DisIRer: Converting a Retargetable Compiler into a Multiplatform Binary Translator

YUAN-SHIN HWANG

National Taiwan University of Science and Technology

and

TZONG-YEN LIN and RONG-GUEY CHANG

National Chung Cheng University

This article proposes an alternative yet effective way of constructing a multiplatform binary translator, by converting a retargetable compiler into a binary translator. The rationale is that a retargetable compiler usually parses source programs into an Intermediate Representation (IR), and then translates IR into object code of different targets after performing analysis and optimizations. Specifically, the mechanism of code generation for multiple platforms from IR is already in place, and the missing link of building a multiplatform binary translator is a tool to transform binary programs back into IR. In order to fill in this missing link, this article presents a tool, called the *disIRer*. Just as a translator from machine language to assembly language is called a disassembler, a tool that translates executable binary programs to IR is called here a *disIRer*. The unique feature of this approach is that the retargetability of the binary translator is inherited directly from the retargetable compiler. A prototype multiplatform binary translator has been implemented upon GCC (the GNU Compiler Collection). *DisIRer* first converts binary programs back into GCC IR (Intermediate Representation), and afterward the GCC backend translates the IR to target binary programs of specified platforms. Experimental results show that x86 binary programs can be translated by this technique into ARM and Alpha binaries with reasonable code density and quality.

Categories and Subject Descriptors: D3.4 [Programming Languages]: Processors—Code generations; optimizations; retargetable compilers

General Terms: Algorithms, Performance, Design, Experimentation

Additional Key Words and Phrases: Binary translation, retargeting, intermediate representation

This research was supported in part by MOEA project 98-EC-17-A-01-S1-034 and NSC grants NSC95-2221-E-011-227-MY3, NSC98-2220-E-194-010, and NSC98-2221-E-011-065-MY3.

Authors' addresses: Y.-S. Hwang, Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei 106, Taiwan; email: shin@csie.ntust.edu.tw; T.-Y. Lin and R.-G. Chang, Department of Computer Science, National Chung Cheng University, Chia-Yi 621, Taiwan; email: {lty93, rgchang}@cs.ccu.edu.tw.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2010 ACM 1544-3566/2010/12-ART18 \$10.00
DOI 10.1145/1880043.1880045 <http://doi.acm.org/10.1145/1880043.1880045>

ACM Transactions on Architecture and Code Optimization, Vol. 7, No. 4, Article 18, Pub. date: December 2010.

ACM Reference Format:

Hwang, Y.-S., Lin, T.-Y., and Chang, R.-G. 2010. DisIRer: Converting a retargetable compiler into a multiplatform binary translator. *ACM Trans. Architect. Code Optim.* 7, 4, Article 18 (December 2010), 36 pages DOI = 10.1145/1880043.1880045 <http://doi.acm.org/10.1145/1880043.1880045>

---

1. INTRODUCTION

A binary translator is a software tool that converts binary executable programs from one ISA to another ISA [Altman et al. 2000; Sites et al. 1993]. It is commonly used to migrate existing application programs from some legacy systems to newly released machines. It can even be applied to facilitate designing new processors as they can provide benchmark programs that can be used to verify and tune the currently underdeveloped processors. Several binary translators have been implemented, designed for either single or multiple platforms [Andrews and Sand 1992; Cifuentes and Emmerik 2000; Cogswell and Segall 1995; Sites et al. 1993].

Compilers have long been considered as a totally different software tool, as their main application is to translate programs written in high-level programming languages to target object code [Aho et al. 2007]. The frontend of a compiler breaks up the source program into constituent pieces and creates an Intermediate Representation (IR) representation of the source program. Then the backend constructs the desired target program from the IR after the optional optimizer analyzes and transforms the IR, as shown in Figure 1(a). However, binary translators and compilers do indeed share some important characteristics.

Like a compiler, a binary translator can be divided into frontend, optimizer, and backend [Cifuentes and Emmerik 2000]. The frontend of a binary translator first decodes the source binary program into an IR representation, while the backend encodes desired target program from the IR after analysis and optimizations are performed by the optimizer, as illustrated in Figure 1(b). The diagrams in Figure 1 show that it is possible for a binary translator and a compiler to share one common backend if their frontends convert input programs into the same IR.

Based on this observation, this article proposes an alternative yet economical way of implementing a multiplatform binary translator, by converting a retargetable compiler into a binary translator. Specifically, the mechanism of code generation for multiple platforms from IR is already in place in the retargetable compiler, and hence the missing link of building a multiplatform binary translator upon a compiler is a tool that can transform binary programs back into IR. In order to fill in this missing link, this article develops such a tool, called the *disIRer*. Just as a translator from machine language to assembly language is called a disassembler, a tool that translates executable binary programs to IR is called here a *disIRer*.

There are advantages and disadvantages to this approach. The first advantage is that it is cost effective. Implementing a binary translator from scratch

Fig. 1. Compiler vs. binary translator.

is a very complicated process, while there are already well-written retargetable compilers around. Converting a compiler into a binary translator will definitely shorten the development process, as many useful modules of the compiler can be reused by the binary translator. In addition, the process of developing the frontend of the binary translator will also be greatly simplified, since the mapping from assembly to IR by the frontend of the binary translator can be viewed as an inverse process of the mapping from IR to instructions in the backend of the compiler. Second, the implemented binary translator will be retargetable as it inherits retargetability from the compiler. When the compiler is retargeted to a new platform, the new backend of the compiler can be used to facilitate retargeting this binary translator to the new platform. Finally, as the binary translator and compiler share the same IR, optimizations and transformations that are implemented in the compiler can be applied by the binary translator as well. The main disadvantage is that disIRer also inherits the limitation of a retargetable compiler, whose backends generate target programs for different platforms from a common IR. Certain architecture-specific or privileged instructions are not supported by the compiler as they can not be mapped from the common IR, and hence disIRer will not be able handle them either.

A prototype implementation of disIRer has been developed upon GCC (the GNU Compiler Collection), which is a well-established and versatile open-source compiler [GCC]. This prototype disIRer can convert x86 user-mode binary programs into GCC IR and then translate the IR into binary executables on bi-endian or little-endian platforms, such as ARM, Alpha, and even x86 itself. Experimental results on DSPstone [Zivojnović et al. 1994] and MediaBench [Lee et al. 1997] show that x86 programs can be translated by this technique into ARM and Alpha binaries with reasonable code density and quality.

Although currently disIRer is implemented upon GCC, it can read object codes that are generated by compilers other than GCC. In Section 4.7, experimental results of translating binary executables emitted by Intel C++ Compiler will be reported [Intel Corporation]. Furthermore, DisIRer is not limited to use the backend of the compiler where it is implemented. In fact, it can utilize the backends of any compilers as long as the compilers can import the internal tree representation exported by disIRer. This approach will be useful when the

target platforms do not support GCC directly, for example, Cygwin is needed on Microsoft Windows to provide a Linux-like environment [Red Hat Inc.]. This article has used the Microsoft Visual C++ compiler as the backend of disIRer in order to produce x86 binaries for Microsoft Windows platforms.

The rest of this article is organized as follows. Section 2 surveys the related work and briefly outlines the GCC architecture. Section 3 details the design and implementation of disIRer, and Section 4 presents the experimental results. Finally, Section 5 discusses some important issues of disIRer and then Section 6 concludes this work.

2. BACKGROUND

This section will survey the related work of binary translation, present a brief description of the structure of GCC, and list the limitations of the current disIRer implementation.

2.1 Related Work

Software-based binary translation systems can be classified into three classes: emulators, dynamic binary translators, and static binary translators [Altman et al. 2000, 2001, Cifuentes and Malhotra 1996]. All three approaches have limitations, but they can complement each other. Some work in this area revolves around innovative hybrid solutions to combine the best features of each approach.

An emulator interprets the source instructions on the target machine at run-time and can be highly compatible with the legacy architecture with relatively minor effort. This approach has been commonly deployed by commercial systems, such as IBM 360/370 emulation of some popular old mainframe models and VAX emulation of PDP-11 [Altman et al. 2000; Bagley 1976]. Similarly, the HP 3000 emulator developed by Bergh et al. enabled an unmodified object code program for previous HP 3000 designs to be loaded and run successfully on the new HP-PA computer family [Bergh et al. 1987]. Later Bedichek built an efficient simulator for the Motorola 88000 at the ISA level that ran on 68020-based Tektronix workstations [Bedichek 1990].

