation is manipulated to specifically represent the target architecture (the unsmooth process). This representation is then encoded into the target's assembly or machine code language. The machine code is then placed inside the target's object file format, or the assembly is given to an assembler to complete the translation.

3.3 System Framework

The Binary Translation System was designed and implemented using C++ because many of the features our representation tree has can be easily organized using C++.

The layout of the system is shown in Figure 4.

The first important data structure to note is the **GenericNode** class. This class is used as a base class for every other class in the framework. It provides a single pointer to the next **GenericNode** in the linked list as well as a single pointer to the start of its meta data linked list. Being the underlying base class it was extremely important to keep its size as small as possible in anticipation of very large source programs. The only other attribute that the **GenericNode** class has is the `refCount` as a way to manage memory. The `refCount` value is a counter to keep track of how many things currently have strong links to this node. Once the last referring node dereferences this node then it will be deconstructed and deleted safely. This memory management technique allows for node references to be maintained naively and without worry of leaking memory or dangling pointers, unless there is a bug that changes the reference counter unnecessarily.

The **MetaDataNode** class goes hand and hand with the **GenericNode** class. The **MetaDataNode** provides ways of adding special variant information to the different instances of the **GenericNode** class (e.g. `AddOpNode` or `Section`, etc.). The **MetaDataNode** has an attribute value that represents its ability to contain any data; namely a textual string, integer, or another **GenericNode**.

The **IRNode** class is the abstract base class for all of the other nodes that make up the various representations. Its main requirements for derived classes is that they implement the identification function `GetNodeType`, which returns an enumerated value to identify what type of **IRNode** it is. Additionally, having the **GenericNode** as a base class allows each representation node to have the next node pointer. Combining this with children attributes (e.g. `operand1` and `operand2` in the `BinOpNode` class), a tree-like structure begins to emerge. In Figure 5, a sample tree illustrates how the Intermediate Representation Tree nodes get built together to make a complete representation using children and the next attributes. We use C and a C-like parse tree representation to illustrate how the nodes are organized in our Intermediate Representation Tree. Everything needed to completely represent the semantics in the source program are created as **IRNodes**, from function, add, and assign nodes to logical and bitwise operators. It should also be noted that the semantics are as low-level as possible without actually emulating the digital logic inside the processor (i.e. conditional and unconditional branches make up while and for loops).

The **AbstractProcessor** class is the part of the framework that provides the interface for implementing specific processors, such as a SPARC or PowerPC processor. These are responsible for decoding

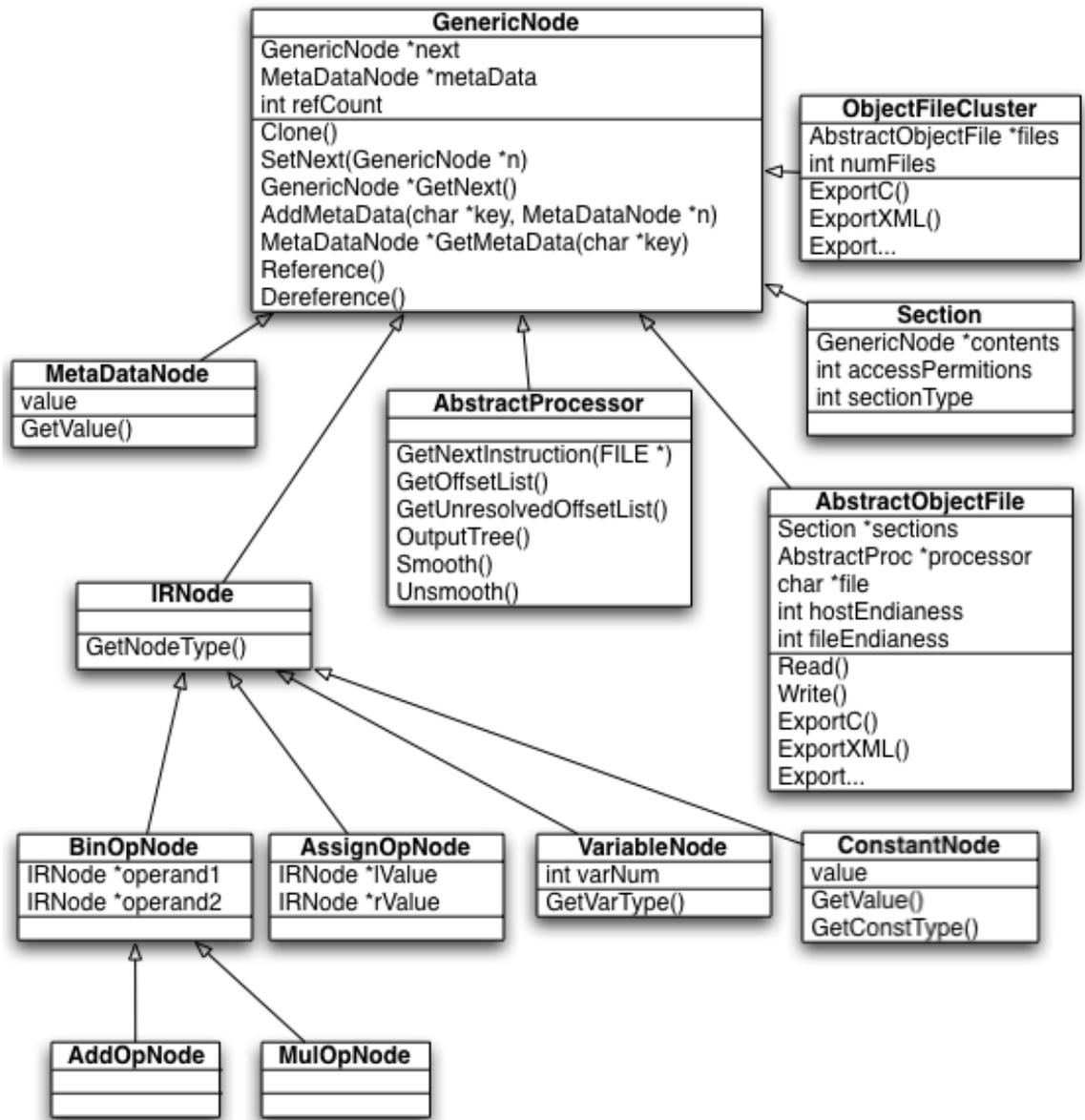
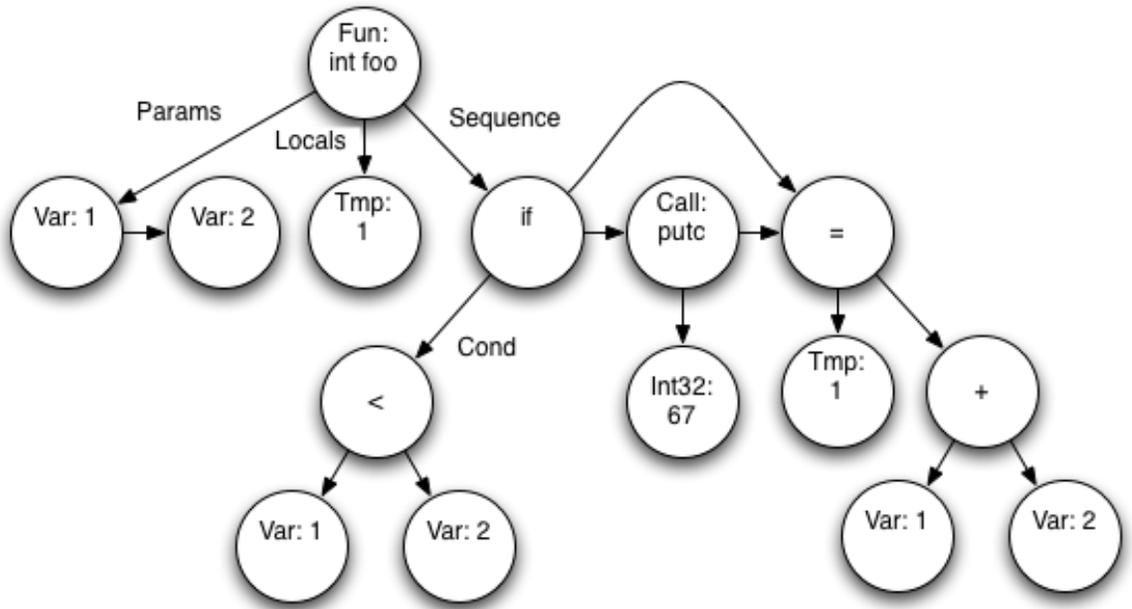


Figure 4: Framework of the Binary Translation System.



```

int foo(int var_1, int var_2)
{
    int tmp_1;
    if(var_1 >= var_2)
        putchar(67);
    tmp_1 = var_1 + var_2
    .....
}

```

Figure 5: Sample Intermediate Representation Tree

and encoding instructions as well as manipulating the machine specific representations.

The **Section** class is a general container for holding object file sections. Everything from data, code, and symbol tables may be contained in a section. The **AbstractObjectFile** provides the interface for reading and writing information to the specific object file formats. Lastly, the **ObjectFileCluster** is a collection of object files that represent the source program. This class is provided simply as a way to organize the translation process.

Other specific methods provided in these classes will be explained in later sections.

3.4 Machine Code Representation

The goal of this project was to experiment with a machine code representation that would be robust enough to capture a superset of all the common invariant semantics of machine code languages while providing the flexibility to allow for exceptions to be made for variant instructions. While ultimately having the goal of assimilating the represented code to adhere to a strict intermediary form that masks its origins and is useful for performing any kind of program manipulation (e.g. optimizations or instrumentations) before being assembled to the target architecture.

The underlying representation that represents the source program at all times is actually a non-traditional tree (as shown previously in Figure 5). It is non-traditional in the sense that there exist links between jump nodes and their targets that would break a pure tree structure, but for all extensive manipulation purposes these weak links can be ignored. Everything from the object file sections and symbol tables to the code and data itself is stored in the tree. The general meta data attribute scheme is also used to catch uncommon variant characteristics.

The concept of a proxy node is introduced for doing generic tree manipulations with links between branching nodes. Typically if a manipulation deletes a node in a sequence of operations, then only its previous neighbor has to be concerned. However, there is no way to tell if the node to be removed is a target of a branch operation. Therefore, a proxy node is created and placed in as meta data for that node. When that node is removed for whatever reason, certain meta data is passed onto its successor, including the proxy node. This proxy node can be the target of all the branches and never be removed, solving the problem of missing targets during manipulations.

As previously mentioned, the tree model itself is a short C++ class hierarchy that emphasizes a linked-list relationship, leaving the children nodes to be the responsibility of the specific derived classes.

Below is a full list of enumerated IRNodes types that we have chosen to represent the full semantics of any given source program.

- NOP - Don't do anything
- PROXY - Proxy representative of another node
- ADD - Add the two children together
- SUB - Subtract the two children
- MUL - Multiply the two children
- DIV - Divide first child by the second
- MOD - First child modulo the second

- ASSIGN - Assign rValue to lValue
- BIT_OR - Perform bitwise-or
- BIT_AND - Perform bitwise-and
- BIT_XOR - Perform bitwise-xor
- BIT_NOT - Perform bitwise-not
- BIT_ROTLEFT - Perform bitwise-rotate with optional buffer
- BIT_ROTRIGHT - Perform bitwise-rotate with optional buffer
- BIT_SHIFLEFT - Perform bitwise-shift
- BIT_SHIFTRIGHT - Perform bitwise-shift
- LOGICAL_AND - Perform logical-and
- LOGICAL_OR - Perform logical-or
- LOGICAL_NOT - Perform logical-not
- LOGICAL_AND_SIDE - Perform logical-and on children with all side-effects (evaluate all disjunctions/conjunctions)
- LOGICAL_OR_SIDE - Perform logical-or on children with all side-effects
- EQUAL - Perform equal comparison on children
- NOT_EQUAL - Perform not equal comparison on children
- LESS - Perform ordered less-than comparison on children
- GREATER - Perform ordered greater-than comparison on children
- LESSEQ - Perform ordered less-than-equal comparison on children
- GREATEREQ - Perform ordered greater-than-equal comparison on children
- PUSH - Perform push of exactly 1 child onto local stack
- POP - Perform pop of exactly 1 child off of local stack
- CONSTANT - By default is a signed 32 bit integer
- VARIABLE - Variable/register node, contains id number
- ADDRESS - The value stored at address given by child 1, with optional offset at child 2 and optional scale factor at child 3, such that it equals $(\text{child1} + \text{child2} * \text{child3})$
- JUMP - Jump to location of the child node
- IF - If child 1, do child 2, else do child 3 (which is optional)
- ATOMIC - Execute children atomically
- SCOPE - Evaluate child 1 in scope of other children
- FUNCTION - A special scope that has a name, argument list, and list of local variables
- FUNCTION_CALL - Call child 1 with parameters of other children
- RETURN - Return to caller – children are return arguments, but

Note that there is no while node, since this can be easily created using a combination of jumps and conditional jumps. Additionally, note that there is no arithmetic negation, since this can also be easily made by taking the two's complement (bitwise negation, then add one).

