Retargetable Binary Utilities

Maghsoud Abbaspour, Jianwen Zhu
Electrical and Computer Engineering
University of Toronto, Ontario M5S 3G4, Canada
jzhu@eecg.toronto.edu

ABSTRACT

Since software is playing an increasingly important role in system-on-chip, retargetable compilation has been an active research area in the last few years. However, the retargetting of equally important downstream system tools, such as assemblers, linkers and debuggers, has either been ignored, or falls short of meeting the requirements of modern programming languages and operating systems. In this paper, we present techniques that can automatically retarget the GNU binutils tool kit, which contains a large array of production-quality downstream tools. Other than having all the advantages enjoyed by open-source software by aligning to a de facto standard, our techniques are systematic, as a result of using a formal model of instruction set architecture (ISA) and application binary interface (ABI); and simple, as a result of leveraging free software to the largest extent.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Retargetable compilers

General Terms

Design, Languages

1. INTRODUCTION

New products in consumer electronics and telecommunications are characterized by increasing functional complexity and shorter design cycle. It is generally conceived that the complexity problem can be best solved by the use of system-on-chip (SOC) technology. And the design cycle problem can be best solved by pushing functionality as much as possible to software. However, the conventional wisdom here that "software is cheaper than hardware", is not necessarily true unless the software development platform, typically includes operating system, compiler, assembler, linker, and debugger are readily available. Unfortunately, all these tools depend intrinsically on the processor architecture, which in an SOC context is usually designed to adapt to the application. The development platform has to be *retargetted* to the new processor architecture and this task is by no means trivial.

The field of retargetable compilation has evolved to the point where an architecture description language (ADL) can be used to model a processor micro-architecture, and a compiler can be generated automatically from such an architecture specification. While research in compiler generation is becoming mature, few efforts shed light on the automatic generation of other tools. This is partly due to the fact that these "downstream" tasks are perceived to be trivial engineering issues compared to optimizing compiler. While this perception has been persistent enough to be reflected in all computer science curriculum, it is no longer valid. Take the linker as an example, while the traditional linker does nothing but threading the object files together, the modern linker has to handle features such as shared libraries and dynamic linking as a result of modern operating system, static constructors and templates as a result of modern programming languages such as C++ and Java, and even inter-procedural optimization as a result of modern compiler theory. The most recent version of Free Software Foundation's `binutils` package, which delivers exactly the downstream development tool suite, has a daunting size of 250k lines of C code.

Neither the manual development nor the automatic generation of software with such complexity is reasonable to fit into an SOC development cycle. Most companies chose to *port*, or reuse the majority of an existing package, while manually rewriting the architecture dependent part of it. The de facto standard of such a package is the GNU `binutils` package, partly due to the fact that it is designed to be "portable", partly due to the fact that it is free software and accessible to everyone in the world. While this package has been ported to virtually every known processor in the world, it still has to be manually ported to every new processor ever created. Unfortunately, the black magic involved and the skill required to hack into this package is mastered only by a limited group of people who, by postulation [1], can be gathered all over the world and still packed in one room.

A tool that can automatically port this mature, robust and standard package then seems both ideal and feasible: the architecture-dependent part of the package is relatively small after all. It is not trivial however, since the interface between the architecture dependent and independent part is neither cleanly defined nor well documented.

In this paper, we focus on techniques that leads to the automatic porting of the GNU `binutils` package, which includes a suite of downstream software development tools such as assembler, linker, library manager, profiler, object file examiner and manipulator and C++ dismangler.

The contribution of this work is three fold: First, we present a formal, abstract model of instruction set architecture (ISA) to capture the architectural information required for retargetting. While this overlaps the goal of ADLs, it is a complementary effort since we can now formulate retargetting algorithms without concerning about the ADL syntax. Our tool can hence be trivially ported to compiler suites with different ADLs. Second, we introduce a formal model of application binary interface (ABI) as a new element of architectural model. While ABI is one of the essential information for retargetting, it hasn't been a subject of architecture specification in previous work. Third, to strike a balance between simplicity and standard compatibility, we have developed an automatic technique that can generate an implementation of obscure, poorly documented interface from a cleanly defined ISA and ABI model.

In the sequel, we will first review the related work in the area of retargetable compilation in Section 2. We will then describe a typical binary package, highlighting its architecture-dependent components. We will then present our model of ISA and ABI that is relevant to binary utility retargetting in Section 4 and Section 5 respectively. Finally, we describe our retargetting tool in Section 6 with experimental result.