Migrating legacy programs to a new platform by an emulator requires the least amount of user interaction and the shortest amount of time. Unfortunately, an emulator also delivers the lowest performance level, typically running at only a very small fraction of the possible native performance level. In contrast, emulators with dynamic compilation and tracing capabilities promise to deliver significantly improved performance. For instance, the MIMIC simulator developed by May made use of caching techniques to drastically improve the code expansion factor when simulating IBM S/370 on the IBM RT RISC processor [May 1987].

Emulation with dynamic compilation frameworks is sometimes also referred to as program shepherding, sandboxing, or virtualization, as it can be used to monitor and control the execution of an existing program without needing to recompile or rebuild any code in that process [Smith and Nair 2005; Bhansali et al. 2006]. In addition, the source ISA and target ISA can be either the same

or different. Shade is an instruction set simulator running on SPARC systems that executes and traces the SPARC and MIPS I programs [Cmelik and Keppe 1994]. Dynamo is an emulator with a software dynamic optimization system that is capable of transparently improving the performance of a native instruction stream as it executes on the HP-PA processor [Bala et al. 1999, 2000]. DELI provides interfaces and dynamic binary translation optimization so that emulation users can develop their emulators and target-specific optimizations [Desoli et al. 2002]. Similarly, Microsoft Nirvana is an x86 runtime system with a lightweight, dynamic translation framework [Bhansali et al. 2006]. Although it is an old idea [Creasy 1981], OS-level virtualization is another application of emulation that enables a single computer to run multiple operating systems simultaneously [Vaughan-Nichols 2006], such as Parallels, Virtual PC, and VMware.

A dynamic binary translator translates instructions from the source ISA into the target ISA at runtime [Altman et al. 2000; Dehnert et al. 2003; Ebcioglu et al. 2001; Gschwind et al. 2000; Kim and Smith 2003; Li et al. 2006; Ung and Cifuentes 2000; Witchel and Rosenblum 1996; Zheng and Thompson 2000]. Java JIT (Just-In-Time) compilers are probably the best-known dynamic binary translators that compile the bytecodes into efficient instruction sequences for the underlying machine [Aycock 2003; Cramer et al. 1997; Suganuma et al. 2005; Yang et al. 1999]. Dynamic binary translation systems usually use varying degrees of profiling, code caching, and optimization in order to improve code quality. Examples include the Digital FX!32 profile-directed binary translator [Chernoff et al. 1998; Thompson 1996], the IBM Daisy and BOA binary translation systems [Ebcioglu and Altman 1997; Gschwind et al. 2000], and Transmeta's Code Morphing software for the Crusoe processor [Dehnert et al. 2003]. Some systems even incorporate emulators with dynamic translators. For instance, the Java HotSpot client compiler deploys a JVM interpreter and a just-in-time compiler to achieve a trade-off between the performance of the generated machine code and compilation speed [Kotzmann et al. 2008]. HP Aries emulator combines fast code interpretation with dynamic translation to execute PA-RISC applications transparently and accurately on IA-64 systems running HP-UX [Zheng and Thompson 2000]. In contrast to the dynamic binary translators listed before, couples of machine-adaptable dynamic binary translators have been developed to support different source and target machines through the specification of properties of these machines and their instruction sets, such as UQDBT [Ung and Cifuentes 2000] and the Walkabout/Yirr-Ma framework [Tröger 2005].

A static translator translates programs offline and can apply more rigorous code optimizations than a dynamic translator [Altman et al. 2000; Andrews and Sand 1992; Cifuentes and Emmerik 2000; Sites et al. 1993; Zhang et al. 2004]. Static binary translation can be categorized into direct binary translation [Andrews and Sand 1992; Sites et al. 1993] and indirect binary translation [Cifuentes and Emmerik 2000]. Direct binary translation is fairly similar to dynamic binary translation, but the translation is done before the execution of the code. The most widely known binary translators are those by Digital to translate code from VAX, MIPS, and SPARC to their new Alpha machine: VEST

and mx [Sites et al. 1993] and Freeport Express [Digital 1995], respectively. Similarly, Tandem has developed Accelerator to migrate legacy CISC object codes into their new RISC machines [Andrews and Sand 1992].

Indirect binary translation is more flexible than direct binary translation. Instead of direct mapping to the target binary codes, source binary codes are first translated into an intermediate representation and then are generated as target codes at low cost. *disIRer* belongs to this class [Lin et al. 2008]. The most closely related work to *disIRer* is UQBT [Cifuentes and Emmerik 2000]. UQBT translates the source binary into its IR, while *disIRer* transforms the source binary to GCC's IR. As a result, *disIRer* is more cost effective and flexible. In addition, *disIRer* can utilize the optimization functions already implemented in GCC to optimize translated codes. More recently, Microsoft has developed a software optimization and analysis framework called Phoenix, which can be adapted to read and write binaries and MSIL assemblies and represent the input files in an intermediate representation [Microsoft]. Currently Phoenix only raises binaries to LIR and it requires certain compiler switches to be used for the binary being built, while *disIRer* does not impose this restriction and can convert object codes into low-level IR and then back to high-level IR.

Hines et al. present a similar ASM2RTL tool for converting assembly code back to the RTL format used in the VISTA optimization framework, which in turn is designed to study the effects and possible benefits of reordering optimization phases via deoptimization and reoptimization on the same source and target platform [Hines 2004; Hines et al. 2005]. ASM2RTL performs almost exactly the same task of *Asm2RTL* of *disIRer*. The only exception is that ASM2RTL parses and translates each line of the assembly file into a semantically equivalent sequence of RTLs (lines 3–5 of Figure 3.3 in Hines [2004]), while *Asm2RTL* maps one or multiple assembly instructions into an appropriate RTL representation. In addition, ASM2RTL only converts assembly instructions to RTL, whereas *disIRer* transforms assembly instructions up to the AST format.

Decompilation is a program transformation that takes an executable file as input, and attempts to create an equivalent, high-level source code [Housel and Halstead 1974; Emmerik 2007]. This process can be viewed as the inverse of compilation. Although static binary translation can be achieved by a combination of decompilation and compilation, it might be beneficial for *disIRer* to only raise binaries into IR, as conditional flags of branch instructions can be represented by IR and then carried to the target backend without creating variables to store them.

Executable editing is a technique that has been commonly used for program instrumentation. It changes executable code by removing existing instructions and adding foreign code that observes or modifies a program's execution [Larus and Schnarr 1995]. OM is an executable editing library that internally represents instructions as RTL and uses relocation information from object files to analyze a program's control structure and relocate the edited code [Srivastava and Wall 1992]. ATOM (Analysis Tools with OM) is a single framework for building a wide range of customized program analysis tools [Srivastava and Eustace 1994]. It uses OM link-time technology to organize the final executable.

Table I. Source Languages and Target Platforms Supported by GCC

Languages	Target Processors
Ada, C, C++, Fortran, Java, Objective- C	Alpha, ARM, H8/300, System/370, System 390, x86, x86-64, IA-64, Motorola 68000, Motorola 88000, MIPS, PA-RISC, PDP-11, PowerPC, SuperH, SPARC, VAX, A29K, ARC, Atmel AVR, C4x, CRIS, D30v, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, MS32K, ROMP, Stormy16, V850, Xtensa

Fig. 2. GCC architecture.

EEL (Executable Editing Library) is a library that directly analyzes and modifies a program's instructions and consequently can operate on programs without relocation information [Larus and Schnarr 1995]. Pin follows the model of ATOM, with the additional ability of accessing architecture-specific details when necessary [Luk et al. 2005].

2.2 GCC Architecture

This section will briefly outline the internal structure of GCC, whose IR will be the target of disIRer. In addition, its backend can be used to perform optimization and code generation once disIRer converts binary code back into GCC IR.

GCC frontend can accept many programming languages such as C, C++, Fortran, and Java, while its backend can generate binary code for various target platforms, as listed in Table I. Figure 2 depicts the process that the GCC frontend transforms source programs in multiple languages into its common intermediate representations, and then its backend generates object code of various targets. Consider a C statement $c = a + b$. The GCC frontend first translates the statement into a high-level, tree-like intermediate representation, called the *Abstract Syntax Tree (AST)*. It then consults the machine description information and transfers the AST into the low-level *Register Transfer Language*

Fig. 3. RTL generation for $c = a + b$.

(RTL) representation. Finally the GCC backend converts the RTL to the target code by looking up the machine description.

2.2.1 Intermediate Representation. GCC has two types of IRs: high-level Abstract Syntax Trees (ASTs) and low-level Register Transfer Language (RTL) representation.

Abstract Syntax Trees (ASTs). An AST representation contains a complete representation for the source program provided as an input to the frontend [Stallman and the GCC Developer Community]. It is then processed by a code generator to produce machine code. In addition, it also can be used to create source browsers, intelligent editors, automatic documentation generators, interpreters, and provide any other programs with the ability to process source codes.