One of the main design concerns with these nodes was to reduce the amount of data in each node as the representation could get quite large quickly depending on the source program.

3.4.1 Tree Types

The underlying tree structure is used in a variety of ways to represent the source program at each stage of the translation.

The **Architecture-Dependent Tree** is a controlled way of segwaying from machine code into a truly architecture-neutral representation, allowing branches and other big-picture things to be resolved before applying similar algorithms to convert it into the architecture-neutral representation.

This tree also has no high-level semantics but instead exhaustively mirrors the low-level register and memory semantics of the source program's machine code. For example, the original C source code of the source program may contain a comparison operator (e.g. `a == b`), and the resulting assembly code may be a subtraction or a special comparison instruction that sets multiple bits in the comparison register (greater than zero, less than zero, is zero etc.). The architecture-dependent tree will create a string of operations to represent each one of those comparison bits getting set. This has the potential to create some very bloated code very fast, however any decent optimizer should be able to reduce back down to its original C source code logic without the extra bloat.

The following is a list of properties that the architecture-dependent tree could have:

- Every addressable block in the register set shall have its own unique variable number, including processor state register bits.
- Many variables will still be represented as memory accesses.
- Function calls will not be present, instead there will be a series of assignments loading arguments registers and an unconditional branch.
- Functions will not be present, instead the function epilogues and prologues will be represented.
- Returns will not be present instead registers will be loaded and a branch to a link register or something similar may be present.
- Less commonly used instruction side-effects such as overflow detection are represented in meta data.

The second use of the tree structure is for the **Architecture-Independent or Architecture-Neutral Tree** which we refer to as the **Intermediate Representation Tree or IR Tree**. The IR tree is the ultimate goal of the decoding process, creating a tree that should universally have the same characteristics and invariants regardless of what machine language the source program is in. It is this tree that should look almost identical to a parse tree to a traditional programming language. Aside from having the potential of coming from a variety of sources, the IR tree also maintains many of the same characteristics a parse tree might. Most importantly, the IR tree may be optimized and manipulated in

this form before it is encoded to the target architecture. Any optimization for a procedural-like parse tree should also be valid on the IR tree, such as popular SSA optimizations[31].

The following is a list of properties that the Intermediate Representation Tree should have:

- Every register will be assigned a temp. variable number on a first come basis.
- Every addressed memory slot will be assigned a variable number and its scope will be assigned depending on its usage.
- Functions will be present with a full list of arguments and local variables.
- Return statements will be present with an optional return value.

3.5 Object File Decoding

3.6 Machine Code Decoding

In order to develop a new decoder for other architectures using the BTS, one needs to be familiar with the particular machine architecture or have the programmer's manuals for the architecture close at hand. Much of the decoding is manually scripted into the C++ framework, matching one instruction at a time to its corresponding representation.

The required interface functions for implementing a decoder are given in the AbstractProcessor class. These functions include GetNextInstruction, GetOffsetList, GetUnresolvedOffsetList, and Smooth which is discussed in a later section. These functions highlight the steps that the BTS goes through while decoding and representing the source program.

GetNextInstruction is used to retrieve the next instruction from an open file and decode it into an architecture-dependent tree. This function is only responsible for the myopic decompilation of a single instruction and representing it in tree form. It also must be concerned with endianness of the source machine code, making sure it is being decoded correctly, and it must also know exactly how large each instruction is to read in the correct amount.

Every pass through the GetNextInstruction decoder must add an entry to an offset list to keep track of the originating offset of each instruction's representation from the start of the code section. Additionally, if a branch is encountered it must also keep track of the target addresses for the representation. GetOffsetList and GetUnresolvedOffsetList will return these two offset lists respectively. These two offset lists will allow the system to be able to resolve any unresolved relative references present in the code.

The smooth function and other manipulations constitute the last phase in the decoding process. They are responsible for essentially decompiling the architecture-dependent representation and manipulating it to conform to our intermediate representation tree requirements. This process varies with each type of processor implemented; more on the techniques used in these functions will be discussed in later sections.

3.7 Tree Manipulations

At the very minimum at least 3 tree manipulations are required to complete a translation.

The smooth operation is responsible for smoothing the architecture-dependent tree into the standard intermediate representation tree, such that the encoding phase can not determine where the source

program came from. The smooth technique identifies pertinent information specific to the source architecture so that things such as functions and function calls can be extracted.

The simplify operation is necessary for doing cleanup from the smooth operation and eliminating meaningless operations from the tree. It will eliminate operations that have empty operands and do very basic expression optimizations (e.g. $(1 == 1)$ should simply resolve to 1). The simplify function is also designed to know exactly how to remove and reorganize nodes. When there are weak links present (i.e. a branch and its target) extra attention must be given to ensure that the link remains correct even in the event of a target getting removed. To achieve this a proxy node is created and is associated with the target through its meta data. Each piece of meta data was then created to have the capability of being transferred in the event that the original node would be replaced with another node.

The unsmooth operation simply transforms the intermediate representation tree back into an architecture-dependent tree to be easily encoded into the target machine code language. It is responsible for generating function prologues and epilogues from a function node as well as allocating memory slots and registers for variables. The actual output to machine or assembly code functions should have an almost one-to-one relationship with the representation at this point.

Optionally, if the source is source code then instead of smoothing the tree a flatten operation also needs to be applied to eliminate subexpressions. The resulting tree is then considered to be the architecture-neutral intermediate tree which may be subsequently unsmoothed and encoded almost identically to how a traditional compiler may work.

These operations may be combined with the other operations as well to make the process more efficient but for all intensive purposes they were described separately to indicate the significance of each one.

3.8 Machine Code Encoding

The process of actually emitting the resulting machine code is not pertinently novel in terms of previous work. In order to take the prepared IR tree and encode it as architecture specific assembly or machine code, a very similar process to compilers is used, or in some cases a compiler may be used as a surrogate back-end.

The thing of importance to note is that we have simply made it a modular process to be reflective on the initial decompilation part of the translation.

4 Static Translation with the Binary Translation System

Simply explaining the concepts of the methods used in the Binary Translation System cannot even begin to describe what type of hurdles a real implementation must get past. Writing support for any instruction set architecture will inevitably require some trickery to get the decompilation process just right. It is also expected that each additional processor will likely require some other changes to the underlying framework and generic support functions to accommodate the new layout and object file, or the new stack usage conventions of a specific ABI, etc. This is because the superset of characteristics from all possible sources grows.

As mentioned previously, our prototype system implements the capabilities to decode PowerPC instructions and encode SPARC instructions, along with the necessary tree manipulation functions. The following subsections explain the many implementation details of our prototype.

4.1 Ad Hoc Static Translation

The first version, while investigating all of the issues presented in this thesis, simply used the architecture-dependent tree to do an ad hoc translation from PowerPC to SPARC. While this was capable of translating a simple test suite, its limitations quickly hindered its scalability.

This method involved doing mappings of registers from PowerPC to SPARC and overflowing other register usages into memory where necessary. Argument passing required similar treatment. The stack layout and usage was also mapped to some degree of success. The key thing that made this somewhat possible is the PowerPC and SPARC are both RISC processors with very similar register sets.

It should be obvious though that such conveniences of mapping most of the registers would not be applicable for translating to any arbitrary architecture. Attempting to mash PowerPC code into X86 would quickly reveal its limitations with the intrinsic difference between stack usage alone.

While this approach may satisfy most of the previously mentioned requirements it does not however meet the stated requirement of providing an adequate intermediate representation to promote code reuse.

4.2 Implementation of the PPC Decoder

In the prototype system, the PowerPC decoder was able to fully support all of the non-floating point user-level instructions using a few opcode indexed jump tables and templated functions. The only maintenance information required between instructions was the final lists of offsets as previously mentioned.

The current implementation for the PowerPC decoder relies heavily upon template functions to take advantage of the fact that some decoding methods may only differ by the type of node that is created (e.g. a multiply vs and add operation). A large decoding function jump table is created with an array of function pointers to various templated functions that take advantage of similar instruction forms. The pointers are assigned statically the values of the appropriate decoding functions and are indexed by the opcodes, sacrificing space for speed. The function `GetNextInstruction` is then responsible for extracting the PowerPC opcode and making the correct jump to the tree generating functions. On the PowerPC architecture each instruction is a fixed size of 32-bits. This allows the current offset counter to be easily incremented by 4-bytes each time the `GetNextInstruction` function is called.

After the complete tree and list of offsets and unresolved offsets is reported (to the object file decoder typically) the tree can then be transformed into our architecture-neutral IR Tree. The architecture-dependent tree will have variable nodes that have identifiers representing the registers that they originated from. For example, the stack pointer which is general purpose register 1 will be represented by the identifier 1. Following this pattern the complete set of general purpose registers are represented by variable identifiers 0-31. After this the 32 floating point registers are represented by identifiers 32-63. The single 32-bit condition register however is represented by not 1 but 32 identifiers since each bit is important and addressable. The remaining registers are represented in a similar manner. Each variable node is also typed to represent 32-bit integers, floats, or boolean bits, etc.

Other characteristics about the PowerPC architecture-dependent tree is that the memory addressing will also be represented in a raw form. For example, if the program was calling a function and pushing an overflowed 9th parameter onto the stack the tree would have an assignment with an address node with a base of the stack pointer (variable 1) and an offset of 56.

A special function, that we refer to as the smooth method, was created to assimilate the entire tree into something that the rest of the system can recognize as the IR tree. In this case the smooth function was responsible for recognizing mainly functions, function calls, returns, and making true variables out of memory slots on the stack. To recognize a function call for example, the branch may be preceded by a list of instructions to load up appropriate registers or memory slots to pass arguments. The smooth process takes this and combines the calling sequence of instructions into one node where the passing of arguments is represented by a single function call node (i.e. `funcall(5,6,7,8);`) where there is no architecture specific setup. The smooth process can be viewed as the simplest form of abstraction taking the machine code semantics into a higher reverse-engineered C-like structure.

4.2.1 Full Instruction Semantics

The completeness requirement we met by representing the complete semantics of every instruction supported. This mainly included keeping track of status bits in the processor state and earmarking the generated tree for instructions with particular side effects. We also would support all legal forms of an instruction using no discrimination towards the operands or flags that are set.

For example, setting and keeping around the carry bit from an addition operation might be necessary for doing 64-bit arithmetic on a 32-bit architecture. There is no simple way of expressing this in C except to use a primitive 64-bit data type. To note that the operation required a carry to be saved, we simply added a flag in the operation's meta data. We anticipate that fundamental instruction side effects that alter the processor state will be present across many of the common architectures. For example, setting a carry flag on an X86 architecture or setting a flag in the exception register on PowerPC or setting a flag in the processor state register on SPARC, is a characteristic across many of the common architectures.

Not all of the semantics are kept in their full form when the immediate decoding takes place. In addition to maintaining proper mechanisms for representing internal processor states, we evaluate any semantics that we can at decoding time. For example, some PowerPC instructions such as `subfic` (subtract from immediate) will require the immediate field be sign extended before it is used as a 32-bit operand in the expression. So before a bitwise copy places the immediate into a constant tree node we extend the sign bit across the upper 16 bits. Another example is `addis` (add immediate shifted) where the immediate field is first shifted into the upper 16 bits of the 32-bit operand before being used.

Any logic that the processor uses internally is not exhaustively represented, rather we simplify it wherever possible. Our discretion on how to represent instructional semantics is purely pragmatic. For example, the digital logic in the ALU of an add operation is just represented by an add node, but a two's complement negation is represented by the arithmetic (bitwise complement and add one).

4.2.2 Peephole Decompilations

As mentioned earlier, the decoding phase will decode one instruction at a time, and any internal processor state bits will be represented as implicit side effects. This then requires that peephole decompilation be

implemented in the decoder's smooth function to ensure that small sequences of code are decompiled to a higher level structure to remain consistent with the rest of our IR tree. We use the same discretion here as with representing an add operation with just an add node, and nothing lower. Likewise, it is advantageous to retain a consistent level of abstraction across sequences of instructions no matter what the source architecture is.