2. RELATED WORK

The first step towards retargetable compilation is the establishment of architectural model and the definition of corresponding architectural description languages (ADLs). The earliest forms of ADLs are various code generator generators (CGGs) [2]. The CGGs typically use tree pattern specifications to drive the generation of the instruction selector. However, such specification is often tied to a particular compiler implementation, for example, a particular intermediate representation.

Architecture descriptions for embedded processors, for example, the DSP processors and application-specific instruction set processors (ASIPs), have also received intense interest in the recent years. MIMOLA [3] used a hardware description language to describe the structural model of the processor, code selection and register allocation can then performed by pattern matching. Since ILP can be concisely represented using a structural model, some recent work adopted a similar approach. For example, CHESS [4], [5] and MESCAL [6] use a graph-based model, while EXPRESSION [7] uses a network of abstract components including those that capture memory hierarchy. Other efforts, including Flexware [8], ISDL [9] and LISA [10], take a more traditional approach.

Many retargetable compiler suite generates assembler by merely "threading" C code segment attached to ADL instruction specification. Among the few exceptions that achieved a reasonable amount of automation for downstream tools, the New Jersey machine-code toolkit [11] can generate instruction encoding and decoding routines from an abstract ADL specification. However, its ADL does not allow a complete specification of ABI, and leave important issues such as the relocation closure organization to the application, which is not automated at all. While CHESS [5] and LISA [12] can generate assembler and linker, the detail of the retargetting process is not yet described and it is not clear if the generated tool can achieve the same level of versatileness as GNU `binutils` in terms of multiple object format and modern programming and OS feature support. Moreover, other useful binary tools, such as library and object file manipulators, are not supported.

3. GNU BINARY UTILITIES

Binary Utilities are tools that generate, examine and manipulate object files. The GNU `binutils` package contains two supporting libraries, `bfd` and `opcodes`, as well as the following tools: the assembler `gas` is used to convert assembly code into object file; the link editor `ld` is mainly used to group object files into executable; the library managers `ar` and `ranlib` are used to create and modify object archive files; `objdump, nm, strings` are used to examine the content of object files; `objcopy, strip` are used to manipulate the content of object files; `c++filt` is used to perform C++ name dismangling; `gprof` is used for program profiling.

While all tools are apparently architecture dependent, `binutils` is designed to be portable by limiting the architecture dependent function within three components, namely, the BFD library, which all tools depend on; the `opcodes` library, which both `gas` and `objdump` rely on, and `gas` itself. In addition, the BFD and `opcodes` library is used by another important GNU package, the GDB debugger. In the sequel, these three components are described in detail.

3.1 Binary File Descriptor Library (BFD)

GNU's BFD library is a package which contains a set of common routines to manipulate object files [13]. Due to historical reasons, object files present with different formats, called the binary file format (BFF). The most commonly used BFFs are `a.out`, `COFF` and `ELF`. The general structure of an BFF contains four major parts: a *file header* containing general information as well as pointers to other parts of the file, a number of *sections* holding code and raw data, *relocation tables* and *symbol table* information.

Since the processing of object files depends on different operating system, CPU target, and BFF configuration, the BFD package is designed with two layers: the frontend and backend. the unique frontend provides the interface to the application so that the differences between different CPU/OS/BFF configuration are abstracted away. The backend layer provides the concrete implementations for each of the CPU/OS/BFF configuration.

One difficulty in using the BFD library is its complexity. The library itself is very large, the number of functions offered in the front end are exceptionally many. The BFD front end was designed in mind to allow the programmer to be able to retrieve all type of information about any BFF, at least the existing ones. The BFD library can be integrated with disassemblers, decompilers, debuggers, etc. Due to this generality and hence its bulkiness, it is difficult to use it without spending a great deal of time learning how to use it. Perhaps because it is too general, it often contains information that is not needed for a particular application.

3.2 `opcodes` Library

For each target CPU `mycpu`, the `opcodes` library contains two C files: `mycpu-dis.c` and `mycpu-opc.c`. While the former provides the implementation for instruction disassembling, the latter constructs a data structure that describes the assembler syntax as well as other information for instructions, each of which is identified by a unique mnemonic. `binutils` defines a unique API for what is supposed to be implemented in `mycpu-dis.c`. On the other hand, the data structure defined `mycpu-opc.c` is rather arbitrary. The current GNU `binutils` distribution contains different representation for different targets.