Register Transfer Language (RTL). RTL is the IR on which GCC performs most of its work [Stallman and the GCC Developer Community]. In this language, the instructions to be output are described in an algebraic form that describes what the instructions do. An RTL representation is a Lisp-like double-linked list of RTL expressions. RTL provides unlimited virtual registers to generate its instances called *insns*.

Figure 3 shows the flow that GCC creates an RTL code for the statement $c = a + b$. GCC first identifies from the AST that the standard name of this RTL is `addsi3`, which is defined in the *define_expand* of the machine description file. It then generates an instance of `addsi3` with three operands, that is, `addsi3 %0, %1, %2`. This instance is used as the index to locate the RTL generation function *gen_addsi3* for the standard name `addsi3` in the array *optab_table*, where GCC stores the RTL generation function for every standard name. Then the corresponding function *gen_addsi3* for standard name `addsi3` is called to

Fig. 4. Assembly generation for $c = a + b$.

check the preparation statements of the operands of `addsi3` and to output the RTL code `(set (reg:SI 0) (plus:SI (reg:SI 2) (reg:SI 1)))`.

2.2.2 Backend. Since GCC is a retargetable compiler, it has a backend for every target, which contains two important components: a Machine Description (MD) file and an assembly generation function.

Machine Description. An MD file consists of a file of instruction patterns and a C header file of macro definitions. The instruction pattern part is the description of native instructions, while the C header file defines macros to describe the information of the target machine that is not included in MD. MD provides the essential information that is needed for RTL generation and assembly generation, as shown in Figure 2.

Assembly Generation. The process of assembly generation is shown in Figure 4. The GCC backend first calls the function *Recog* to look up the machine description information in order to find the index of the RTL instruction in *insn_data*, which is an array that includes all functions that output assembly instructions. Once the index is found, the corresponding output function of the RTL instruction will be invoked to generate one or more assembly instructions.

2.3 Limitations

2.3.1 Self-Modified Code. Handling self-modifying code is not possible with a purely static translator, such as disIRer, since the target architecture must provide some means to detect modification of legacy code so that translations corresponding to the modified code can be invalidated [Altman et al. 2000; Kamunyor 2007]. Self-modifying code is usually handled by including an interpreter and marking modified code pages as “interpret only.” In other words, the target platform using a static binary translator must have a

runtime fallback mechanism such as a binary interpreter that is used to handle the situations that a static translator cannot [Ung and Cifuentes 2000].

2.3.2 Conditional Execution. The current implementation of disIRer can only read x86 binary instructions, and hence it does not handle conditionally executed instructions. However, as GCC supports some architectures that support conditional execution, disIRer can use the backends targeted for these architectures to generate conditionally executed code.

2.3.3 Dynamically Linked Libraries. Since disIRer is a static binary translator, extra care is needed in order to handle dynamically linked libraries. As the dynamically linked libraries that will be invoked by the source binary programs might or might not have been ported to the target platform, disIRer lets users specify whether the libraries should be searched and translated or not. If the libraries and OS have been ported to the target platform, users could direct disIRer to translate source binaries only. Otherwise, users will have to provide the library archives as well and instruct disIRer to search and transform the routines that will be invoked.

2.3.4 ABI and Calling Conventions. An ABI (Application Binary Interface) is the set of runtime conventions followed by all of the tools that deal with binary representations of a program, including compilers, assemblers, linkers, and language runtime support, and calling conventions are a subset of an ABI that specify how arguments are passed and function results are returned [GCC]. Since GCC backends generally conform to standard ABIs, disIRer should basically be able to exploit this feature and generate target binary programs that conform to ABIs.

Currently disIRer has not taken advantage of this feature, as its present implementation reads only x86 binaries and inserts “jacket” routines that move arguments to a stack. In the future, disIRer should abandon this approach and adopt some specification language (e.g., the PAL of UQBT [Cifuentes and Emmerik 2000]) in order to specify the calling conventions, parameter-passing conventions, stack frames, and local variable locations of source binary programs. Afterward, disIRer can translate the procedure calls in the source binary programs back to IR, and then the GCC backends of the target platforms can generate appropriate procedure call instructions.

2.3.5 System Calls. When application programs need to be migrated to a different platform, a jacket layer should be deployed to transform OS calls from the semantics of the legacy OS to the new one [Altman et al. 2000]. The current implementation of disIRer does not handle the mapping of system calls from different operating systems, and hence only platforms with the POSIX OS environment are supported now.

2.3.6 Architecture-Specific and Privileged Instructions. As disIRer is currently implemented upon GCC, its ability to handle architecture-specific and privileged instructions is determined by capacity of generating these instructions by the GCC backends. This capacity is decided by the MD (Machine

Fig. 5. The disIRer architecture.

Description) file, as it lists the mapping from RTL patterns to any architecture-specific or privileged instructions that can be generated by the GCC backend. Consequently, disIRer should be able to handle these special instructions by implementing a reverse mapping from the instructions back to RTLs. Unfortunately, most architecture-specific and privileged instructions (such as CPUID, RDTSC instructions on x86) are not supported by GCC, and hence currently disIRer cannot handle source binary code with architecture-specific and privileged instructions. Similarly, disIRer cannot handle arbitrary binaries, as certain hand-written code or obfuscated code might include some instructions that are not supported by the GCC backend.

2.3.7 Data Endianness and Size. The current implementation of disIRer copies the data section of the original binary code to a contiguous chunk of memory locations right after the translated code section, and hence the translated data section has the same size of the original data section. As the data items in the data section are not interpreted, disIRer does not handle translating data to a machine with different data endianness and size.

3. DISIRER

A disIRer is a tool that is used in this article to convert the retargetable compiler GCC into a multiplatform binary translator. It translates a binary executable program to the RTL representation and then to the AST format, which will be fed into GCC for further optimizations and target code generation, as shown in Figure 5. As a result, binary executable programs on the source platform can be migrated to any target platform that is supported by GCC.

A disIRer consists of three components: the disassembler, Asm2RTL, and RTL2AST. The disassembler converts the source binary code into the assembly code, then the Asm2RTL translates the assembly code back into the RTL format, and finally the RTL2AST transforms the RTL representation back into its corresponding ASTs. As indicated in Figure 5, any optimization modules in GCC can be applied to perform analysis and optimizations on the ASTs when beneficial before the GCC backend translates the ASTs into binary code.

Fig. 6. Disassembler.

of any supported platforms. Specifically, any analysis and optimizations implemented in GCC can be freely exploited once the disIRer turns binary code into ASTs.

3.1 Disassembler

Binary executable programs are commonly stored in the ELF format [TIS 1995], and their contents can be conveniently disassembled by `objdump` function in the GNU binutils library [GNU]. The disassembled code will then be forwarded into the `Clean()` of the disassembler to remove unneeded information, as shown in Figure 6. Finally, only the code and data accessed by the flow starting from `main()` will be fed into the `Map()` of the disassembler.

One important issue is how to deal with the global variables and data in the data section of a source binary code. DisIRer has implemented `Map()` to overcome this problem. It maps the global variables and data of the original data section to a contiguous chunk of memory locations right after the translated code section. The translated data section has the same size of the original data section, and hence the relative offset of each item in the data section is the same as the displacement of its source item from the beginning address of the source data section. However, the absolute address of every mapped data item will be different, as the data section is placed right after the code section, whose size is usually different from the size of its source code section, as shown in Figure 7. Therefore, all instructions that access data items in the data section using the absolute addressing mode must be modified in order to reach the data at the new addresses in the translated data section. Consider the following example. For the instruction “`load 0x8000`” in the original binary program, the disIRer can translate it as two instructions on the righthand side of arrow to access the memory, where *offset* is the distance from the start address of the data section to load 0x8000 and *array_starting_address* is the absolute address of the translated code.

$$\text{load } 0x8000 \Rightarrow \begin{array}{l} \text{add } r1, \text{offset}, \text{array\_starting\_address} \\ \text{load } r1 \end{array}$$

Fig. 7. Memory remapping.

Fig. 8. Possible RTL → assembly conversions in GCC.