A great example of this is 64-bit arithmetic. If the original source program compiled 64-bit operations the representation may end up being completely different depending on the architecture. A 32-bit architecture would show two add operations happening with a carry bit being passed around, while a 64-bit architecture would simply use just one operation on two of its registers. Such common program behavior should then be abstracted to the most simplest form, in this case a single operation with 64-bit variables. Again we take a completely pragmatic approach to what should and should not be decompiled using these techniques, because our goal after all is not an informative decompilation but an *accurate decompilation*.

The following snippet of code illustrates the difference with instructions and compiling/decompiling between 64-bit addition on a 64-bit architecture and a 32-bit PowerPC architecture. The following C code represents a high-level abstraction that 64-bit addition should be represented as while compiling or decompiling. Paying close attention to the status bits and temporary variables, a peephole decompilation using *trans-instructional semantics* should be able to extract this representation from 32-bit code.

C Code:

```
long long a,b,c;  
a=b+c;
```

Corresponding 32-bit PowerPC Assembly:[36]:

```
A low 32 bits in r2, high bits in r1  
B low 32 bits in r4, high bits in r3
```

```
addc r6, r2, r4  
adde r5, r1, r3
```

Corresponding 64-bit PowerPC Assembly:

```
add r3, r1, r2
```

4.2.3 Sub-Instructional Variables

In order to keep the translation as consistent and streamlined as possible, a constraint on the decoding process was made to keep all operations in the decoded tree in *three-address form*. Because of this, some instructions had to be represented in a short sequence of operations instead of just one. Some instructions on the PowerPC ISA will essentially execute a series of operations, and some instructions execute complex expressions, both of which in our representation need to be represented by a sequence of operations.

When a complex expression in particular, requires simplification into three address form, the results of these extra operations are then placed into additional temporary variables to represent the values present in the processor pipeline upon execution. These extra temporary variables we call *sub-instructional variables*. For example, on the PowerPC architecture, the instruction mulhw (multiply high

word) will place the high order 32 bits of the 64-bit result into the destination register. The following sample illustrates this:

```
mulhw r0,r2,r3 (place the high word result of r2 * r3 into r0)
```

To achieve the same result in C, a shift is required to move the high word result down before it can be assigned. A C representation may look like: `r0 = (r2*r3)>>32;` //the variables are declared as long long

Similarly, our IR tree represents it the same exact way only in three-address form. Notice that a sub-instructional variable with the identifier 1026 is used as the 64-bit representation for the result of the multiplication in the processor's pipeline. We completely synthesized the existence of this temporary variable just for purposes like this. In fact in our PowerPC representation, any variable identifier lower than 1024 is reserved for actual addressable registers on the processor.

```
<ASSIGN>
  <VARIABLE size='64'>var_1026</VARIABLE>
  <MUL>
    <VARIABLE>var_2</VARIABLE>
    <VARIABLE>var_3</VARIABLE>
  </MUL>
</ASSIGN>
<ASSIGN>
  <VARIABLE size='64'>var_0</VARIABLE>
  <BIT_SHIFTRIGHT>
    <VARIABLE>var_1026</VARIABLE>
  </BIT_SHIFTRIGHT >
</ASSIGN>
```

4.3 Simple Translation Optimization

The complete semantics of every instruction can be executed on the physical processor without any overhead. However, when the BTS decodes the instructions there may be lots of extraneous statements generated as a result of side effects or stack maintenance. For example, a comparison operation may compare a value for less than, greater than, and equals against 0. These superfluous operations can be reduced to just the one the subsequent assignment or conditional branch depends upon. As a remedy, we introduced basic optimizations to clean up this bloat from the decoding. The following described optimizations are applied to the IR tree only to ensure that any indicative register usages aren't removed before the representation is smoothed.

The following optimization rules may be followed to remove much of this extra bloat.

The first rule is essentially constant propagation, except that we generalized this to propagation of

any expression. The rule can be stated: If the program makes an assignment to variable X with expression Y, and variable X is subsequently assigned to variable Z, then the expression Y may be directly assigned to variable Z; if and only if there is no writes to variable X or Y, branches, or targets between the two statements, and X is not part of the r-value expression Y in the first statement. The propagation itself cannot violate the three-address form. After the propagation other optimizations will decide whether or not the first statement can be removed or not, because X may be used later on in the program and the X = Y assignment is necessary.

1. If two assignments induce transitivity then propagate the expression.

```
x = y; //may be a target and X is not on the right hand side
... (x is not used, no branches, or targets)
z = x; //not a target
```

Becomes...

```
x = y;
...
z = y;
```

Similarly to the first rule, the second rule may be stated as follows: If the program makes two assignments to a variable X and there are no reads from this variable X and no branches between the two statements then the first one may be removed. An exception to this rule is if second assignment depending on the first, i.e. the l-value is also an operand in r-value in the second assignment.

2. If variables are not read then the last two writes should be reduced to just the last write.

```
x = y; //remove this statement
... (x is not read, no branches)
x = 6;
```

The third rule is likewise similar to the second in that it can be stated: If the program makes an assignment to a variable X and X is not accessed and there is no branch before the flow reaches a return statement, then the statement is useless and should be removed.

3. Eliminating useless assignments invalidated by a return.

```
x = y; //remove this statement
... (x is not read, no branches)
return; //x is not returned
```

The fourth rule being the most basic is very important to clean up the deficiencies of the decoding process. Simply stated: If a function only makes writes to a variable X, then all of these statements should be removed.

4. Variables that are only written to get removed.

```
x = 5; //remove this assignment
```

```
.... (x is not read)
x = 7; //remove this assignment
.... (x is never read)
```

It will be clear in the following sections how useful these basic optimizations can improve the efficiency of the generated code. It is also important to mention our motivation for using such a basic optimization rule set. First and foremost, our system was aimed to be simple and new optimization algorithms was not our focus by any means. But also we made the presumption that many of the optimizations that were applied to the source program at compile time may be carried over through in the translation. In particular, any optimizations that a compiler would do on the source program before it is reduced from its own intermediate representation to register transfer lists or assembly code would be present in our IR Tree. For example, the semantics of loop unrolling should be present in our IR tree as well, because part of the loop code is present outside of the loop itself. If the correctness requirement was properly met then this part of the program, a partially unrolled loop, will also be present in the decoded representation. Constant propagation and other optimizations similar to loop unrolling will also be present in the decoded program because of correctness. Obviously any dead code that was stripped in the original program will also remain absent in the representation, since the BTS is not aware of the original source code.

4.4 Implementation of the Sparc Encoder

To complete a simple prototype system, the SPARC encoder was implemented in a somewhat inefficient fashion of turning all variable reads and writes into memory load and stores. No post IR Tree optimizations, such as peephole optimizations, were implemented. Although, it is possible for the BTS to export the source program back to C and allow gcc or another compiler to run its own optimizations when it gets recompiled. Our implementation outputs SPARC 32- and 64-bit assembly code and relies on the native assembler to package this up into the final object file.

Recall it is the responsibility of the `OutputTree()` function in each processor class to output the final translated program. The `OutputTree()` function must loop through each IR tree node in each function in the representation and generate its resulting assembly code. This loop first checks each node to see if it is a target of a branch, i.e. a join point in the control flow, then its label is printed out. After this the `unsmooth` function is called for the node. The `unsmooth` function may or may not return an entirely new tree to represent the node. For example, an empty return operation probably doesn't need to be unsmoothed at all, and the IR Tree statement will be the same as the architecture-dependent tree. Similar to the decoding phase, the resulting tree nodes are then hashed against a jump table of generating functions. When the hashed function returns, pieces of the assembly output will be placed into specialized strings along with a format identifier. The mnemonic and arguments can then be placed together as the output. This was done because all of the nodes types have their own functions in the hash table, and one function that generates a binary operation will also be calling generating functions for each of its operands.

Necessary operations during the unsmoothing phase include the following. The unsmoothing will turn the function calls and return statements into a series of operations to load arguments before the

control is transferred. If statements will also be fully expanded to its base components of a comparison and then a jump. Most importantly, is the setup and layout of the stack. While the tree is being unsmoothed every variable is assigned an exclusive location on the stack based on a first come first serve basis. A variable's assigned slot is then reused as often as necessary for the entire scope of the program's function. Each and every variable usage will be converted to any necessary load and stores.

4.5 Implementation of Tree Manipulations

4.5.1 ResolveOffsets

The first generic tree operation is ResolveOffsets function. This operation isn't so much a tree manipulation as something that will just augment it. The purpose of this method is to take the offset list and the unresolved offset list from the decode process and link together branches to their target locations using proxy nodes. The ResolveOffsets function also takes a list of external symbols including the function names in the object file, and translates jumps to those locations as function calls.

4.5.2 Simplify

The next manipulation is the simplify operation. The simplify operation is a recursive function that will take appropriate measures to deal with the replacement and removal of nodes in the IR tree that can be removed. We implemented this functionality into the program to allow forgiveness in the decoding process, where any overcomplicated operations may be generated. We also implemented this functionality to remove some of the burden of node removal from other manipulation functions (e.g. the smooth or optimize function).

In more detail, the simplify operation will walk through the tree representation and at each node determine whether or not it can be simplified. One illustrative example of this is: $X = Y \mid Y;$, where the or operation simply does nothing but result in Y. We recognize simple patterns like this and change them appropriately. Another example would be: $X = 0 \ \&\& \ Y;$, where again the statement could be simplified to just an assignment of $X = 0;$.

We also use the simplify function as a cleanup for the smooth function, where the process of removing some nodes is a bit more involved. In our PowerPC smooth method some operations that are part of a functions prologue or epilogue are simply ignored because they will be represented with a simple function node instead. These operations will just have their operands set to NULL so that the simplify process may take care of it later on. We designed the simplify function to recognize invalid operands and remove these nodes from the representation. Likewise, the optimize function will instead mark operations for deletion using meta data.

4.5.3 Smooth

The Smooth function, beyond the PowerPC specific details of recognizing function calls and other semantic units, we see as having a general algorithmic form that can be applied across many different architectures. The methods we used to overcome the hurdles of extracting functions, function calls, and return statements is described as follows.

During the smooth process our variable nodes will represent both memory slots as well as registers. Those two categories both are given their own set of identifiers. The registers are treated as temporary variables used in the program and their identifying variable nodes have an extra meta data flag set to indicate that these are temps, and are to be treated as a different set of variables. The reason why we treat the register representations differently is because at some points during the smooth process they are still considered registers. For example, when a function call is reached our smooth heuristic will look at all the variables that could be filled for possible argument passing. We take the largest set of arguments, that start in the register set and then spill into memory, and place them as being the arguments in the single function call node in the representation.

Actually, much of the smoothing algorithm relies on this set of "temp" variables that represents registers. At the very start of the the Smooth function hash tables are created to store our new set of variables and temps. The temp hash table is initialized with indexes to every register identifier. All of the temps begin by having their new IR tree identifiers set to -1 to indicate it has yet to be used. One exception is that the temps representing the function argument registers are not specified to be temps, they actually remain as variables that are read as parameters to the function being smoothed. Once overwritten with a new value we consider these variables to be temps. Another exception is that registers we are not concerned with should have a NULL variable representing them in the resulting representation. These NULL values are placed into the mentioned hash table. When the smooth process comes across a variable node (representing a register) it replaces it with the new temp node. Any expression with a NULL operand will be removed in the later Simplify function. Registers that have specific purposes for function prologues/epilogues may be potential candidates for the quick NULL cleanup process, allowing other nodes to persist that represent the original program structure.

Memory slots are treated in a similar way and their identifiers are accumulated in a hash table. The goal is to replace all the memory accesses with an abstract variable representation. Like the temps, each address is given a unique identifier to consistently represent it throughout the scope of the function being smoothed.

The primary way of recognizing functions in the smooth process is to find the original stack change in the function prologue. The other less indicative operations in the prologue and epilogue can be eliminated by the previously mentioned NULL variable process or be simply matching other patterns of operations. Other information for the function, the local variable and parameter lists, can be compiled at the end of the smooth process by keeping track of the usages of allocation of our new variables and temps.

We are able to process function calls by maintaining an ongoing queue of potential temps and variables that represent places for argument passing. Similar to discovering function parameters, function arguments are only considered valid if they were not read before their last assignment, since their is nothing to stop a compiler from using in registers as just extra general computation registers. When a function call is encountered the largest contiguous list possible of potential arguments is copied into the new function call node. Return statements likewise work this way only the list is restricted to just a single value.