3.3 GNU Assembler

The GNU tool `gas` is in fact a family of assemblers. `gas` is primarily intended to assemble the output of a C compiler for use by the linker. It can be configured to produce several alternative object formats and for several targets.

The assembler performs three tasks. In the first step, it reads the input file and performs preprocessing. In the second step, it parses and assembles each input line. While the processing of most directives and all expressions are common to all targets, the processing of some directives and all instructions are target-specific. In this step it also inserts relocations for unresolved symbols during the second pass. In the third step, it writes the object files including symbolic and debugging information. For each target `mycpu`, `gas` provides a C file `tc-mycpu.c` to implement the second step. In addition to other target-dependent code, `tc-mycpu.c` will also access BFD and `opcodes` libraries, which themselves are target-dependent. While the third step is both target-dependent and BFF-dependent, all the dependency is in fact encapsulated in the BFD library and therefore the implementation can be shared by all targets.

4. ISA MODELING

Before we can automatically generate the target-dependent parts of the binary utilities, a model that can be used to capture the instruction set of the target architecture is needed. To abstract away the irrelevant architecture information and the syntactical variances of different ADLs, we use the *formal algorithm notation* (FAN) to specify our model. The type system of FAN is based on sets, which allows us to formulate retargetting algorithms naturally with the architectural model.

As shown in Definition 1, our model of an ISA is characterized by stores, fields and instructions.

DEFINITION 1. *An* ***ISA architectural model***[1] *is a member of*

```
$ISA$ = tuple {
    $S$          : $\langle\rangle^{ISAStore}$;
    $F$          : $\langle\rangle^{ISAField}$;
    $I$          : $\langle\rangle^{ISAInstrn}$;
    }
```

where S is a set of ***stores****, F is a set of* ***fields*** *and I is a set of* ***instructions****.*

4.1 Stores

The stores of an ISA model provide an abstraction for register files contained in a target. As shown in Definition 2, a store is characterized by its granularity, its size, and the set of cells it contains. The first two define `size` number of physical storage cells, each of which has a bitwidth of `gran`. The latter defines a set of logical storage cells, each of which occupies a continuous range of physical cells.

DEFINITION 2. *A store is a member of set*

```
$ISAStore$ = tuple {
    $gran$       : $Z$;
    $size$       : $Z$;
    $cells$      : $\langle\rangle^{ISACell}$;
    }

$ISACell$ = tuple {
    $index$      : $Z$;
    $size$       : $Z$;
    }
```

4.2 Fields

Fields in our ISA model provide an abstraction to concepts such as opcodes and addressing mode. Fields are essential for defining instruction formats, which can be shared by many instructions.

As shown in Definition 3, A field is first characterized by its `kind`, which can either be an opcode (`OPC`), or a register (`REG`), or a unsigned immediate value (`IMM`), or a signed immediate value (`SIMM`), or a symbol (`SYM`). `size` defines the number of bits the field occupies within an instruction. `argno` associates the field to a particular operand of the instruction behavior. `dest` signals if the field is the destination of the instruction. `reloc` indicates the relocation type of the field, which we will define in detail in Section 5.

DEFINITION 3. *An field is a member of set*

```
$ISAField$ = tuple {
    $kind$       : $\{OPC, REG, IMM, SIMM, SYM\}$;
    $size$       : $Z$;
    $argno$      : $Z$;
    $dest$       : $B$;
    $reloc$      : $Reloc$;
    }
```

4.3 Instructions

As shown in Definition 4, an instruction in our ISA model is characterized by its behavior, its assembly format and binary encoding format.

DEFINITION 4. *An instruction is a member of set*

```
$Instrn$ = tuple {
    $behavior$   : $\langle\rangle^{Signature}$;
    $asmFormat$  : $string$;
    $binFormat$  : $string$;
    }
```

Every instruction is associated with a *behavior*, or computational task that transform values. Because behaviors are architecture-independent and hence can be shared by different architectural descriptions, we decide not to model the detail of the behavior in the ISA model, instead, behaviors appear as signatures. Because it is very hard to find a canonical behavioral representation, each instruction is associated with a set of signatures, each of which is one representation of the abstract behavior. Similar to a method signature of an object oriented language, a signature in our model consists of a class type, an identifier and a sequence of argument types. Since the behavior of an instruction is needed only for code selection or simulator generation, we omit the detail of instruction behavior modeling in this paper.