3.2 Asm2RTL: Translating Assembly to RTL

Asm2RTL basically performs the opposite operation of the GCC code generator, translating assembly instructions back to the RTL representation. Consequently, it is essential to understand the mechanism of generating assembly code from RTL by the GCC code generator. Figure 8 displays the three possible cases: (a) one RTL is translated into one assembly statement, (b) multiple RTLs are mapped into one assembly statement, and (c) one RTL is transformed into several assembly statements. In order to reverse the code generation process, Asm2RTL must determine which of these three cases generates the current assembly statement. In other words, for a given assembly instruction Asm2RTL must determine which *insn* pattern produces it, which can be done by comparing the operand constraints, operand predicates, and operand machine modes defined in an *insn* pattern with the operands of the current assembly instruction. The constraint and predicate of an operand are used by the GCC code generator to decide that the operand is a constant, a register, or a memory address, while the operand machine mode is

```

Input:  $A$  = an assembly file
        $\Gamma$  = an MD file containing insn patterns
Output: RTL insn list

// Symbols
Let  $I_i$  be the  $i$ th insn pattern in  $\Gamma$ ,
 $S_{I_i}$  be the set of cases to generate assembly instructions in  $I_i$ 
 $S_{I_i}[j]$  be the  $j$ th case of  $S_{I_i}$ 
 $|S_{I_i}[j]|$  be the number of instructions in  $S_{I_i}[j]$ 
 $O_n$  be the set of insn patterns for the instruction  $A[n]$ 

// Building inverse mapping table  $M$ 
for  $i = 1$  to  $|\Gamma|$  // i.e. every insn pattern in  $\Gamma$ 
  for  $j = 1$  to  $|S_{I_i}|$  // i.e. every case in  $S_{I_i}$ 
    for  $k = 1$  to  $|S_{I_i}[j]|$ 
       $M[i][j][k] = S_{I_i}[j][k]$  // i.e.  $k$ th instruction in  $S_{I_i}[j]$ 

// Performing translation
 $n_{asm} = 1$  // # of assembly instructions will be mapped to one RTL
for  $n = 1$  to  $|A|$  // i.e. every assembly instruction in  $A$ 
  if ( $--n_{asm} > 0$ ) continue;
   $O_n = \phi$  // i.e. empty set
  for  $i = 1$  to  $|\Gamma|$ 
    for  $j = 1$  to  $|S_{I_i}|$ 
      for  $k = 1$  to  $|S_{I_i}[j]|$ 
        if ( $!opcode\_match(A[n+k-1], M[i][j][k])$ ) ||
            $!constraint\_match(A[n+k-1], M[i][j][k])$ 
          break
        if ( $k == |S_{I_i}[j]|$ )  $O_n = O_n \cup I_j$ 
       $n_{asm} = select\_RTL(O_n)$  // convert to the appropriate RTL

```

Fig. 9. Algorithm that translates assembly to RTL.

used to decide that the operand type is integer, float point, signed, or unsigned, etc. In the machine description, *insn* patterns are usually listed in the order according to the frequencies that they would be used. Therefore, Asm2RTL selects the *insn* pattern in the order listed in the output field while there are multiple *insn* patterns with matched operand constraint, operand predicate, and operand machine mode.

Figure 9 lists the algorithm of Asm2RTL, which performs the inverse operation of GCC code generators, translating an assembly code back to RTL. During the code generation phase, the GCC backend maps the RTL of an *insn* pattern I_i to a set of assembly code sequences S_{I_i} , each of which has one or multiple assembly instructions. For instance, the following *insn* pattern I_i

```

(define_insn "*mulsi3_1"
  [(set (match_operand:SI 0 "register_operand" "=r,r,r")
        (mult:SI (match_operand:SI 1
                  "nonimmediate_operand" "%rm,rm,0")
                  (match_operand:SI 2 "general_operand" "K,i,mr"))))
  (clobber (reg:CC FLAGS_REG))]

```

```

"GET_CODE (operands[1]) != MEM ||
GET_CODE (operands[2]) != MEM"
"@
imul{1}\t{%2,%1,%0|%0,%1,%2}
imul{1}\t{%2,%1,%0|%0,%1,%2}
imul{1}\t{%2,%0|%0,%2}"
[] )

```

maps the RTL to three possible assembly instruction sequences, that is, $S_i = \{\text{imul}\{1\}\backslash\t\{\%2,\%1,\%0\mid\%0,\%1,\%2\}, \text{imul}\{1\}\backslash\t\{\%2,\%1,\%0\mid\%0,\%1,\%2\}, \text{imul}\{1\}\backslash\t\{\%2,\%0\mid\%0,\%2\}\}$, where $\{1\}$ means that 1 is optional, $\backslash\t$ denotes a space or tab, and $\{\%2,\%1,\%0\mid\%0,\%1,\%2\}$ represents $\%2,\%1,\%0$ or $\%0,\%1,\%2$. Each sequence in S_i has only one instruction (i.e., $|S_i[j]| = 1, j = 1, 3$). Therefore, Asm2RTL must be able to identify if a sequence of assembly code matches any code sequence in S_i , for example, `imul %ecx, %ebx` matches the third case in S_i . Specifically, Asm2RTL first builds an inverse mapping table M from assembly code sequences to RTLs.

Once the mapping table M has been generated, Asm2RTL will then examine every assembly in the input object file in order to identify matching instruction sequences. For every input assembly instruction $A[n]$, this algorithm will determine if $A[n]$ is the starting instruction of any code sequences in $S_i[j]$, ($j = 1, |S_i|$). Specifically, it checks if the opcodes and operands of the instructions $A[n], A[n+1], \dots, A[n+|S_i[j]|-1]$ match those of the code sequence $S_i[j]$ by invoking `opcode_match(A[n+k-1], M[i][j][k])` and `constraint_match(A[n+k-1], M[i][j][k])`, ($k = 1, |S_i[j]|$). If all $|S_i[j]|$ pairs of instructions are matched, the *insn* pattern I_i will be included in O_n , which is the set of *insn* patterns that could produce an assembly code sequence in A starting from $A[n]$. As there might be more than one *insn* pattern in O_n , `select_RTL( $O_n$ )` will select the one listed first in the MD file and return the size of the matched code sequence (i.e., n_{asm}).

Figure 10 presents an example of translating a code sequence with only an assembly instruction into an RTL. Asm2RTL first searches the output template and identifies a match as the opcode of the current instruction matches the “add” entry in the output template. Then Asm2RTL recognizes that the types of three operands are registers, which are compatible with the operand predicate that requests registers, immediate values, or memory addresses. In addition, the operand machine mode is *SI* (Single Integer) mode. Since the operands of the add instruction match the aforesaid conditions defined in operand information, the RTL listed in the *insn* pattern is outputted.

Occasionally a code sequence with multiple assembly instructions will be identified as a possible candidate for RTL conversion, as only few *insn* patterns generate multiple assembly instructions in the GCC backend. Consider the following *insn* pattern I_i .

```

(define_insn "udivmodsi4"
  [(set (match_operand:SI 0 "register_operand" "=a")
        (udiv:SI (match_operand:SI 1 "register_operand" "0")
                  (match_operand:SI 2 "nonimmediate_operand" "rm")))]

```

Fig. 10. An assembly $\rightarrow$ RTL translation example.

```
(set (match_operand:SI 3 "register_operand" "&d")
      (umod:SI (match_dup 1) (match_dup 2)))
(clobber (reg:CC FLAGS_REG))
""
"xor{1}\t%3, %3\;div{1}\t%2"
[(set_attr "type" "multi")
 (set_attr "length_immediate" "0")
 (set_attr "mode" "SI")]
```

This pattern generates a code sequence with two assembly instructions (i.e., $|S_i| = 1$ and $|S_i[1]| = 2$). When Asm2RTL encounters the following x86 assembly code segment

```
xorl %edx,%edx
divl 12(%ebp)
```

it will determine that these two instructions match the code sequence generated by the preceding *insn* pattern, and then translate them back into the RTL representation of the instruction `udivmodsi4`.

```
(parallel
  [(set (reg:SI 0 ax)
        (udiv:SI (reg:SI 0 ax)
                  (mem:SI (plus:SI (reg:SI 6 bp)
                                   (const_int 12 [0xc])) [0 b+0 S4 A32]))])
  (set (reg:SI 1 dx)
        (umod:SI (reg:SI 0 ax)
```


Fig. 11. Translating an RTL *insn* back to AST.

```

(mem:SI (plus:SI (reg:SI 6 bp)
  (const_int 12))))
(use (reg:SI 1 dx))
(clobber (reg:CC 17 flags))]]

```

3.3 RTL2AST: Translating RTL to AST

Since GCC chooses a set of standard names that are meaningful in the RTL generation pass [Stallman and the GCC Developer Community], it is essential to identify the standard name of every *insn* pattern in the RTL representation first. Therefore, RTL2AST operates in two phases: it first maps every *insn* in the RTL to its standard name and then builds the AST using these standard names, as shown in Figure 11.