4.5.4 Unsmooth

The Unsmooth function also has commonalities across all architectures. Assuming again that we are restricting it to just an inefficient load and store implementation, the memory slot allocation in particular can be reused. Initially, for most architectures, when emitting a function the stack must first be allocated and this requires counting all of the local variables. On processors such as the PowerPC or SPARC, that have a relatively static stack setup, also count the number of overflowed arguments for its function calls. Once the stack slots are established, we predict that they can be allocated on a similar first-come, first-serve basis as our SPARC implementation. Keeping track of which slots were given to which variable identifier is as simple as using a hash table. We implemented the keying into the hash tables by the variable's identifier and the value as the byte offset of the slot from the original stack pointer location of 0, since we don't know or really care of the real address of the stack pointer when it is executed. This will result in negative but consistent values that can easily be converted to offsets from the current stack pointer offset in the implementation. Additionally, the Unsmooth function will always look at just one operation at a time and return the same representation or a newer unsmoothed version that may have a longer sequence of nodes.

4.6 Sample Trees

To illustrate just how the translation works a look at simple translations is necessary. Provided below are illustrations of sample decodings and how the tree nodes are built from the machine code. The architecture-dependent representations are presented in XML-like format for clarity. In the following sample provided below, the instruction move register is a pseudo mnemonic which copies one register to another. In this case r1, the stack pointer, is being copied into r30 as part of a function prologue in our test environment. The move register instruction is actually a bitwise or instruction and its opcode reflects this. Also note that once our tree is simplified using the simplify operation the bitwise or operation will realize that it is a completely useless operation and should be replaced with a direct assignment instead.

```
mr r30,r1 (set r30 to the contents of r1)
```

```
<ASSIGN>
  <VARIABLE>var_30</VARIABLE>
  <BIT_OR>
    <VARIABLE>var_1</VARIABLE>
    <VARIABLE>var_1</VARIABLE>
  </BIT_OR>
</ASSIGN>
```

This next decoding example illustrates how quickly useless code can build up in our representation. A compare operation on the PowerPC will set 3 status bits via 3 different types of comparisons

```
cmpw cr7,r0,r2 (compare register r0 and r2 and put the results into compare register 7)
```

```
<ASSIGN>
  <VARIABLE>var_92</VARIABLE>
  <LESS_THAN>
    <VARIABLE>var_0</VARIABLE>
```

```

        <VARIABLE>var_2</VARIABLE>
    </LESS_THAN>
</ASSIGN>
<ASSIGN>
    <VARIABLE>var_93</VARIABLE>
    <GREATER_THAN>
        <VARIABLE>var_0</VARIABLE>
        <VARIABLE>var_2</VARIABLE>
    </GREATER_THAN>
</ASSIGN>
<ASSIGN>
    <VARIABLE>var_94</VARIABLE>
    <EQUAL>
        <VARIABLE>var_0</VARIABLE>
        <VARIABLE>var_2</VARIABLE>
    </EQUAL>
</ASSIGN>
<ASSIGN>
    <VARIABLE>var_95</VARIABLE>
    <VARIABLE>var_128</VARIABLE>
</ASSIGN>

```

The subsequent branch instruction shown below, only uses one of the flags set in the previous compare operation.

`ble cr7,L2` (branch to L2 if less-than or equal (i.e. not greater than) (the labels L2 and `jmp0` are the same))

```

<IF>
    <CONDITION>
        <EQUAL>
            <VARIABLE>var_93</VARIABLE>
            <CONSTANT>0</CONSTANT>
        </EQUAL>
    </CONDITION>
    <THEN>jmp0</THEN>
</IF>

```

4.7 Sample Translations

Now let's look at an entire translation and not just sample decodings. The following code sample is completely trivial, but is necessary to fully solidify the understanding of the BTS. The code snippet below will create a function that takes two arguments and returns a single result from adding the two arguments together.

```

int add(int x, int y)
{
    return x + y;
}

```

A pass through gcc without any optimizations yields the following PowerPC assembly code. Comments have been added for clarity.

```
.section __TEXT,__text,regular,pure_instructions
.section __TEXT,__picsymbolstub1,symbol_stubs,pure_instructions,32
.machine ppc
.text
.align 2
.globl _add
_add:
    stmw r30,-8(r1)
    stwu r1,-48(r1) # create the stack frame
    mr r30,r1
    stw r3,72(r30) # store the incoming arguments
    stw r4,76(r30)
    lwz r2,72(r30) # load the arguments up for use
    lwz r0,76(r30)
    add r0,r2,r0 # perform the addition
    mr r3,r0 # place the result into the outgoing return register
    lwz r1,0(r1) # restore the stack frame
    lmw r30,-8(r1)
    blr # return via the link register
.subsections_via_symbols
```

Notice how the gcc output produces relatively inefficient code without any optimizations, resorting to load and stores. These inefficiencies will likewise be present when the function is decoded. The following shows the tree after decoding the add function from its object file. Notice that all of the loads and stores are still in place.

```
<ASSIGN>
  <ADDRESS>
    <BASE>
      <VARIABLE>var_1</VARIABLE>
    </BASE>
    <INDEX>
      <CONSTANT>-8</CONSTANT>
    </INDEX>
  </ADDRESS>
  <VARIABLE>var_30</VARIABLE>
</ASSIGN>
<ASSIGN>
```

```

<ADDRESS>
  <BASE>
    <VARIABLE>var_1</VARIABLE>
  </BASE>
  <INDEX>
    <CONSTANT>-4</CONSTANT>
  </INDEX>
</ADDRESS>
<VARIABLE>var_31</VARIABLE>
</ASSIGN>
<ASSIGN>
  <ADDRESS>
    <BASE>
      <VARIABLE>var_1</VARIABLE>
    </BASE>
    <INDEX>
      <CONSTANT>-48</CONSTANT>
    </INDEX>
  </ADDRESS>
  <VARIABLE>var_1</VARIABLE>
</ASSIGN>
<ASSIGN>
  <VARIABLE>var_1</VARIABLE>
  <ADD>
    <VARIABLE>var_1</VARIABLE>
    <CONSTANT>-48</CONSTANT>
  </ADD>
</ASSIGN>
<ASSIGN>
  <VARIABLE>var_30</VARIABLE>
  <BIT_OR>
    <VARIABLE>var_1</VARIABLE>
    <VARIABLE>var_1</VARIABLE>
  </BIT_OR>
</ASSIGN>
<ASSIGN>
  <ADDRESS>
    <BASE>
      <VARIABLE>var_30</VARIABLE>
    </BASE>
    <INDEX>

```

```

        <CONSTANT>72</CONSTANT>
    </INDEX>
</ADDRESS>
    <VARIABLE>var_3</VARIABLE>
</ASSIGN>
<ASSIGN>
    <ADDRESS>
        <BASE>
            <VARIABLE>var_30</VARIABLE>
        </BASE>
        <INDEX>
            <CONSTANT>76</CONSTANT>
        </INDEX>
    </ADDRESS>
    <VARIABLE>var_4</VARIABLE>
</ASSIGN>
<ASSIGN>
    <VARIABLE>var_2</VARIABLE>
    <ADDRESS>
        <BASE>
            <VARIABLE>var_30</VARIABLE>
        </BASE>
        <INDEX>
            <CONSTANT>72</CONSTANT>
        </INDEX>
    </ADDRESS>
</ASSIGN>
<ASSIGN>
    <VARIABLE>var_0</VARIABLE>
    <ADDRESS>
        <BASE>
            <VARIABLE>var_30</VARIABLE>
        </BASE>
        <INDEX>
            <CONSTANT>76</CONSTANT>
        </INDEX>
    </ADDRESS>
</ASSIGN>
<ASSIGN>
    <VARIABLE>var_0</VARIABLE>
    <ADD>

```

```

        <VARIABLE>var_2</VARIABLE>
        <VARIABLE>var_0</VARIABLE>
    </ADD>
</ASSIGN>
<ASSIGN>
    <VARIABLE>var_3</VARIABLE>
    <BIT_OR>
        <VARIABLE>var_0</VARIABLE>
        <VARIABLE>var_0</VARIABLE>
    </BIT_OR>
</ASSIGN>
<ASSIGN>
    <VARIABLE>var_1</VARIABLE>
    <ADDRESS>
        <BASE>
            <VARIABLE>var_1</VARIABLE>
        </BASE>
        <INDEX>
            <CONSTANT>0</CONSTANT>
        </INDEX>
    </ADDRESS>
</ASSIGN>
<ASSIGN>
    <VARIABLE>var_30</VARIABLE>
    <ADDRESS>
        <BASE>
            <VARIABLE>var_1</VARIABLE>
        </BASE>
        <INDEX>
            <CONSTANT>-8</CONSTANT>
        </INDEX>
    </ADDRESS>
</ASSIGN>
<ASSIGN>
    <VARIABLE>var_31</VARIABLE>
    <ADDRESS>
        <BASE>
            <VARIABLE>var_1</VARIABLE>
        </BASE>
        <INDEX>
            <CONSTANT>-4</CONSTANT>

```

```

        </INDEX>
    </ADDRESS>
</ASSIGN>
<RETURN></RETURN>

```

Recall, the next step in our process is to take the architecture-dependent tree and smooth it into something that be easily retargeted. The resulting IR Tree can then be represented using C code; the following output of the unoptimized IR Tree looks like the following:

```

int add(int var_4, int var_6)
{
    int tmp_0, tmp_1, tmp_2, tmp_3, tmp_4;
    int var_0, var_1, var_2, var_3, var_5, var_7, var_8;
    var_3 = var_4;
    var_5 = var_6;
    tmp_2 = var_3;
    tmp_3 = var_5;
    tmp_3 = (tmp_2 + tmp_3);
    tmp_4 = tmp_3;
    tmp_0 = var_7;
    tmp_1 = var_8;
    return (tmp_4);
}

```

After applying our translation optimization rules that we established, the IR Tree is then reduced to just the following:

```

int add(int var_4, int var_6)
{
    int tmp_4;
    tmp_4 = (var_4 + var_6);
    return (tmp_4);
}

```

The unsmooth process is then used to take the above optimized tree and transform it into SPARC code as presented below:

```

.section ".text"
.align 4
.global add
.type add, #function
.proc 04
add:
    .register %g2, #scratch

```

```

.register %g3, #scratch
save %sp, -104, %sp ! create the stack frame
mov %i0, %g2 ! copy arg 1 (var_4) into a temporary register
mov %i1, %g3 ! copy arg 2 (var_6) into a temporary register
add %g2, %g3, %g1 ! perform the addition
st %g1, [%fp + -8] ! store the result into the space for tmp_4
ld [%fp + -8], %i0 ! load tmp_4 into the return space
ret ! delayed return to the caller
restore ! restore the stack frame
.size add, .-add

```

5 Programming with the BTS Framework

The Binary Translation System framework was primarily designed to provide reusable code for creating newer decoders and encoders. In order to develop support for other architectures in the BTS one needs to be intimately familiar with particular machine architecture or prepared to dive into a programmer's manuals for the architecture of interest. Much of the decoding needs to be manually scripted into a C++ based framework in the BTS. The basic requirements for this are to extend the AbstractProc class that is responsible for encoding and decoding a particular architecture.

The required interface functions for the AbstractProc include GetNextInstruction, GetOffsetList, GetUnresolvedOffsetList, OutputTree, and Smooth and Unsmooth. These functions highlight many of the steps that the BTS goes through while making a translation. GetNextInstruction is used to retrieve the next instruction and decode it into the architecture-dependent tree form. When a code section has been exhausted, the decoder must also report the section's relative offsets for each instruction as well as the offsets for the instructions that have unresolved relative references to somewhere else in the code. These offsets are reported through the GetOffsetList and GetUnresolvedOffsetList functions respectively. If the processor is going to be a back-end encoder, then implementing the Unsmooth and OutputTree functions are required. The Unsmooth function may actually be skipped however, because the only external function should be the OutputTree function, but conceptually, an unsmooth operation is required to actually make the IR Tree an architecture-dependent tree.

5.1 Statistical Analysis

One method we found effective during development was to gather statistics on the usage of instructions in what may be actual translatable programs. The relative frequency of each instruction, bi-gram, tri-gram, and 4-gram, 5-gram were gathered to predict common usage. We gathered the data from a group of common unix programs for the PowerPC architecture. The programs used were: cat, dd, expr, ls, pwd, chmod, df, hostname, mkdir, rm, cp, domainname, kill, mv, rmdir, date, echo, ln, ps, sleep. The following shows frequencies from single instructions and groups of three instructions.