asmFormat specifies the assembly syntax of an instruction. While a general way of syntax specification needs a LEX/YACC type of BNF formalism, it is an overkill for this case. We choose instead a more intuitive approach based on usage: assembly syntax of an instruction is specified by a series of patterns starting with a common mnemonic. Another benefit of this approach is that the same specification can be used by the retargetable compiler for assembly emission. In each assembly pattern, instruction destinations or operands, called the *assembly fields*, are identified by the special character **%** for register fields, and **#** for immediate or symbol fields. These special characters are followed by a number: when the number value is zero, it indicates that the corresponding field is a destination; when it is non-zero, it indicates that it is an operand and the number specifies the argument number. The argument number can be used by the compiler to emit appropriate register, immediate or symbol values. It can also be used by the assembler to associate the operands to appropriate binary fields.

[1]In FAN, we use the notation $\langle\rangle^{A}$ to represent a power set of A, and the notation $[\,]^{A}$ to represent the set of all sequences over elements of A.

binFormat specifies the binary format of an instruction. The binary encoding of an instruction consists of a sequence of *binary fields*. Each field may be annotated inside the parenthesis pair with some additional information. For example, an opcode field is annotated with the opcode value of the instruction.

5. ABI MODELING

An ABI defines a binary interface for application programs that are compiled and packaged for a specific OS running on a specific hardware architecture. An ABI is a protocol between different software development tools, so that software created in different languages and compiled by different compilers can still be linked and interoperate with each other. An ABI is also a protocol between the application and the OS, so that the OS loader can create the correct process image from an executable file and possibly many shared object files. Rather than presenting a complete ABI model, in this paper we focus only on part of the ABI that is relevant to binary utilities and describe the modeling of *relocation* and *procedure linkage table* (PLT). An architecture description can be specified using this model and serve as the input to our automatic porting tool.

Since software programs are compiled separately into object files, each object file may contain data or instructions that reference symbols defined elsewhere. Even for the reference of local symbols, the actual address of these local symbols cannot be resolved at compile time since the enclosing sections can be moved to arbitrary locations at link or load time. The process of calculating the correct values of these external or local symbol references, called the *relocated values*, and adjusting the bits within the corresponding instructions or data, called the *relocation field*, is called relocation. Typically, an object file contains an array of *relocation entries* in a special relocation section, each of which points to the instruction or datum to be relocated. Depending on different BFF used, the relocation entry may contain a *relocation type*, which designates the calculation method of relocated values, and a *relocation addend*, which is an integer-sized storage that can help store useful information for the calculation in case the relocation field of the instruction or datum to be relocated is not large enough to hold the information. While the exact source of this information may vary, it is always an integer value that will be added to relocated value, and hence the name.

To support shared object and dynamic linking, position-independent code (PIC) whose instructions need no relocation should be supported. The linker usually creates a *global offset table* (GOT) that contains pointer to all the global data that the executable file addresses. GOT entries can be considered as global data themselves and therefore any reference to them need relocation. Similarly, the linker may also create a procedure linkage table (*PLT*) for procedure symbols. Due to the need for lazy evaluation, that is, not providing procedure addresses until they are called for the first time, each PLT entry contains a series of architecture-dependent instructions that call routines defined in the dynamic linker.

The calculation of the relocated value involves the following parameters, which are either information kept in the relocation field, or relocation entry, or values maintained by applications such as linker or loader:

- A: the addend, which can either be stored in the relocation field of the instruction or datum to be relocated, or the relocation entry;
- B: the base address at which a shared object is loaded into the memory during execution;
- GOT: the address of the global offset table;
- G: the offset into the global offset table at which the address of the referenced symbol resides during execution;
- L: the place (section offset or address) of the procedure linkage table entry for the reference procedure symbol;
- P: the place (section offset or address) of the instruction or datum to be relocated;
- S: the value of the referenced symbol.

The relocation model for ABI specification can thus be characterized by the calculation method as well as the identification of relocation field. Definition 5 gives our simple model of relocation type, which is on the other hand a complete one due to the specialty of the relocation calculation expression.