In order to determine the standard name for every RTL *insn*, a new machine description file *translation.md* that only contains *insn* patterns will be created and then be fed to GCC to produce a mapping function. For each input RTL *insn*, RTL2AST calls the mapping function to find its corresponding standard name. One complication is that some standard names for RTL generation cannot be handled with single *insn* on some target machines of GCC. Instead, a sequence of RTL *insns* are used to represent them. Consequently, RTL2AST needs to take one or multiple RTL *insns* each time in order to produce a standard name, and it solves this problem by using *translation.md* to accommodate all combinations of RTL *insns*.

Once a standard name is recognized, it is straightforward for RTL2AST to build its corresponding tree nodes. Consider the example shown in Figure 12.

Fig. 12. An RTL *insn* → AST example.

The standard name of the RTL *insn* currently processed by RTL2AST has the following form.

```
addsi3 operand1, operand2, operand3
```

The register constraints of this standard name are registers, the operand predicates are registers, and machine modes of its operands are *SI* modes. Consequently, the tree representation of the expression $r0 = r1 + r2$ is built.

3.4 Handling Conditional Branch Instructions

Handling conditional branch instructions is a tricky issue for a multiplatform binary translator, as the source and target platforms might take different ways to represent conditions and handle branches. Some processors are equipped with flags registers (or status registers) to store flags (or conditions), while some processors just use general-purpose registers. Furthermore, even the sets of flags or conditions that are stored in the flags or status registers vary among processors. As a result, some extra temporary locations or instructions might be needed in order to transfer conditions (or flags) from the source to the target platforms. Fortunately, disIRer can avoid such overhead by exploiting the features of GCC IR, since GCC backends can relay condition (status) information from AST down to RTL for efficient code generation to the target platforms.

The key manner for disIRer is to carry the condition (status) information of conditional branch instructions back to GCC IR. Consider the following x86 comparison and branch instructions.

```
    cmp %eax, %ebx
    je  label
    :
label:
```

Asm2RTL will transform these two instructions into RTL format:

```
(set (reg:CC 17 flags)           // cmp %eax, %ebx
 (compare:CC (reg:SI 1 bx)
  (reg:SI 0 ax)))
```

Fig. 13. AST representation of conditional branch instruction.

```

(set (pc)                                     // je label
  (if_then_else (eq (reg:CC 17 flags)
    (const_int 0 [0x0]))
    (label_ref 0)
    (pc)))

```

The first RTL instruction specifies that the result of this 32-bit integer comparison will be stored in the flags register, while the second RTL indicates that the program counter (pc) will be set to label if the result is equal to the constant 0.

RTL2AST identifies the standard names of these two RTL instructions as

```

cmpsi operand0 (eax), operand1 (ebx)
beq operand0 (label_ref 0)

```

and then constructs an AST tree for this pair of comparison and branch instructions, as shown in Figure 13. The AST subtree in the left box represents the comparison instruction `cmpsi %eax, %ebx`, while the *REG_DEC* nodes are new AST nodes introduced by disIRer to denote values stored in registers. The root node (*COND_EXPR*) of this AST tree designates a conditional branch instruction, whose destinations for the true and false conditions are represented by the AST subtrees for true label and false label, respectively.

If the target platform is an ARM processor, the ARM backend can take this AST tree to generate ARM assembly instructions. It will first generate a jump RTL instruction and two label RTL instructions, as shown in Figure 14(a). Since a conditional branch instruction will not jump to the target location when the condition is false, it is not necessary to generate a label for false condition. However, a jump instruction is still needed at the end of the false branch, and hence a jump RTL node is generated in order to bypass the instructions of the true branch. As for the two label RTL objects, the first label RTL is created as the entry for the true condition, while the other label RTL is constructed to designate the first instruction right after the conditional branch instruction.

The ARM backend will then convert the children of AST root node into a series of RTL objects. It will transform the left branch into two RTL objects:

Fig. 14. Generating ARM conditional branch instruction from AST.

one for the compare RTL instruction and the other for the conditional branch instruction, as shown in Figure 14(b). It will then translate the AST subtrees for the true and false branches into RTL objects, as depicted in Figure 14(c). Finally, it will transform these RTL objects into a sequence of ARM instructions, as listed in Figure 14(d). This result shows that the two x86 instructions `cmp %eax, %ebx; je label` have been translated by disIRer to ARM assembly instructions `cmp r2, r3; beq .L0`.

4. EXPERIMENTAL RESULTS

This section presents the experimental results of the binary translator implemented based on the disIRer and GCC. Performance evaluation will be conducted on x86, Alpha, and ARM platforms using the DSPstone and MediaBench benchmarks.

4.1 Implementation

A prototype implementation of disIRer for x86 binary executables has been built upon GCC 4.0.2. Out of 918 files in GCC, only three files have been modified and five new files have been created for implementing disIRer. Table II lists the names and sizes of the files that have been modified or created for the three components of disIRer, that is, the disassembler, Asm2RTL, and RTL2AST.

No files are created or modified for the disassembler. The `objdump` function of the GNU binutils library is used as part of the disassembler of disIRer as it can read and disassemble source binary executables in the ELF format, while `Clean()` and `Map()` functions are included in the *grammar.y* file.

Several new files have been created for the implementation of Asm2RTL, whose main function is coded in the *asm2rtl.c* file. The files *token.l* and *grammar.y* contain the scanner and parser that analyze the disassembled instructions generated by the disassembler in order to identify possible instruction sequences which can be mapped back to RTL. In addition, *final.c* in GCC is slightly modified to place an invocation entry to Asm2RTL.

Table II. DisIRer Implementation

	File	Type	Lines of New Code	Comment
Disassembler	objdump			in GNU binutils library
Asm2RTL	<i>asm2rtl.c</i>	New	9136	Asm2RTL main
	<i>asm2rtl.h</i>	New	527	
	<i>token.l</i>	New	402	
	<i>grammar.y</i>	New	4871	including <code>Clean()</code> and <code>Map()</code>
	<i>final.c</i>	Modified	9	invoking ASM2RTL
RTL2AST	<i>tree_expand.c</i>	Modified	1027	RTL2AST main
	<i>passes.c</i>	Modified	341	
	<i>translation.md</i>	New	3939	mapping RTLs to standard names

Table III. Hardware/Software Configurations

Platform	Hardware/Software Configuration	
x86	Compiler	GCC 4.0.2
		Visual C++ Compiler 6.0
		Intel C++ Compiler 10.1
	OS	Linux 2.6.9
		Windows XP
	Processor	Intel Pentium 4 3.06 GHz
ARM	Compiler	GCC 4.0.2
	OS	Linux version 2.6.9
	Simulator	GNU GDB/ARMulator 6.3
Alpha	Compiler	GCC 4.0.2
	OS	Alpha DEC OSF
	Simulator	SimpleScalar 3.0

The main function of RTL2AST is placed in the file *tree_expand.c*, which handles AST creation and manipulation in GCC. The file *passes.c* has also been updated to handle the interaction between RTL2AST and GCC. Finally, a new file called *translation.md* has been created to accommodate the mapping from RTLs to their corresponding standard names.

4.2 Setup

Table III lists the platforms currently supported by the disIRer and their environmental settings. The source platform is an x86 Linux system, while the target platforms of binary translation are x86, ARM, and Alpha. DisIRer is implemented on GCC 4.0.2 so that it can translate the source x86 binaries into GCC IR, which will be in turn translated by GCC backends to target binaries on x86, ARM, and Alpha systems. In addition, Microsoft Visual C++ 6.0 is used as the backend to produce the translated x86 binaries for Windows XP systems. Specifically, two x86 platforms will be targeted, x86 Linux and Windows XP systems. Performance evaluation of translated x86 binaries is conducted on a 3.06 GHz Intel Pentium 4 system with 4GB memory, while ARM binary executable programs are executed on GDB/ARMulator, which is an instruction set simulator for ARM processors embedded inside the GNU debugger (GDB). As for Alpha, the translated codes are executed on SimpleScalar 3.0 [Austin et al. 2002].

Fig. 15. Verification methodology.

In order to verify the translation process, this article deploys a methodology to compare the results of the source and target binary programs, as shown in Figure 15. This process can facilitate debugging the implementation of disIRer, as any discrepancies indicate programming errors on the disIRer implementation.

Four source x86 binaries will be generated natively for every benchmark by GCC 4.0.2 on x86 Linux platform with -O0, -O1, -O2, and -O3 optimization options, and then each source binary is translated into the target binary code with the proper level of optimizations. Specifically, the disIRer will first convert the source binary code compiled with -O0 option to AST, and then invoke the GCC -O0, -O1, -O2, and -O3 optimization options on the AST before it is translated into target binary code by the GCC backend. Similarly, source binaries with the -O1 option will be translated to target binary executable programs with -O1, -O2, and -O3 level of optimizations, respectively. Finally, source binary programs with -O2 option will be optimized by the -O2 and -O3 options, while those source programs with -O3 option will be translated with -O3 optimizations invoked.