The single instruction as shown in Figure 6 were quite revealing, as they show that the most common instruction usage was with some of the most simple instructions. These results helped us to prioritize which instructions should be supported first in the development cycle, and supported the notion of

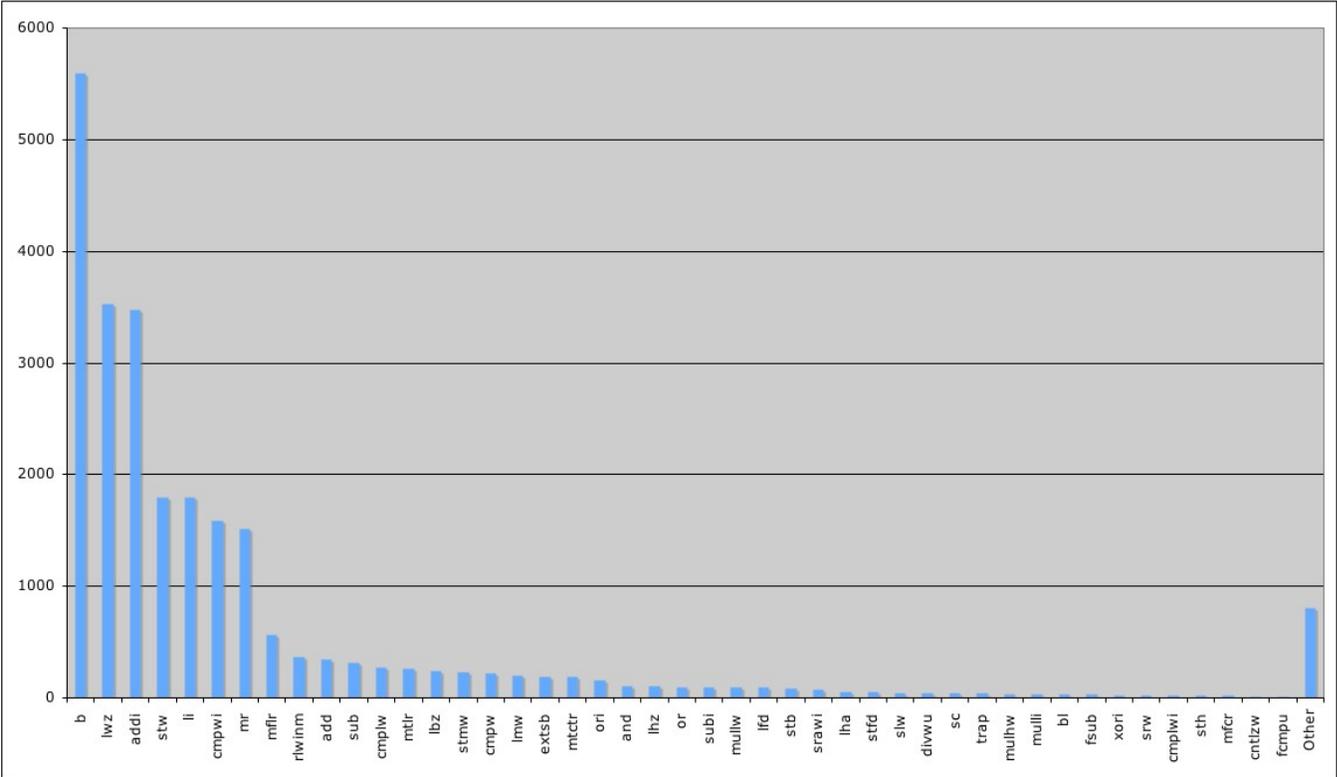


Figure 6: PowerPC instruction frequency.

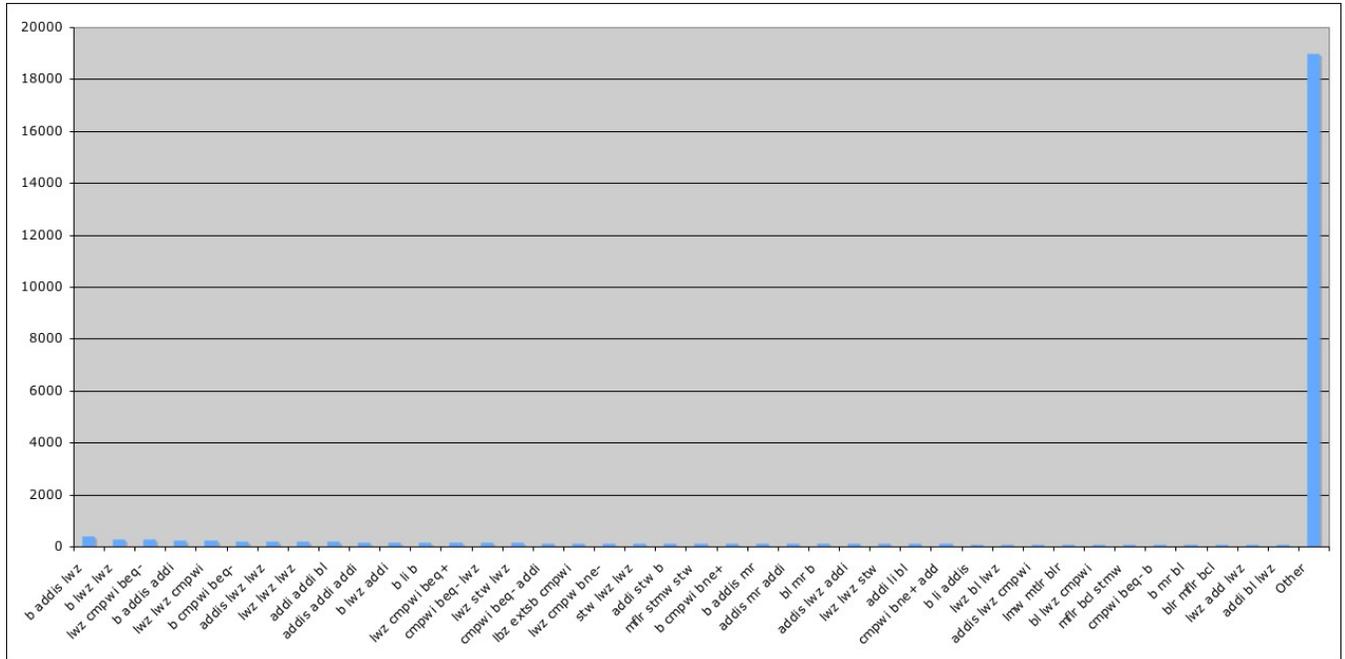


Figure 7: PowerPC tri-gram instruction frequency.

supporting simple instructions first to work out any bugs.

The relative frequencies of groups of instructions, on the other hand, were not conclusive enough to draw any useful conclusions from. The results in the tri-gram analysis in Figure 7 were not indicative of expected gcc compilation patterns. For example, we expected a short sequence of: stmw, stwu, mr for function prologues, but this is nowhere to be seen at all in the top tri-grams. In fact, no tri-grams strike us as being relatively common enough to be of any significance. Perhaps the culprit was that these programs were heavily optimized without many patterns to discover, while we would see very characteristic sequences of instructions when invoking gcc without optimizations.

5.2 Difficulties in Binary Translation

With implementing a binary translation system there is a great deal of inter-dependencies and how coding one architecture is going to affect coding another, since it is largely a task of generalizing a superset representation across many ISAs. Applying a single change to accommodate a feature in a particular ISA will likely change the representation tree. This change may then resonate throughout the rest of the framework as well, wherever the specifics of the IR tree are handled. Decoupling parts of the process from the IR tree therefore was the part that required the most cleverness. The main way we accomplished this decoupling was to have some of the tree manipulation functions apply to the base class GenericNode, and not the higher level ProcNode.

Aside from architectural design issues there are also many problems that arise when attempting to do binary translation. Again thinking in over-simplified terms these problems are the same problems decompilers and compilers face. The following is just a subset of issues a universal binary translator

may encounter.

Endianness is the first issue a binary translator must deal with. Reading from the object file requires proper byte-swapping to be in place in order to load information correctly. The BTS implements macros to swap 2-byte and 4-byte words, as well as functions to swap arrays of words. Other non-standard word sizes may need to be supported, but these can easily be dealt with in a similar manner.

It is also important that the translation system recognize all types of expected patterns for functions and other semantic units. For example, making presumptions about how the stack is setup may be a fatal mistake if the system is asked to translate a program that has an uncommon stack setup sequence. In this regard, the BTS only makes one presumption about the setup of the PowerPC stack, and that is that the general purpose register r1 is used as the stack pointer. This is a fairly safe presumption to make however, since the use of r1 as the stack pointer is stated in the Motorola's PowerPC architecture manual[43] not in the context of an ABI specification. This rule is the only rule that the BTS uses to find a function in PowerPC code, where the addition of a constant to r1 signifies the information about a function. In this respect, the BTS will not recognize leaf procedures as functions, but will be able to translate their code nevertheless.

More issues when dealing with stack frames include the various Application Binary Interface (ABI) specifications that can be created with any given architecture. ABI issues include how variables are passed and returned, memory alignment, stack layout, and some of the following basic questions. What happens when all the arguments to a function cannot be passed through registers? For the PowerPC architecture, the general purpose registers r3 - r10 can be used to pass 8 words to a function, if more are needed then the calling function must place them on its own stack frame. Without a function definition how can you detect what arguments to expect? The BTS makes an educated guess at this by the types of memory and register accesses a function makes. What happens to the stack when the function being translated calls another function? In one of its passes over the IR Tree, the BTS takes note of the largest amount of data a function call requires. This is then used to add extra slots to the stack in the event the arguments cannot be passed via registers. What happens when the ABI specifies 8-byte alignment? Stack alignment is simple and can easily be taken into account in the output when the size of the stack is created, the size simply needs be increased until it is divisible by 8. What happens when the byte-alignment of data structures varies? This is also a very important issues to be concerned about. The BTS does not currently take into account byte-alignment of data structures. What about functions that receive a variable number of arguments? The implementation of variable arguments needs to be accounted for in all forms, it is yet one more thing that needs to be dealt with aside from the semantics of each instruction. The BTS does not handle variable arguments.

What about floating point instructions? The concept of floating point variables is very old and if they need to be emulated on the output then there already exists compilation methods for doing this. The BTS supports basic PowerPC floating point variables, but our prototype system does not actually make it a point of testing them.

Static data in the object file is also important for completing the translation. It is important that the data be properly byte-swapped if necessary and output and linked in its entirety. There may also be instances where static data is used as executable code. In these instances it is up to the translation writer to decide whether or not to translate any information in the data sections. One such instance

is when indirect jump tables are used for switch cases or other means. Identifying switch cases alone can be a large task all its own, where the patterns can vary depending on the complexity of the switch statement.[5]. The BTS does not handle any issues relating to static data in the object file, we focus largely on the translation of the instructional semantics.

Similarly, a translation writer may also encounter more exotic forms of code. Such as, code that modifies itself on-the-fly, and obfuscated code[48][47]. The BTS makes no attempts to handle obfuscated code of any sort, but simple obfuscated such as abusive NOPs or JUMPs may be easily simplified using tree manipulations.

Perhaps the most important difficulty in binary translation is the various API and system calls that need to be mapped if the operating system environments will also be different.

6 Evaluations

The evaluation of our system was done across a small suite of programs that demonstrate common functionality. Since our prototype was created to translate PowerPC code into SPARC code we only tested it against comparative technologies for these architectures. The PowerPC testing environment and source for compiled object files was OS 10.4 running on a Power Mac G4 with dual 500MHz G4 processors. The SPARC environment was Solaris 9 running on an Ultra 10 Workstation with a 333MHz UltraSPARC III processor.

6.1 Metrics

6.2 Existing Technologies

Our testing evaluated the following solutions. Portable source code was used and compiled on both PowerPC and SPARC architectures. Java Byte code was used to test the JVM translations on both PowerPC and SPARC architectures. The boomerang decompiler was used to decompile PowerPC object files back into C code, and then recompiled on the SPARC architecture. Unfortunately, we did not evaluate the UQBT because they did not readily support a PowerPC to SPARC translation.