DEFINITION 5. *A* **relocation type** *is a member of*

```
Reloc = tuple {
    id          : Z;
    expCode     : Z;
    rightshift  : Z;
    bitsize     : Z;
    bitpos      : Z;
    complain    : {IGNORE, BIT, SIGN, UNSIGNED};
    }
```

where id is an unique integer identifier, expCode $= \langle C_7, C_6, ..., C_0 \rangle$ *encodes the expression* $\Sigma_i C_i P_i$*, with* $P_7, ...P_0$ *being* $-GOT, A, B, G, GOT, L, -P, S$ *respectively; rightshift represents the number of bits at the right side of the calculated* $\Sigma_i C_i P_i$ *that should be dropped; bitpos and bitsize represents the bit position as well as the size of the relocation field within the instruction or datum to be relocated; complain encodes the action to take when specific type of overflow occurs.*

Figure 1 shows an example of relocation expression.
Example 1 shows the specification of SPARC relocation types.

EXAMPLE 1. *SPARC relocation description.*

```
none          = ⟨0,0x00,0,0,0,IGNORE⟩;          // none
sparc8        = ⟨1,0x41,0,8,0,BIT⟩;             // S+A
sparc16       = ⟨2,0x41,0,16,0,BIT⟩;            // S+A
sparc32       = ⟨3,0x41,0,32,0,BIT⟩;            // S+A
sparcDISP8    = ⟨4,0x43,0,8,0,UNSIGNED⟩;        // S+A-P
sparcDISP16   = ⟨5,0x43,0,16,0,UNSIGNED⟩;       // S+A-P
sparcDISP32   = ⟨6,0x43,0,32,0,UNSIGNED⟩;       // S+A-P
sparcWDISP30  = ⟨7,0x43,2,30,0,UNSIGNED⟩;       // S+A-P>>2
sparcWDISP22  = ⟨8,0x43,2,30,0,UNSIGNED⟩;       // S+A-P>>2
sparcHI22     = ⟨9,0x41,10,22,0,IGNORE⟩;        // S+A>>10
sparc22       = ⟨10,0x41,0,22,0,BIT⟩;           // S+A
sparc13       = ⟨11,0x41,0,13,0,BIT⟩;           // S+A
sparcLO10     = ⟨12,0x41,0,10,0,IGNORE⟩;        // (S+A)&0x3ff
sparcGOT10    = ⟨13,0x10,0,10,0,IGNORE⟩;        // G&0x3ff
sparcGOT13    = ⟨14,0x10,0,13,0,BIT⟩;           // G
sparcGOT22    = ⟨15,0x10,10,22,0,IGNORE⟩;       // G>>10
sparcPC10     = ⟨16,0x43,0,10,0,IGNORE⟩;        // (S+A-P)&0x3ff
sparcPC22     = ⟨17,0x43,10,22,0,BIT⟩;          // S+A-P>>10
sparcWPLT30   = ⟨18,0x46,2,30,0,UNSIGNED⟩;      // L+A-P>>2
sparcCOPY     = ⟨19,0x00,0,0,0,IGNORE⟩;         // none
sparcGLOBDAT  = ⟨20,0x41,0,0,0,IGNORE⟩;         // S+A
sparcJMPSLOT  = ⟨21,0x00,0,0,0,IGNORE⟩;         // none
sparcRELATIVE= ⟨22,0x60,0,0,0,IGNORE⟩;          // B+A
sparcUA32     = ⟨23,0x41,0,0,0,IGNORE⟩;         // S+A
```

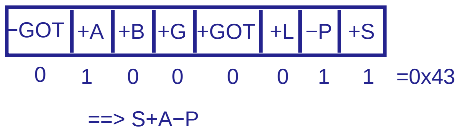


Figure 1: Relocation calculation expression.