4.3 x86 to x86 Translation

In order to measure the translation cost of disIRer, this section presents the experimental results of translating DSPstone and MediaBench binaries from an x86 Linux back to the same x86 Linux system. Figure 16 shows the performance and code size impact of binary translation of DSPstone benchmarks from x86 to x86, where “original” denotes the performance of the original x86 code and “translated” represents that of the x86 code translated by disIRer. The numbers above the figure denote the levels of optimizations that are invoked for producing the source binaries and translating the target binaries.

Fig. 16. x86 to x86 translation of DSPstone.

For instance, the phrase “O1 -> O2” means that the source binary code is produced with the “gcc -O1” compiler optimization option and the target binary program is generated by the GCC backend after the “-O2” optimizations have been performed on the IR.

Figure 16(a) reveals that the performance impact incurred by the translation is small for the DSPstone benchmarks, ranging from 19% speedup to 51% slowdown. When the -O0 source binary programs are translated to the target binary code with no optimizations, the average slowdown is 29%. As more and more optimizations are applied by -O1, -O2, and -O3 options, the average performance degradation reduces to 14%, 15%, and even just 1%. This result does show that optimization routines of GCC can be applied by disIRer to perform optimizations for performance improvement before the final target code generation.

Optimizations can be performed on the IR translated from optimized source binaries as well, although the improvement is not as significant as that seen by the translations from the -O0 source programs. The main reason is that optimized object programs are already optimized by the native compiler and hence their execution times are smaller than those of unoptimized source binaries. In addition, optimized object programs are generally harder to analyze as their original program structures have been aggressively transformed by the

optimization routines. The object code translated from the -O1 source binary program runs slower than the source binary by 19% on average. The average slowdown can be reduced to only 6% when -O3 optimizations are performed, although the -O2 level does not help. Similar improvement can also be observed from the -O2 source binary program, that is, the 12% of average performance degradation seen by the -O2 translated object code is converted to a 1% speedup when -O3 optimizations are applied. Finally, for the source and target binaries with -O3 level of optimizations, the slowdown is only 4%, which indicates the code quality of the binary translation process is very good.

Figure 16(b) indicates that the binary translation process does increase the code size of the target binaries. The average ratios of code size expansion range from 2.6 to 4.1 for different levels of optimizations. This poor code density seems to contradict the good code quality seen in Figure 16(a). The reason is that a source x86 instruction is usually translated by *disIRer* into several simpler target x86 instructions, each of which takes fewer cycles to execute than the original instruction. As a result, the execution times of the original x86 code and the translated x86 code are very close, while their sizes are not.

The performance of the translated MediaBench binaries is not as ideal, since these benchmarks contain more instructions and more complicated flow of control than DSPstone benchmarks. The source of performance degradation would be the extra instructions introduced by the binary translation process to handle memory remapping. Figure 17(a) illustrates that the average slowdown ratios of translated x86 binaries are within the range of 1.6 to 3.1 for different levels of optimizations. However, optimizations can still improve the performance of the translated binary programs. For instance, the worst average slowdown ratio 3.1 comes from the unoptimized object code translated from the unoptimized source binary program, while the optimized target object code of the same source binary reduces the average slowdown ratio down to 1.6.

The translated x86 binary executables of MediaBench benchmarks suffer worse code expansion than that of DSPstone programs, as shown in Figure 17(b). Average code expansion ratios span from 3.2 to 5.4. In addition to the reason that a source x86 instruction might be translated into several simpler target x86 instructions, code expansion and performance degradation are caused by the extra instructions introduced by the binary translation process to handle memory remapping.

4.4 x86 to ARM Translation

Figure 18 and Figure 19 present the performance and code size impact of binary translation from x86 to ARM for DSPstone and MediaBench, respectively, where “native” means the ARM code generated natively by the GCC 4.0.2 compiler and “translated” represents the ARM code translated from x86 binaries by *disIRer*.

Figure 18(a) shows that the average slowdown ratios incurred by the x86-to-ARM translation vary from 1.1 to 3.9 for the DSPstone benchmarks. The worst performance penalty occurs when the unoptimized source binary programs are

Fig. 17. x86 to x86 translation of MediaBench.

translated directly to the object code without any optimizations, that is, the “O0 -> O0” case. However, the average slowdown is lowered to only a 14% of performance degradation when -O3 optimizations are conducted during the translation. As for optimized source binaries that are generated with -O1, -O2, or -O3 optimizations, the average performance slowdown ratios are about 2.1~2.2. The performance degradation is generally caused by the extra load and store instructions introduced by the CISC to RISC translation process and the memory remapping issue.

Since this is a CISC to RISC translation process, an x86 instruction may be converted into a sequence of ARM instructions. However, this binary translation process does not introduce significant amounts of instructions to the target object programs. Figure 18(b) reveals that the average ratios of code size expansion incurred by the translation are indeed very small for the DSPstone benchmarks, ranging from 7% to 53% for various levels of optimizations. Furthermore, the worst code expansion ratio of 53% for “O0 -> O0” case can be reduced to 7% when -O2 or -O3 level of optimizations is applied.

Similar performance degradation and code size expansion can be observed for the MediaBench programs. Figure 19(a) depicts the average slowdowns of MediaBench benchmarks can be as bad as 6.3, while Figure 19(b) shows that the average code expansion ratios range from 1.2 to 2.1. However, optimizations

Fig. 18. x86 to ARM translation of DSPstone.

still help reduce code expansion and performance degradation, for example, the average code expansion ratio can be reduced from 2.1 to 1.2 and the average runtime impact is cut from 6.3 to 2.2.

Both DSPstone and MediaBench benchmarks suffer performance degradation and code size expansion incurred by the extra load and store instructions introduced by the CISC to RISC translation process and the memory remapping issue. For example, the large amount of load and store instructions in the *adpcm* program of DSPstone results in about 4 times of the performance slowdown and 2 times of code size expansion. Another noticeable example is *matrix* of DSPstone. Although its code size is small, its main loop is executed many times with lots of load and store instructions. The outcome is that its code size only expands about 10% while its execution is about 5 times slower than the native code.

The situation is worse for the MediaBench programs because they generally execute more loops and access larger amounts of data, such as matrices. The ARM object programs that are translated from optimized x86 binaries run about 5 times slower than the natively generated ARM binary code.

Optimizations invoked by disIRer generally do not improve performance and code density much when the source binary programs are generated and optimized by the compiler. However, significant improvement can be observed

Fig. 19. x86 to ARM translation of MediaBench.

when optimizations are performed by disIRer on unoptimized source binary code. The main reason is that more opportunities for various optimizations can be found and exploited by disIRer in unoptimized source binaries than optimized source code.

4.5 x86 to Alpha Translation

Figure 20 and Figure 21 illustrate the performance and code size impact of binary translation from x86 to Alpha, where “native” represents the native Alpha codes compiled by GCC 4.0.2 and “translated” represents the Alpha binaries translated by disIRer from x86 codes. The sizes of translated binaries are expanded about 50% over the sizes of the native Alpha code for DSPstone and MediaBench benchmarks with only few exceptions, as shown in Figure 20(b) and Figure 21(b). Since this is a CISC to RISC translation process, an x86 instruction may be converted into a sequence of Alpha instructions and hence this code size impact indicates that the code quality of this x86-to-Alpha translation is reasonably good.

Figure 20(a) shows that the average slowdown ratios incurred by the x86-to-Alpha translation vary from 1.9 to 9.7 for the DSPstone benchmarks. The worst penalty occurs when the unoptimized source binary programs are

(a) performance impact

(b) code size impact

Fig. 20. x86 to Alpha translation of DSPStone.

(a) performance impact

(b) code size impact

Fig. 21. x86 to Alpha translation of Mediabench.

translated directly to the object code without any optimizations, but that average slowdown of 9.7 times the native execution times is lowered to a 2.9 times of performance degradation when -O3 optimizations are conducted before target code generation. The performance degradation is generally caused by the extra load and store instructions introduced by the CISC to RISC translation process and the memory remapping issue.

Similar performance impact can be observed for MediaBench programs as well, as shown in Figure 21(a). Slowdown ratios of unoptimized source binary programs are over 7 times when they are translated with no optimizations. However, the average slowdown ratios are generally reduced to about 3 times when various levels of optimizations are performed.