The straight gcc compilations performed as expected, with times decreasing as optimizations increase. The Java virtual machine tests also performed as expected, with times not much slower than native code, because the programs were so simple. What was surprising however, was the decompiler results. Only the most simple fibonacci calculator could be successfully recompiled with negligible manual changes. All of the other tests produced decompilations that were beyond recognition, and could not be fixed manually.

6.3 Translations with the BTS

BTS also performed as expected with acceptable performance for a prototype system. The speeds in all the test except for the fib test were close to the Java performance. The fib test in particular is very sensitive to extra instructions, etc. since it is heavily recursive. Granted that our optimization algorithm was not very sophisticated and the exhaustive load/store methods used in emitting SPARC code the BTS has great potential.

	fib(40)	gcd	negdouble	selectsort	floattest
PowerPC Native gcc 3.3					
gcc -O0	26.2	0.00000038	0.00000016	0.0000027	0.00000382
gcc -O1	10.45	0.0000002	-	0.00000066	0.00000242
gcc -O2	10.12	0.0000002	-	0.00000054	0.0000017
gcc -O3	8.56	0.0000002	-	0.00000054	0.0000017
SPARC Native gcc 3.42					
gcc -O0	43.43	0.000001620	0.0000002	0.000118	0.00000608
gcc -O1	17.03	0.00000118	0.00000004	0.000026	0.00000316
gcc -O2	18.69	0.00000114	0.00000004	0.000022	0.00000316
gcc -O3	18.65	0.00000124	0.00000002	0.000024	0.00000316
JavaVM on PowerPC					
	12.114	0.000000306	0.000000058	0.000025718	0.000003168
JavaVM on SPARC					
	15.388	0.000001016	0.000000064	0.000051936	0.00002403
Boomerang Decompiler (PPC) to gcc -O0 (SPARC)					
	50.48	-	-	-	-
Binary Translation System (PowerPC to SPARC)					
	122.45	0.00000224	0.00000044	0.000742	-

Table 1: Performance Metrics in Seconds

It should also be noted that the BTS was the only solution that could statically translate PowerPC to SPARC, aside from the decompiler, which could only work on an even smaller subset of programs. In our test suite, the BTS could only not translate the floattest because it does not currently support floating point instructions.

6.4 Blind Tests

We depended on the above test cases to debug our system. Therefore, in fairness to the other static method we tested, the boomerang decompiler, we also conducted blind tests to evaluate if our system would be accurate for general code. The first test was a CRC16 calculation, that involved calculating a checksum on a buffer using lots of bitwise operations. Our system was able to translate the code, but it did not provide the correct results when run. Boomerang was not able to provide any legible or recompilable code. The second blind test was a number to comma-formatted string function. Likewise, our system produced a result but was incorrect when run. Boomerang did not provide legible or recompilable code. The third test was an easter month and day calculator. This function used data structures, which we do not currently support in the BTS. The boomerang decompiler produced legible results, but were incomplete. The last test was a random number generator. This function made use of static variables to generate a series of random numbers. The BTS currently does not support static variables, and could not produce a result that was correct. Boomerang once again produced illegible and non-compilable code.

We expected better results than a complete series of failures for both systems, but at least it shows that the chosen decompiler did not fail in the first set of tests due to our choice of test code. It also shows that we need much more work in debugging test cases to get a good general system for decompiling PowerPC code. We predict that the more test cases that are fully debugged will generalize the system for decompiling PowerPC and other architectures.

7 Conclusions

Overall, we feel that we achieved positive results. We also still feel that with more work this system could support many of our original predictions of generality, speed, and extensibility. While the Binary Translation System is only prototype, we did manage to validate some of our goals and requirements with both proven results or potential. The following highlights how well we did and did not meet our goals and requirements.

Explore and implement the minimum requirements for a prototype universal binary translator. We originally proposed the minimum requirements for a universal static binary translator and highlighted them in section 2.1. After implementing this system, we still feel that those should remain the minimum requirements. The BTS in its current state attempts to meet the requirements, but does have flaws. We will highlight these flaws shortly.

Document the specific methods implemented and used in this prototype. This thesis did document the specific methods implemented and techniques they employ. Detailed information is not given, but conceptually someone should be able to reproduce these methods with a clear understanding of the documentation in this thesis.

Establish a working generalizable framework with those methods that can be useful for other architectures. Our prototype system does this as best it can short of supporting other architectures. While many of the methods presented can be used for any of the common architectures, it cannot be verified until they are actually tested. We predict that some system wide changes would have to be made to support other types of architectures such as X86, but these changes would quickly become less unnecessary as more support is added and the superset of processor logic becomes larger in our representation.

Evaluate the system quantitatively for speed and performance. Tests were conducted and presented in the prior section. There are two main bottlenecks and flaws to the performance of the BTS prototype. The first is the encoding of the SPARC output. If we implemented more sophisticated register allocation and peephole optimization algorithms we think that the performance would be greatly improved. However, our focus was to not re-implement existing technologies. The second flaw is also one of the main design features of our system. Using our framework to support additional architectures, especially in its current state, may require changes to the framework itself. Although, we also suspect that any other generalizable system also suffers from this flaw, regardless if its framework-based.

Determine how this specific system compares to the quality of other systems that provide comparable solutions to software portability. We conducted tests comparing the BTS prototype to other existing solutions. The most important requirement for software portability is completeness and correctness. Java and portable C both support these requirements much better than the BTS does in its current state. Therefore, the BTS prototype currently would not be a good choice for establishing code portability.

Hypothesize if the prototype system demonstrates potential to be universally applicable across all common architectures. Our fulfillment of this goal can essentially be stated in how well we met our established requirements, and how much better could we potentially do. The first two requirements of completeness and correctness were not completely satisfied in our prototype. The PowerPC decoder did not support floating point instructions, data structures and other concepts we pointed out previously. Of the instructions and concepts we did support they were complete, however as the blind tests show are not yet correct in all cases. The third requirement of universal adaptability again was met only as much as needed to implement the PowerPC decoder. As more architectures are supported, system-wide changes may be necessary to account for newer concepts or semantics. How much change would the framework potentially need? We reviewed manuals from the PowerPC[43], SPARC[44], and MIPS[45], and saw we would be able to fully support all of these RISC-based architectures with the current framework. The X86 architecture however may need more changes to successfully support. For example, The rotate through carry operations (RCL, RCR) will do a bitwise rotation using the carry flag as part of the value. While we have support for an optional extra buffer in our rotate representations, there is no mechanism in the code for dealing with this optional buffer. The magnitude of the changes to support this instruction would then be dependent upon if the optional buffer concept can be smoothed over before the IR tree is created. If the IR tree must deal with this concept then the changes would have to be propagated throughout the system. The last requirement of transparency we feel is inherent to a system that can fully meet the other three requirements, because translations could be done completely without user intervention in the background given correct output. Since the BTS does not fulfill all of the requirements perfectly, it could not be run transparently for all translations. While the BTS does not completely fulfill all of our requirements and is currently not the best choice for a solution for

software portability, we remain optimistic that it someday could be.

In contrast, the UQBT system in test runs proved to be successful, and it appears to be offering a very viable solution for translating machine code statically. Their design and implementation however has some debatable flaws in a few key areas. The system may be too complicated and prove to be difficult when supporting arbitrary computing architectures. The design of the system, specifically the specification language centric aspect, may actually require more effort to add support than it would be to specify a decoder using our system. However, only more work and comparative tests will be able to fairly conclude on that hypothesis.

We learned from this experience that binary translation is as much an art as it is a science; practicality needs to be used as much as idealism to get good results. We also believe after these experiences, that universal static binary translation is a great candidate for an AIC solution because of the following reasons. The BTS does not have to meet any strict on-demand performance requirements, this allows it more time to create better optimized code, and keep a very modular framework design that does not have to be convoluted for the sake of speed. The BTS is capable and has proven to provide the framework to represent many common machine code semantics. The BTS provides layers of object abstractions to promote code reuse. The BTS framework could be adapted to create dynamic translators or other binary manipulation systems.

7.1 Side Effects

One side effect of these efforts is that the output target could easily be C or other similar languages that can represent the IR tree, and actually create a semantically correct decompilation of the source program.

Another side effect is that the IR tree provides an excellent platform for binary program manipulation at the IR Tree abstraction. While this may be possible with decompilers, my experience has been that decompilers focus on information extraction opposed to correctness. So while it is theoretically possible to manipulate programs using decompilers we have not seen any application that could. Since the target architecture is arbitrary using the BTS it is possible to translation the changes back to its original architecture or to a completely new one. Potential uses for this could include automated security patches or software grafting, where various pieces of code are used to change the behavior of the source program.

7.2 Future Work

The many potential spin-offs and immediate applications of universal binary translation technology we feel are very exciting. Whether programming using the BTS framework or your own, the advancements in this area could provide innovative solutions for many problems.

Building advanced machine code representations could help to make more sensible decompilations, and also more advanced program manipulations and analyzers. On great area for advanced analysis and modification is computer security. Program analysis and manipulations have been used in the past to expose and close up vulnerabilities such as buffer overflow bugs.[35] We also foresee these types of machine code representations to assist in automated security audits.

Another future application is applying the BTS framework or similar system to retarget the original architecture of the source program to do post-compilation optimizations. Our framework would be capable of implementing optimizations on the abstract representation as well as the specific machine code. The translation process could be used to target variants of an architecture, 64-bit or to shift computation to vector co-processors.

Going a bit further down the road would be to introduce developer and then user-level tools for splicing programs and their functionalities together, permitting true extensibility for anything.

8 References

References

- [1] Charnoff, et al. "FX32! A Profile-Directed Binary Translator". DEC.
- [2] J. Hayes, W.G. Griswold, S. Moskovics. "Component Design of Retargetable Program Analysis Tools that Reuse Intermediate Representations," Proceedings of 2000 International Conference on Software Engineering (ICSE 2000), June 2000.
- [3] Fernandez and Espasa. "Dixie: A Retargetable Binary Instrumentation Tool". Universitat Politècnica de Catalunya-Barcelona.
- [4] C Cifuentes, M Van Emmerik, and N. Ramsey. "The Design of a Resourceable and Retargetable Binary Translator". Proceedings of the Sixth Working Conference on Reverse Engineering, Atlanta, USA, October 1999, IEEE-CS Press, pp 280-291.
- [5] Cristina Cifuentes and Mike Van Emmerik. "Recovery of Jump Table Statements from Binary Code", Department of Computer Science and Electrical Engineering, The University of Queensland, Technical Report 444, December 1998.
- [6] C. Cifuentes and S. Sendall. "Specifying the semantics of machine instructions". In Proceedings of the International Workshop on Program Comprehension, pages 126– 133, Ischia, Italy, 24–26 June 1998. IEEE CS Press.
- [7] C. Cifuentes and Doug Simon. "Procedural Abstraction Recovery from Binary Code", Technical Report 448, Department of Computer Science and Electrical Engineering, The University of Queensland, Sep 1999.
- [8] C. Cifuentes, Et. Al. "The University of Queensland Binary Translator (UQBT) Framework", The University of Queensland, Sun Microsystems, Inc.
- [9] C. Cifuentes, M. Van Emmerik, N. Ramset, and B. Lewis. "Experience in the Design, Implementation and use of a Retargetable Static Binary Translation Framework", Sun Microsystems Laboratories, Technical Report TR-2002-105, January 2002.
- [10] Cristina Cifuentes and Norman Ramsey. "A Transformational Approach to Binary Translation of Delayed Branches with Applications to SPARC and PA-RISC Instruction Set", Sun Microsystems Laboratories, Technical Report TR-2002-104, January 2002.