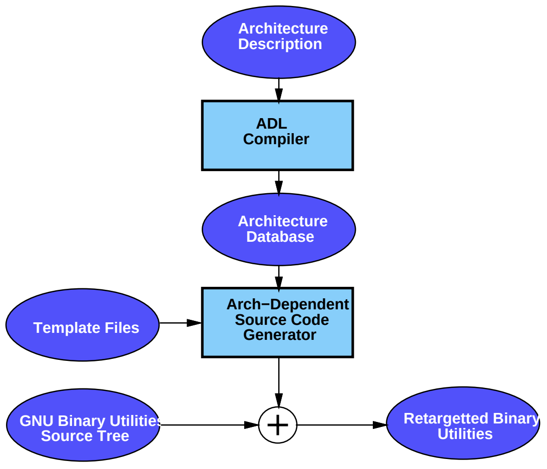
6. RETARGETTING BINARY UTILITIES

As shown in Figure 2, our retargetting tool, called `rbinutils`, starts by compiling a target specification in an ADL into an architecture database. We have developed a *general purpose* Intellectual-Property (IP) specification language, called Babel, to capture the non-functional aspects (including architectural aspect) of an IP. Babel can be considered the textual form of FAN. Provided explicitly to the user, our ISA and ABI model is captured in Babel as types. The target information can then be captured in the form of typed values. The type inference engine of Babel can help check the validity of the target specification. While we have used Babel for our own experiments, target specifications in other ADLs can potentially be processed by our tool since our architectural database is independent of Babel. In fact, we have created a simple API to create and access the target information based on the presented ISA and ABI model.

Our tool also provides a set of template files, which help create the C headers, C files as well as configuration scripts to be retargetted. These templates include not only partial implementation, but also the placeholders which can direct our tool to generate codes at appropriate locations. The template files themselves, however, are architecture independent.

The code generator in `rbinutils` generates files from the template files that can be merged into the GNU `binutils` source tree. The user can subsequently configure the modified source tree and follow the normal building process to build *all* the tools.

To exercise the tool, we have specified the processor model for both SPARC and Intel 386, which exemplify the RISC and CISC architectures respectively. In the text that follows, we describe in detail the important components of our tool, while using data for the SPARC target (called `mysparc`) to demonstrate the result.



Architecture Description
ADL Compiler
Architecture Database
Template Files
Arch−Dependent Source Code Generator
GNU Binary Utilities Source Tree
Retargetted Binary Utilities

Figure 2: `rbinutils` block diagram.

Generated Files	#line (generated)
/bfd/config/mysparc-elf.mt	3
/bfd/archures.c	1483
/bfd/configure.host	112
/bfd/configure.in	286
/bfd/config.bfd	166
/bfd/elf32-mysparc.c	1482
/include/elf/common.h	229
config.sub	1014
/bfd/target.c	785
/bfd/cpu-mysparc.c	44
/bfd/elfcode.h	6582

Table 1: Generated and changed files for `bfd`.

6.1 Retargetting BFD Library

Since BFD is used by all binary tools, the first task of porting binary utilities is to port the BFD library. This unfortunately is a painstaking procedure since no clean interface has been defined by BFD implementers due to historical reasons. To make things worse, it is winded together with BFF-specific code and scattered in many different C files. `rbinutils` fully automates this task by generating an implementation of an obscure interface consisting of type declarations, macros, data and functions, from a clean target model. The generated implementation contains the following:

- type: It includes the definition of an enumeration type which defines the relocation type identifiers.
- data: It contains general information about the target processor, such as word size, address size and name. It includes the definition of a relocation *howto table*, an internal representation that characterizes the relocation calculation methods of each relocation type. It also contains an internal representation of the PLT entries to be generated.
- functions: It contains the functions for checking relocations as well as generating dynamic sections. The dynamic sections, such as *.dynamic*, *.hash*, *.got* and *.plt*, are used by dynamic linker for creating process image. They are created by the linker to hold various data, symbol table, global offset table and procedure linkage table respectively. It also contains the function to relocate all relocation entries of a section.

Table 1 shows the generated and changed BFD files for `mysparc` target.

6.2 Retargetting `opcodes` Library

As described in Section 3.2, the `opcodes` library creates an internal representation for each instruction, which can be accessed by a hash table keyed by instruction mnemonic. In the GNU distribution, such representation is specific to each target. In `rbinutils`, we have created a common representation for all targets. Therefore, the code generator can generate, in the same fashion for all targets, the initialized C struct data which can be derived from the ISA field and instruction definitions.

Table 2 shows the generated or changed opcodes files for `mysparc` target.

6.3 Retargetting Assembler

As described in Section 3.3, the target-dependent part of `gas` involves instruction parsing, assembling and relocation generation.

Generated Files	#line (generated)
/include/opcode/mysparc.h	60
/opcode/mysparc-opc.c	1552
/opcode/configure	262
/opcode/Makefile.in	1045

Table 2: Generated and changed files for `opcodes`.

Generated Files	#line (generated)
/gas/config/tc-mysparc.c	2480
/gas/config/tc-mysparc.h	368
/gas/configure.in	940
/gas/Makefile.in	2763

Table 3: Generated and changed files for `gas`.