4.6 x86 to x86 Translation on Windows XP

DisIRer can be ported to Microsoft Windows without any effort by using the Linux-like environment Cygwin. However, the translated binary code cannot be executed directly on Windows and must be invoked by Cygwin, since the x86 backend of GCC is targeted to Linux. As a result, disIRer can not use the GCC x86 backend to generate the target code on Windows after processing the source binary programs. Instead, disIRer now exports its IR to Microsoft Visual C++ compiler, whose backend will then generate binary executables that can be executed directly on Windows. The drawback of this approach is that disIRer now has to generate several extra instructions to perform the memory remapping for each load or store instruction.

Figure 22 shows the performance and code size impact of binary translation of DSPstone benchmarks from x86 to x86, where “native” denotes the performance of the native x86 code generated by the Visual C++ 6.0 compiler and “translated” represents that of the x86 code translated by disIRer and Visual C++ backend. Figure 22(a) depicts that there is no performance degradation at all for the translated object programs for DSPstone benchmarks, although the code sizes expand 3.5 times on average, as illustrated in Figure 22(b). Good performance can still be achieved despite the expanded code sizes because there are not many memory remapping operations introduced by DSPstone benchmarks.

Runtime impact is obvious for MediaBench programs, as disIRer needs to generate many memory remapping instructions. The average slowdown ratios of different optimization options range from 2.1 to 3.6, as shown in Figure 23(a). The other consequence of extra remapping instructions introduced by load and store instruction would be code expansion. Figure 23(b) depicts the code sizes of translated MediaBench benchmarks can be expanded to as much as 4.4 times their original sizes.

4.7 Translating Binaries Generated by *icc*

In order to demonstrate that disIRer can handle binary executables that are generated from other compilers, *djpeg* is first compiled by the Intel C++ Compiler with -O0, -O1, -O2, and -O3 optimization flags, and then its x86 object files are translated by disIRer back to x86 instructions with -O0, -O1, -O2, and

Fig. 22. x86 to x86 translation of DSPstone (on Windows).

-O3 optimization options, respectively. Figure 24 presents the code size and performance impact of *djpeg* by comparing the translated binaries with locally generated object code. The O0->O0, O1->O1, O2->O2, and O3->O3 results of *djpeg* compiled by GCC are taken from Figure 17 as a comparison. The results show that disIRer performs worse when translating binary executables compiled by *icc*. The main reason is because the code quality of binary executables locally compiled by *icc* is better than that by GCC.

5. DISCUSSION

This section discusses some advantages and issues of disIRer.

5.1 Advantages of Using Existing Retargetable Compilers

The main advantage of using an existing retargetable compiler to implement a multiplatform binary translator is that the mechanism of code generation for multiple platforms from IR is already in place in the retargetable compiler. Specifically, it relieves the developer of the burden of reimplementing the backends of various target platforms into the binary translator. In fact, this advantage is twofold as it will also greatly simplify the process of developing the frontend of the binary translator which transforms assembly back to IR, since the mapping from assembly to IR can be viewed as an inverse process of

Fig. 23. x86 to x86 translation of MediaBench (on Windows).

Fig. 24. x86 to x86 translation of jpeg.

the mapping from IR to instructions in the backend. Consequently, if only in very few occasions an RTL pattern is mapped to multiple target instructions in GCC, the inverses of most mapping functions in the GCC backends can be used by Asm2RTL to transform assembly instructions up to RTL objects.

According to Section 3.2, there are three possible mappings from RTL to assembly instructions in GCC backends: one-to-one, many-to-one, and one-to-many (see Figure 8). As each of one-to-one and many-to-one mapping functions

Table IV. Mapping from RTL to Assembly Instructions

	gcc -O0			gcc -O1			gcc -O2			gcc -O3		
	single	multi	%multi	single	multi	%multi	single	multi	%multi	single	multi	%multi
rawaudio	332	0	0.00%	250	0	0.00%	292	0	0.00%	295	0	0.00%
rawdaudio	351	0	0.00%	267	0	0.00%	311	0	0.00%	314	0	0.00%
epic	6608	15	0.23%	5513	11	0.20%	6042	9	0.15%	6600	13	0.20%
unepic	5283	16	0.30%	3280	11	0.33%	3535	10	0.28%	3919	14	0.36%
g721.encode	2501	1	0.04%	1862	1	0.05%	2010	0	0.00%	2388	0	0.00%
g721.decode	2507	1	0.04%	1867	1	0.05%	2013	0	0.00%	2339	0	0.00%
toast	12308	12	0.10%	8131	9	0.11%	8085	2	0.02%	9219	5	0.05%
untoast	12308	12	0.10%	8131	9	0.11%	8085	2	0.02%	9219	5	0.05%
cjpeg	42686	10	0.02%	31667	7	0.02%	33464	7	0.02%	40801	5	0.01%
djpeg	41224	10	0.02%	30567	7	0.02%	32230	7	0.02%	39762	5	0.01%

of GCC backends maps one or multiple RTL objects into one single assembly instruction, every inverse of these mapping functions maps an assembly instruction to one or multiple RTL objects. As a result, these mapping functions can be used to facilitate the implementation of Asm2RTL. Only those one-to-many cases need special attention, as each inverse of these functions now matches multiple assembly instructions back to one single RTL object. Fortunately, almost all of mappings performed during the code generation phase of GCC backends are one-to-one and many-to-one, while only very few transformations are one-to-many.

Table IV presents the frequencies of different mappings when MediaBench programs are compiled by GCC on the x86 platform, where the “single” columns categorize the numbers of times that one-to-one and many-to-one transformations are performed and the “multi” columns indicate the occurrences of one-to-many mappings. It shows that a RTL pattern is translated into multiple assembly instructions only in very few occasions. Specifically, only 0.00%~0.36% of instruction mappings are one-to-many, as displayed in the “%multi” columns. This result has demonstrated that the RTL-to-assembly mapping functions of GCC backends can be conveniently served as a starting point of implementing assembly-to-RTL transformations of Asm2RTL.

5.2 Issues

There are some important issues that are closely related to the disIRer implementation. The most tricky one is how to select an RTL to match an assembly instruction in the translation process. In GCC, the MD file describes how many RTL combinations can generate an assembly instruction. Currently, disIRer always chooses the first one that appears in MD as the output of an assembly instruction. However, this might not be the best selection scheme. Further study must be conducted in order to determine if it is optimal or to devise a better approach.

The second issue arises from the fact that the high-level information of source programs is not present in the binary code. For instance, the structures defined by the struct construct in the C language cannot be easily acquired from the binary code. As a result, superfluous instructions might be generated

during the binary translation process. Currently, disIRer relies on the built-in optimizations of GCC to improve the code quality.

The third issue is how to reduce the overhead of the memory remapping operations that are incurred by binary translation. As the memory addresses of the data will be different on the target platform, disIRer must generate extra instructions to perform memory remapping for load and store instructions, as shown in Figure 7. The amount of extra instructions depends on the instruction set architecture of the target platform. From the results shown in Section 4, this overhead indeed degrades the performance of the translated code.

Finally, disIRer is retargetable as it inherits retargetability from GCC. When GCC is ported to a new platform, disIRer can translate source binary executables onto the new platform. Another key advantage of this approach is that the intrinsic optimizations and built-in functions of GCC can be applied to optimize the translated code.

6. CONCLUSIONS

This article has presented an economical way of implementing a multiplatform binary translator, by converting a multiple-target compiler into a binary translator. The main idea is to make a binary translator share some common components, such as optimizers and target code generators, with a retargetable compiler. This article has used GCC to develop a prototype implementation of disIRer, which converts source binary programs back into GCC IR. Consequently, disIRer can directly use the optimizer and backend of GCC to optimize and then translate IR to target binary code. The advantages of this approach come from its cost effectiveness and its retargetability. Experimental results have shown that x86 programs can be translated by this technique into ARM and Alpha binaries with reasonable code density and quality.

REFERENCES

- AHO, A. V., LAM, M. S., SETHI, R., AND ULLMAN, J. D. 2007. *Compilers: Principles, Techniques, and Tools* 2nd Ed. Addison Wesley.
- ALTMAN, E. R., EBCIOĞLU, K., GSCHWIN, M., AND SATHAYE, S. 2001. Advances and future challenges in binary translation and optimization. *Proc. IEEE* 89, 11, 1710–1722.
- ALTMAN, E. R., KAEI, D., AND SHEFFER, Y. 2000. Welcome to the opportunities of binary translation. *IEEE Comput.* 33, 3, 40–45.
- ANDREWS, K. AND SAND, D. 1992. Migrating a CISC computer family onto RISC via object code translation. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*. 213–222.
- AUSTIN, T., LARSON, E., AND ERNST, D. 2002. SimpleScalar: An infrastructure for computer system modeling. *IEEE Comput.* 35, 2, 59–67.
- AYCOCK, J. 2003. A brief history of just-in-time. *ACM Comput. Surv.* 35, 2, 97–113.
- BAGLEY, J. D. 1976. Special feature: Microprogrammable virtual machines. *IEEE Comput.* 9, 2, 38–42.
- BALA, V., DUESTERWALD, E., AND BANERJIA, S. 1999. Transparent dynamic optimization: The design and implementation of dynamo. Tech. rep. HPL-1999-78, HP Laboratories.

- BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1–12.
- BEDICHEK, R. 1990. Some efficient architecture simulation techniques. In *Proceedings of the Winter USENIX Conference*. 53–63.
- BERGH, A. B., KEILMAN, K., MAGENHEIMER, D. J., AND MILLER, J. A. 1987. HP 3000 emulation on HP precision architecture computers. *Hewlett-Packard J.* 87–89.
- BHANSALI, S., CHEN, W.-K., DE JONG, S., EDWARDS, A., MURRAY, R., DRINIĆ, M., MIHOČKA, D., AND CHAU, J. 2006. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the Virtual Execution Environments Conference*. 154–163.
- CHERNOFF, A., HERDEG, M., HOOKWAY, R., REEVE, C., RUBIN, N., TYE, T., YADAVALLI, S. B., AND YATES, J. 1998. Fx!32: A profile-directed binary translator. *IEEE Micro* 18, 2, 56–64.
- CIFUENTES, C. AND EMMERIK, M. V. 2000. UQBT: Adaptable binary translation at low cost. *IEEE Comput.* 33, 3, 60–66.
- CIFUENTES, C. AND MALHOTRA, V. 1996. Binary translation: Static, dynamic, retargetable? In *Proceedings of the International Conference on Software Maintenance*. 340–349.
- CMELIK, B. AND KEPPE, D. 1994. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. 128–137.
- COGSWELL, B. AND SEGALL, Z. 1995. Timing insensitive binary-to-binary migration across multi-processor architectures. In *Proceedings of the 3rd Workshop on Parallel and Distributed Real-Time Systems*. 193–194.
- CRAMER, T., FRIEDMAN, R., MILLER, T., SEBERGER, D., WILSON, R., AND WOLCZKO, M. 1997. Compiling java just in time. *IEEE Micro* 17, 3, 36–43.
- CREASY, R. J. 1981. The origin of the VM/370 time-sharing system. *IBM J. Res. Devel.* 25, 5, 483–490.
- DEHNERT, J. C., GRANT, B. K., BANNING, J. P., JOHNSON, R., KISTLER, T., KLAIBER, A., AND MATTSO, J. 2003. The Transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*. 15–24.
- DESOLI, G., MATEEV, N., DUESTERWALD, E., FARABOSCHI, P., AND FISHER, J. A. 2002. DELI: A new runtime control point. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*. 257–268.
- DIGITAL. 1995. Freeport express. <http://www.novalink.com/freeport-express/>.
- EBCIOGLU, K., MEMBER, S., ALTMAN, E., GSCHWIND, M., AND SATHAYE, S. 2001. Dynamic binary translation and optimization. *IEEE Trans. Comput.* 50, 6, 529–548.
- EBCIOGLU, K. AND ALTMAN, E. R. 1997. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*. 26–37.
- EMMERIK, M. J. V. 2007. Static single assignment for decompilation. Ph.D. thesis, The University of Queensland.
- GCC. The GNU compiler collection. <http://gcc.gnu.org/>.
- GNU. GNU binary utilities (GNU binutils). <http://www.gnu.org/software/binutils/>.
- GSCHWIND, M., ALTMAN, E. R., SATHAYE, S., LEDAK, P., AND APPENZELLER, D. 2000. Dynamic and transparent binary translation. *IEEE Comput.* 33, 3, 54–59.
- HINES, S., KULKARNI, P., WHALLEY, D., AND DAVIDSON, J. 2005. Using de-optimization to re-optimize code. In *Proceedings of EMSOFT'05*. 114–123.
- HINES, S. R. 2004. Using de-optimization to re-optimize code. M.S. thesis, The Florida State University.
- HOUSEL, B. C. AND HALSTEAD, M. H. 1974. A methodology for machine language decompilation. In *Proceedings of the ACM Annual Conference* Vol. 1, 254–260.
- INTEL CORPORATION. Intel C++ compiler for linux, i486 platform. <http://software.intel.com/en-us/articles/intel-c-compiler-professional-edition-for-linux-documentation/>
- KAMUNYORI, J. W. 2007. Handling self-modifying code using software dynamic translation. In *Proceedings of the Conference on Diversity in Computing*. 32–32.

- KIM, H.-S. AND SMITH, J. E. 2003. Dynamic binary translation for accumulator-oriented architectures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*. 25–35.
- KOTZMANN, T., WIMMER, C., MOŠSENBOČK, H., RODRIGUEZ, T., RUSSELL, K., AND COX, D. 2008. Design of the Java HotSpot client compiler for Java 6. *ACM Trans. Architect. Code Optim.* 5, 1.
- LARUS, J. R. AND SCHNARR, E. 1995. EEL: Machine-Independent executable editing. In *Proceedings of SIGPLAN Conference on Programming Languages, Design and Implementation*. 291–300.
- LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communication systems. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. 330–335.
- LI, J., ZHANG, Q., XU, S., AND HUANG, B. 2006. Optimizing dynamic binary translation for simd instructions. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)*. 269–280.
- LIN, T.-Y., CHANG, R.-G., AND HWANG, Y.-S. 2008. DisIRer. In *Proceedings of 14th Workshop on Compiler Techniques for High-Performance Computing*.
- LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 190–200.
- MAY, C. 1987. MIMIC: A fast System/370 simulator. In *Proceedings of SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*. 1–13.
- MICROSOFT. Phoenix. <http://research.microsoft.com/Phoenix/>.
- RED HAT INC. Cygwin. <http://www.cygwin.com/>.
- SITES, R., CHERNOFF, A., KIRK, M., MARKS, M., AND ROBINSON, S. 1993. Binary translation. *Comm. ACM* 36, 2, 69–81.
- SMITH, J. AND NAIR, R. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann.
- SRIVASTAVA, A. AND EUSTACE, A. 1994. Atom: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN Conference on Programming Languages and Implementation (PLDI'94)*. 196–205.
- SRIVASTAVA, A. AND WALL, D. W. 1992. A practical system for intermodule code optimization at link-time. Tech. rep. WRL-92-6, Digital Western Research Laboratory. December.
- STALLMAN, R. AND THE GCC DEVELOPER COMMUNITY. GNU compiler collection internals. <http://gcc.gnu.org/onlinedocs/gccint/>.
- SUGANUMA, T., YASUE, T., KAWAHITO, M., KOMATSU, H., AND NAKATANI, T. 2005. Design and evaluation of dynamic optimizations for a java just-in-time compiler. *ACM Trans. Program. Lang. Syst.* 27, 4, 732–785.
- THOMPSON, T. 1996. An Alpha in PC clothing. In *Byte*. 195–196.
- TIS. 1995. Tool interface standard (TIS) executable and linking format (ELF) specification, version 1.2. <http://refspecs.freestandard.org/elf/elf.pdf>
- TRÖGER, J. 2005. Specification-driven dynamic binary translation. Ph.D. thesis, Queensland University of Technology.
- UNG, D. AND CIFUENTES, C. 2000. Machine-Adaptable dynamic binary translation. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*. 41–51.
- VAUGHAN-NICHOLS, S. J. 2006. New approach to virtualization is a lightweight. *IEEE Comput.* 39, 11, 12–14.
- WITCHEL, E. AND ROSENBLUM, M. 1996. Embra: Fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 68–79.
- YANG, B.-S., MOON, S.-M., PARK, S., LEE, J., LEE, S., PARK, J., CHUNG, Y. C., KIM, S., EBCIOGLU, K., AND ALTMAN, E. 1999. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*. 128–138.

- ZHANG, K., ZHANG, T., AND PANDE, S. 2004. Binary translation to improve energy efficiency through post-pass register re-allocation. In *Proceedings of the 4th ACM International Conference on Embedded Software*. 74–85.
- ZHENG, C. AND THOMPSON, C. 2000. PA-RISC to IA-64: Transparent execution, no recompilation. *IEEE Comput.* 33, 3, 47–52.
- ZIVOJNOVIĆ, V., VELARDE, J. M., SCHLÄGER, C., AND MEYR, H. 1994. DSPstone: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing Applications and Technology*.

Received March 2008; revised January 2010; accepted August 2010