- [11] Altman, E.R., Ebcioğlu, K., Gschwind, M., Sathaye, S. "Advances and Future Challenges in Binary Translation and Optimization", Proceedings of the IEEE, Vol. 89, Issue 11, Pages. 1710-1722, Nov. 2001.
- [12] Wilson C. Hsieh, Dawson R. Engler, and Godmar Back. "Reverse-Engineering Instruction Encodings", 2001 USENIX Annual Technical Conference.
- [13] C. Collberg. "Reverse interpretation + mutation analysis = automatic retargeting", In Proc. of PLDI '97, June 1997.
- [14] Ebcioğlu, K. and Altman, E.R. "DAISY: Dynamic Compilation for 100% Architectural Compatibility", IBM T. J. Watson Research Center, Yorktown Heights, NY, Res. Rep. RC20538, 1996.
- [15] J.R. Larus and E. Schnarr. "EEL: Machine-Independent Executable Editing", SIGPLAN Conference on Programming Languages, Design and Implementation, Pages 291-300, June 1995.
- [16] David Larochelle and David Evans. "Statically Detecting Likely Buffer Overflow Vulnerabilities", 2001 USENIX Security Symposium, Washington, D.C.
- [17] David Evans and David Larochelle. "Improving Security Using Extensible Lightweight Static Analysis", IEEE Software, Jan/Feb 2002.
- [18] Richard Sites, Anton Chernoff, Matthew Kirk, Maurice Marks, Scott Robinson. "Binary Translation", Digital Technical Journal, Vol. 4 No. 4, Special Issue 1992.
- [19] Mark Probst. "Dynamic Binary Translation", UKUUG Linux Developer's Conference 2002.
- [20] C Cifuentes, B Lewis and D Ung. "Walkabout - A Retargetable Dynamic Binary Translation Framework", Fourth Workshop on Binary Translation, Sep 22, 2002, Charlottesville, Virginia.
- [21] Tao Group Limited. "intent 2 Overview". White Paper.
- [22] Rusty Lee. "VORTOS: Versatile Object-Oriented Real-Time Operating System". MIT Libraries, Cambridge, MA, 2001.
- [23] Bryce Denney. "Bochs Documentation Manual". [online] Available: <http://bochs.sourceforge.net>.
- [24] "Boomerang machine code decompiler". [online Availabel:]<http://boomerang.sourceforge.net>.
- [25] C Cifuentes. "Reverse Compilation Techniques". Queensland University of Technology, Department of Computer Science, PhD thesis. July 1994.
- [26] James Gosling and Henry McGilton. "The Java Language Environment". White Paper.
- [27] "GCC online documentation" [online] Available: <http://gcc.gnu.org/onlinedocs/>.
- [28] "Perl Documentation" [online] Available : <http://perldoc.perl.org/>.
- [29] Giampiero Caprino. "Reverse Engineering Compiler". [online] Available: <http://www.backerstreet.com/rec/rec.htm>.
- [30] Norman Ramsey and Mary Fernandez. "The New Jersey Machine-Code Toolkit". Proceedings of the 1995 USENIX Technical Conference, New Orleans, LA, January 1995, pp 289-302.
- [31] Ron Cytron, Et. al. "Effeciently Computing Static Single Assignment Form and the Control Dependence Graph", IBM Research Division and Brown University.

- [32] Andrew S. Tanenbaum, Et. al. "A practical tool kit for making portable compilers", September 1983, Communications of the ACM, Volume 26 Issue 9.
- [33] Steven Pemberton and Martin Daniels. "Pascal Implementation: The P4 Compiler and Interpreter", <http://homepages.cwi.nl/~steven/pascal/book/>.
- [34] C. Consel and F. Noel. "A general approach for run-time specialization and its application to C", In Proc. of 23rd POPL, St. Petersburg, FL, Jan. 1996.
- [35] Christopher Dahn and Spiros Mancoridis. "Using Program Transformation to Secure C Programs Against Buffer Overflows", 10th Working Conference on Reverse Engineering, Victoria, B.C., Canada, November 2003.
- [36] John Bunda, Terence Potter, and Robert Shadowen. "PowerPC Microprocessor Developer's Guide", Sams Publishing, Indianapolis, Indiana, 1995.
- [37] J.R. Cordy, "TXL - A Language for Programming Language Tools and Applications", Proc. LDTA 2004, ACM 4th International Workshop on Language Descriptions, Tools and Applications, Barcelona, April 2004.
- [38] Stelios Sidiroglou and Angelos D. Keromytis. "Countering Network Worms Through Automatic Patch Generation", Columbia University technical report CUCS-029-03, New York, NY, November 2003.
- [39] S. Mancoridis, Et. al. "Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures", Drexel University and AT&T Labs Research.
- [40] D. Jackson and K. Sullivan. "COM Revisited: Tool-Assisted Modeling and Analysis of Complex Software Structures", ACM SIGSOFT International Symposium on the Foundations of Software Engineering, Nov. 2000.
- [41] Daniel Jackson. "Alloy: A Lightweight Object Modelling Notation", ACM Transactions on Software Engineering and Methodology (TOSEM) Volume 11, Issue 2 , pp. 256-290, April 2002.
- [42] Julian Smart, Robert Roebling, Vadim Zeitlin, Robin Dunn, et al. "wxWidgets 2.6.2: A portable C++ and Python GUI toolkit", September, 2005. [online] Available: <http://www.wxwidgets.org/>.
- [43] "PowerPC Programming Environments Manual For 32-Bit Implementations of the PowerPC Architecture", Motorola, Inc., 2001.
- [44] "The SPARC Architecture Manual Version 8", SPARC International Inc., Menlo Park, California, 1992.
- [45] "MIPS Assembly Language Programmer's Guide", Silicon Graphics, Inc., Mountain View, CA, 1992.
- [46]
- [47] Eldad Eilam. "Reversing: Secrets of Reverse Engineering", Wiley Publishing, Inc., Indianapolis, IN, 2005.
- [48] C. Collberg, C. Thomborson, D. Low. "A Taxonomy of Obfuscating Transformations", Department of Computer Science, The University of Auckland, Technical Report 148.
- [49] Apple Rosetta [online] Available: <http://www.apple.com/rosetta/>.

9 Appendix

9.1 fib.c

Fibonacci Function:

```
int fib(int x)
{
    if (x > 1)
        return (fib(x - 1) + fib(x - 2));
    else return (x);
}
```

9.1.1 Decompiled C Source

```
int fib(int var_6)
{
    int tmp_0;
    int tmp_1;
    int tmp_5;
    int tmp_8;
    int tmp_10;
    int var_5;
    int var_7;
    var_5 = var_6;
    tmp_0 = var_6;
    tmp_5 = (var_6 > 1);
    tmp_8 = (tmp_5 == 0);
    if (tmp_8)
        goto jmp1;
    tmp_10 = (var_5 + -1);
    tmp_10 = fib(tmp_10);
    tmp_1 = tmp_10;
    tmp_10 = (var_5 + -2);
    tmp_10 = fib(tmp_10);
    tmp_0 = tmp_10;
    tmp_1 = (tmp_1 + tmp_10);
    var_7 = tmp_1;
    goto jmp2;
jmp1:
    tmp_0 = var_5;
    var_7 = tmp_0;
jmp2:
    tmp_0 = var_7;
```

```

    tmp_10 = tmp_0;
    return (tmp_10);
}

```

9.1.2 Encoded SPARC Assembly

```

.section ".text"
.align 4
.global fib
.type fib, #function
.proc 04
fib:
    .register %g2, #scratch
    .register %g3, #scratch
    save %sp, -128, %sp
    mov %i0, %g1
    st %g1, [%fp + -32]
    mov %i0, %g1
    st %g1, [%fp + -28]
    mov %i0, %g2
    subcc %g2, 1, %g0
    ble .b0
    set 0, %g1
    set 1, %g1
.b0:
    st %g1, [%fp + -24]
    ld [%fp + -24], %g2
    subcc %g2, 0, %g0
    bne .b1
    set 0, %g1
    set 1, %g1
.b1:
    st %g1, [%fp + -20]
    ld [%fp + -20], %g2
    subcc %g2, 1, %g0
    be .jmp1
    nop
    ld [%fp + -32], %g2
    add %g2, -1, %g1
    st %g1, [%fp + -16]
    ld [%fp + -16], %o0

```

```

    call fib, 0
    mov %o1, %g1
    st %g1, [%fp + -16]
    ld [%fp + -16], %g1
    st %g1, [%fp + -12]
    ld [%fp + -32], %g2
    add %g2, -2, %g1
    st %g1, [%fp + -16]
    ld [%fp + -16], %o0
    call fib, 0
    mov %o1, %g1
    st %g1, [%fp + -16]
    ld [%fp + -16], %g1
    st %g1, [%fp + -28]
    ld [%fp + -12], %g2
    ld [%fp + -16], %g3
    add %g2, %g3, %g1
    st %g1, [%fp + -12]
    ld [%fp + -12], %g1
    st %g1, [%fp + -8]
    b .jmp2
    nop
.jmp1:
    ld [%fp + -32], %g1
    st %g1, [%fp + -28]
    ld [%fp + -28], %g1
    st %g1, [%fp + -8]
.jmp2:
    ld [%fp + -8], %g1
    st %g1, [%fp + -28]
    ld [%fp + -28], %g1
    st %g1, [%fp + -16]
    ld [%fp + -16], %i0
    ret
    restore
    .size fib, .-fib

```

9.2 Greatest Common Denominator Function

9.2.1 Original C Source

```

int gcd(int u, int v)
{

```

```

int nu;
int nv;
while(v!=0)
{
    nu = v;
    nv = u - u/v * v;
    u = nu;
    v = nv;
}
return u;
}

```

9.2.2 Decompiled C Source

```

int gcd(int var_4, int var_6)
{
    int tmp_2;
    int tmp_3;
    int tmp_6;
    int tmp_8;
    int tmp_9;
    int var_3;
    int var_5;
    int var_7;
    int var_8;
    var_3 = var_4;
    var_5 = var_6;
    goto jmp1;
jmp0:
    tmp_2 = var_5;
    var_7 = tmp_2;
    tmp_3 = var_3;
    tmp_3 = (tmp_3 / var_5);
    tmp_3 = (tmp_3 * var_5);
    var_8 = (var_3 - tmp_3);
    var_3 = var_7;
    var_5 = var_8;
jmp1:
    tmp_2 = var_5;
    tmp_6 = (tmp_2 == 0);
    tmp_8 = (tmp_6 == 0);
    if (tmp_8)

```

```

        goto jmp0;
tmp_9 = var_3;
return (tmp_9);
}

```

9.2.3 Encoded SPARC Assembly

```

.section ".text"
.align 4
.global gcd
.type gcd, #function
.proc 04
gcd:
    .register %g2, #scratch
    .register %g3, #scratch
save %sp, -136, %sp
mov %i0, %g1
st %g1, [%fp + -40]
mov %i1, %g1
st %g1, [%fp + -36]
b .jmp1
nop
.jmp0:
    ld [%fp + -36], %g1
    st %g1, [%fp + -32]
    ld [%fp + -32], %g1
    st %g1, [%fp + -28]
    ld [%fp + -40], %g1
    st %g1, [%fp + -24]
    ld [%fp + -24], %g2
    ld [%fp + -36], %g3
    sdiv %g2, %g3, %g1
    st %g1, [%fp + -24]
    ld [%fp + -24], %g2
    ld [%fp + -36], %g3
    smul %g2, %g3, %g1
    st %g1, [%fp + -24]
    ld [%fp + -40], %g2
    ld [%fp + -24], %g3
    sub %g2, %g3, %g1
    st %g1, [%fp + -20]

```

```

    ld [%fp + -28], %g1
    st %g1, [%fp + -40]
    ld [%fp + -20], %g1
    st %g1, [%fp + -36]
.jump1:
    ld [%fp + -36], %g1
    st %g1, [%fp + -32]
    ld [%fp + -32], %g2
    subcc %g2, 0, %g0
    bne .b0
    set 0, %g1
    set 1, %g1
.b0:
    st %g1, [%fp + -16]
    ld [%fp + -16], %g2
    subcc %g2, 0, %g0
    bne .b1
    set 0, %g1
    set 1, %g1
.b1:
    st %g1, [%fp + -12]
    ld [%fp + -12], %g2
    subcc %g2, 1, %g0
    be .jmp0
    nop
    ld [%fp + -40], %g1
    st %g1, [%fp + -8]
    ld [%fp + -8], %i0
    ret
    restore
    .size gcd, .-gcd

```

9.3 Negate and Double Function

9.3.1 Original C Source

```

int negdouble(int val)
{
    if(val>>31)
    {
        val = ~val + 1;
        val = val << 1;
    }
}

```

```

else
{
    val = val << 1;
    val = ~val + 1;
}
return val;
}

```

9.3.2 Decompiled C Source

```

int negdouble(int var_4)
{
    int tmp_2;
    int tmp_5;
    int tmp_7;
    int tmp_8;
    int tmp_9;
    int var_3;
    var_3 = var_4;
    tmp_2 = (var_4 >> 31);
    tmp_5 = (tmp_2 == 0);
    tmp_7 = (tmp_5 == 1);
    if (tmp_7)
        goto jmp0;
    tmp_8 = (~var_3);
    var_3 = (tmp_8 + 1);
    tmp_2 = (var_3 << 1);
    var_3 = (var_3 << 1);
    goto jmp1;
jmp0:
    tmp_2 = var_3;
    tmp_2 = (tmp_2 << 1);
    tmp_8 = tmp_2;
    tmp_8 = (~tmp_8);
    var_3 = (tmp_8 + 1);
jmp1:
    tmp_2 = var_3;
    tmp_9 = tmp_2;
    return (tmp_9);
}

```

9.3.3 Encoded SPARC Assembly

```
.section ".text"
.align 4
.global negdouble
.type negdouble, #function
.proc 04
negdouble:
    .register %g2, #scratch
    .register %g3, #scratch
    save %sp, -120, %sp
    mov %i0, %g1
    st %g1, [%fp + -24]
    mov %i0, %g2
    sra %g2, 31, %g1
    st %g1, [%fp + -20]
    ld [%fp + -20], %g2
    subcc %g2, 0, %g0
    bne .b0
    set 0, %g1
    set 1, %g1
.b0:
    st %g1, [%fp + -16]
    ld [%fp + -16], %g2
    subcc %g2, 1, %g0
    bne .b1
    set 0, %g1
    set 1, %g1
.b1:
    st %g1, [%fp + -12]
    ld [%fp + -12], %g2
    subcc %g2, 1, %g0
    be .jmp0
    nop
    ld [%fp + -24], %g2
    not %g2, %g1
    st %g1, [%fp + -8]
    ld [%fp + -8], %g2
    add %g2, 1, %g1
    st %g1, [%fp + -24]
    ld [%fp + -24], %g2
    sll %g2, 1, %g1
```

```

    st %g1, [%fp + -20]
    ld [%fp + -24], %g2
    sll %g2, 1, %g1
    st %g1, [%fp + -24]
    b .jmp1
    nop
.jmp0:
    ld [%fp + -24], %g1
    st %g1, [%fp + -20]
    ld [%fp + -20], %g2
    sll %g2, 1, %g1
    st %g1, [%fp + -20]
    ld [%fp + -20], %g1
    st %g1, [%fp + -8]
    ld [%fp + -8], %g2
    not %g2, %g1
    st %g1, [%fp + -8]
    ld [%fp + -8], %g2
    add %g2, 1, %g1
    st %g1, [%fp + -24]
.jmp1:
    ld [%fp + -24], %g1
    st %g1, [%fp + -20]
    ld [%fp + -20], %g1
    st %g1, [%fp + -4]
    ld [%fp + -4], %i0
    ret
    restore
    .size negdouble, .-negdouble

```

9.4 Select Sort Function

9.4.1 Original C Source

```

void sort(int array[], int low, int high)
{
    int i, j;
    int min, min_index;

    for(i = 0; i < (high - 1); ++i)
    {
        for(min_index = i, min = array[i], j = i + 1; j < high; ++j)
        {

```

```

        if(array[j] < min)
        {
            min = array[j];
            min_index = j;
        }
    }
    array[min_index] = array[i];
    array[i] = min;
}
}

```

9.4.2 Decompiled C Source

```

void sort(int var_4, int var_6, int var_8)
{
    int tmp_2;
    int tmp_3;
    int tmp_4;
    int tmp_5;
    int tmp_8;
    int tmp_9;
    int var_3;
    int var_7;
    int var_9;
    int var_10;
    int var_11;
    int var_12;
    var_3 = var_4;
    var_7 = var_8;
    tmp_2 = 0;
    var_9 = 0;
    goto jmp4;
jmp0:
    tmp_2 = var_9;
    var_10 = tmp_2;
    tmp_3 = (var_9 << 2);
    tmp_3 = (tmp_3 + var_3);
    var_11 = (*(long *) (tmp_3 + 0));
    tmp_3 = var_9;
    tmp_2 = (var_9 + 1);
    var_12 = (var_9 + 1);
    goto jmp3;

```

```

jmp1:
    tmp_2 = var_12;
    tmp_2 = (tmp_2 << 2);
    tmp_3 = tmp_2;
    tmp_3 = (tmp_3 + var_3);
    tmp_3 = (*(long *) (tmp_3 + 0));
    tmp_2 = var_11;
    tmp_4 = (tmp_3 < var_11);
    tmp_5 = (tmp_3 > tmp_2);
    tmp_8 = (tmp_4 == 0);
    if (tmp_8)
        goto jmp2;
    tmp_3 = (var_12 << 2);
    tmp_3 = (tmp_3 + var_3);
    var_11 = (*(long *) (tmp_3 + 0));
    var_10 = var_12;
jmp2:
    tmp_3 = var_12;
    tmp_2 = (tmp_3 + 1);
    var_12 = tmp_2;
jmp3:
    tmp_2 = var_12;
    tmp_3 = var_7;
    tmp_4 = (tmp_2 < tmp_3);
    tmp_5 = (tmp_2 > tmp_3);
    tmp_8 = (tmp_4 == 1);
    if (tmp_8)
        goto jmp1;
    tmp_2 = var_10;
    tmp_2 = (tmp_2 << 2);
    tmp_9 = (tmp_2 + var_3);
    tmp_3 = (var_9 << 2);
    tmp_3 = (tmp_3 + var_3);
    (*(long *) (tmp_9 + 0)) = (*(long *) (tmp_3 + 0));
    tmp_3 = (var_9 << 2);
    tmp_3 = (tmp_3 + var_3);
    (*(long *) (tmp_3 + 0)) = var_11;
    var_9 = (var_9 + 1);
jmp4:
    tmp_3 = var_7;
    tmp_3 = (tmp_3 + -1);

```

```

    tmp_2 = var_9;
    tmp_4 = (tmp_3 < tmp_2);
    tmp_5 = (tmp_3 > tmp_2);
    tmp_8 = (tmp_5 == 1);
    if (tmp_8)
        goto jmp0;
    return;
}

```

9.4.3 Encoded SPARC Assembly

```

.section ".text"
.align 4
.global sort
.type sort, #function
.proc 04
sort:
    .register %g2, #scratch
    .register %g3, #scratch
    save %sp, -144, %sp
    mov %i0, %g1
    st %g1, [%fp + -48]
    mov %i2, %g1
    st %g1, [%fp + -44]
    set 0, %g1
    st %g1, [%fp + -40]
    set 0, %g1
    st %g1, [%fp + -36]
    b .jmp4
    nop
.jmp0:
    ld [%fp + -36], %g1
    st %g1, [%fp + -40]
    ld [%fp + -40], %g1
    st %g1, [%fp + -32]
    ld [%fp + -36], %g2
    sll %g2, 2, %g1
    st %g1, [%fp + -28]
    ld [%fp + -28], %g2
    ld [%fp + -48], %g3
    add %g2, %g3, %g1
    st %g1, [%fp + -28]

```

```

    ld [%fp + -28], %g2
    ld [%g2 + 0], %g1
    st %g1, [%fp + -24]
    ld [%fp + -36], %g1
    st %g1, [%fp + -28]
    ld [%fp + -36], %g2
    add %g2, 1, %g1
    st %g1, [%fp + -40]
    ld [%fp + -36], %g2
    add %g2, 1, %g1
    st %g1, [%fp + -20]
    b .jmp3
    nop
.jmp1:
    ld [%fp + -20], %g1
    st %g1, [%fp + -40]
    ld [%fp + -40], %g2
    sll %g2, 2, %g1
    st %g1, [%fp + -40]
    ld [%fp + -40], %g1
    st %g1, [%fp + -28]
    ld [%fp + -28], %g2
    ld [%fp + -48], %g3
    add %g2, %g3, %g1
    st %g1, [%fp + -28]
    ld [%fp + -28], %g2
    ld [%g2 + 0], %g1
    st %g1, [%fp + -28]
    ld [%fp + -24], %g1
    st %g1, [%fp + -40]
    ld [%fp + -28], %g2
    ld [%fp + -24], %g3
    subcc %g2, %g3, %g0
    bge .b0
    set 0, %g1
    set 1, %g1
.b0:
    st %g1, [%fp + -16]
    ld [%fp + -28], %g2
    ld [%fp + -40], %g3
    subcc %g2, %g3, %g0

```

```

        ble .b1
        set 0, %g1
        set 1, %g1
.b1:
        st %g1, [%fp + -12]
        ld [%fp + -16], %g2
        subcc %g2, 0, %g0
        bne .b2
        set 0, %g1
        set 1, %g1
.b2:
        st %g1, [%fp + -8]
        ld [%fp + -8], %g2
        subcc %g2, 1, %g0
        be .jmp2
        nop
        ld [%fp + -20], %g2
        sll %g2, 2, %g1
        st %g1, [%fp + -28]
        ld [%fp + -28], %g2
        ld [%fp + -48], %g3
        add %g2, %g3, %g1
        st %g1, [%fp + -28]
        ld [%fp + -28], %g2
        ld [%g2 + 0], %g1
        st %g1, [%fp + -24]
        ld [%fp + -20], %g1
        st %g1, [%fp + -32]
.jmp2:
        ld [%fp + -20], %g1
        st %g1, [%fp + -28]
        ld [%fp + -28], %g2
        add %g2, 1, %g1
        st %g1, [%fp + -40]
        ld [%fp + -40], %g1
        st %g1, [%fp + -20]
.jmp3:
        ld [%fp + -20], %g1
        st %g1, [%fp + -40]
        ld [%fp + -44], %g1
        st %g1, [%fp + -28]

```

```

    ld [%fp + -40], %g2
    ld [%fp + -28], %g3
    subcc %g2, %g3, %g0
    bge .b3
    set 0, %g1
    set 1, %g1
.b3:
    st %g1, [%fp + -16]
    ld [%fp + -40], %g2
    ld [%fp + -28], %g3
    subcc %g2, %g3, %g0
    ble .b4
    set 0, %g1
    set 1, %g1
.b4:
    st %g1, [%fp + -12]
    ld [%fp + -16], %g2
    subcc %g2, 1, %g0
    bne .b5
    set 0, %g1
    set 1, %g1
.b5:
    st %g1, [%fp + -8]
    ld [%fp + -8], %g2
    subcc %g2, 1, %g0
    be .jmp1
    nop
    ld [%fp + -32], %g1
    st %g1, [%fp + -40]
    ld [%fp + -40], %g2
    sll %g2, 2, %g1
    st %g1, [%fp + -40]
    ld [%fp + -40], %g2
    ld [%fp + -48], %g3
    add %g2, %g3, %g1
    st %g1, [%fp + -4]
    ld [%fp + -36], %g2
    sll %g2, 2, %g1
    st %g1, [%fp + -28]
    ld [%fp + -28], %g2
    ld [%fp + -48], %g3

```

```

add %g2, %g3, %g1
st %g1, [%fp + -28]
ld [%fp + -28], %g2
ld [%g2 + 0], %g1
ld [%fp + -4], %g2
st %g1, [%g2 + 0]
ld [%fp + -36], %g2
sll %g2, 2, %g1
st %g1, [%fp + -28]
ld [%fp + -28], %g2
ld [%fp + -48], %g3
add %g2, %g3, %g1
st %g1, [%fp + -28]
ld [%fp + -24], %g1
ld [%fp + -28], %g2
st %g1, [%g2 + 0]
ld [%fp + -36], %g2
add %g2, 1, %g1
st %g1, [%fp + -36]
.jump4:
ld [%fp + -44], %g1
st %g1, [%fp + -28]
ld [%fp + -28], %g2
add %g2, -1, %g1
st %g1, [%fp + -28]
ld [%fp + -36], %g1
st %g1, [%fp + -40]
ld [%fp + -28], %g2
ld [%fp + -40], %g3
subcc %g2, %g3, %g0
bge .b6
set 0, %g1
set 1, %g1
.b6:
st %g1, [%fp + -16]
ld [%fp + -28], %g2
ld [%fp + -40], %g3
subcc %g2, %g3, %g0
ble .b7
set 0, %g1
set 1, %g1

```

```

.b7:
    st %g1, [%fp + -12]
    ld [%fp + -12], %g2
    subcc %g2, 1, %g0
    bne .b8
    set 0, %g1
    set 1, %g1
.b8:
    st %g1, [%fp + -8]
    ld [%fp + -8], %g2
    subcc %g2, 1, %g0
    be .jmp0
    nop
    ret
    restore
    .size sort, .-sort

```

9.5 floattest

Float Test Function:

```

float floattest(float array[], float numElements)
{
    float total;
    float i;

    for(i = numElements - 1.0, total = 0.0; i>=1.0; i = i - 1.0)
    {
        total = total + array[(int)i];
    }
    return total/numElements;
}

```