Current distribution of GNU `gas` has ad hoc parser implementations for each target, partly due to the fact the `opcodes` library is add hoc. `rbinutils` uses a single algorithm to generate an assembly line parser for different targets. During parsing, the mnemonic of the instruction is first extracted from input assembly line. The internal instruction description is then searched and retrieved from the hash table maintained by the `opcodes` library. The remaining texts are then matched against the assembly patterns stored, during which symbol, immediate or register information is extracted and stored.

After collecting all assembly fields, the generated parser encodes the instruction according the the binary format specified in the target specification. As stated earlier, the matching between the assembly fields and the binary fields are achieved by matching the argument number in target specification. Opcodes and immediate values can thus be directly encoded by bit masking and shifting. Register values are encoded after retrieving its index from a register table, which is generated according to the store information specified in the target specification. For fields that need relocation, as indicated in the target specification, relocation entries are added in the object file.

Table 3 shows the generated `gas` files for target `mysparc`.

6.4 Retargetting Other Tools

Other tools depend only on the BFD and `opcodes` libraries. For example, the linker uses BFD to handle both dynamic or static linking. Retargetting the rest of the binary utilities is just a matter of changes in configuration files. Table 4 shows the files generated for `ld` retargetting.

6.5 Verification

To verify our tool, we used `gcc` to compile all SPEC2000 integer benchmarks including `gzip`, `mcf`, `eon`, `vortex`, `vpr`, `crafty`, `perlbmk`, `bzip2`, `gcc`, `parser`, `gap` and `twolf`, into assembly codes of the SPARC target. We then assembled, and linked them using the assembler and linker generated by `rbinutils`. We compared the generated executables with those generated by the original GNU binutils, and the results were exactly the same.

Generated Files	#line (generated)
/ld/configure.in	183
/ld/emulparam/elf32_mysparc.sh	9
/ld/Makefile.in	868
/ld/config/mysparc-elf.mt	1

Table 4: Generated and changed files for `ld`.

7. CONCLUSION

In conclusion, we have argued that the complexity involved in modern downstream tools such as assemblers and linkers have made development or automatic generation of these tools from scratch an impractical task. This has led to our effort which seeks to automatically port an existing tools that is powerful, robust and freely available. By augmenting the specification of instruction set information of a processor with ABI information, we are able to automatically port the GNU's `binutils` package, which itself consists of a quarter million lines of code. Our experiment shows that this approach is both feasible and practical. In future work, we will extend the same methodology to another extremely useful and freely available GNU development tool, the `gdb` source level debugger.

8. REFERENCES

[1] John R. Levine, *Linkers and Loaders*, Morgan Kufmann Publishers, 2000.

[2] C. W. Fraser, R. R. Henry, and T. A. Proebsting, "BURG—fast optimal instruction selection and tree parsing," *SIGPLAN Notices*, vol. 27, no. 4, pp. 68–76, April 1992.

[3] P. Marwedel, "The MIMOLA design system: Tools for the design of digital processors," in *Proceeding of the 21st Design Automation Conference*, June 1984, pp. 587–593.

[4] A. Fauth, J.V. Praet, and M. Freericks, "Describing instruction sets using nML," Tech. Rep., Technische Universiteat Berlin and IMEC, Berlin(Germany)/Leuven(Belgium), 1995.

[5] J. Van Prate, D. Lanneerand W.Geurts, and G. Goossens, "Processor modeling and code selection for retargetable compilation," *ACM Transaction on Design Automation of Electronic Systems*, vol. 6, no. 3, July 2001.

[6] *Mescal Architecture Description*, `http://www.ee.princeton.edu/MESCAL/mad.html`.

[7] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "Expression: A language for architecture exploration through compiler/simulator retargetability," in *Proceedings of the Design Automation and Test Conference in Europe*, March 1999.

[8] P. Paulin, C. Liem, T. May, and S. Sutarwala, "Flexware: A flexible firmware development environment for embedded systems," in *Code Generation for Embedded Processors*, P. Marwedel and G. Goossens, Eds. Kluwer Academic Publishers, 1995.

[9] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An instruction set description language for retargetability," in *Proceeding of the 34th Design Automation Conference*, June 1997.

[10] *LISA Language for Instruction Set Architectures*, Institute for Integrated Signal Processing System, ISS - RWTH Aachen, October 2000.

[11] N. Ramsey and M. Fernandez, "The New Jersey machine-code toolkit," in *Proceedings of the 1995 USENIX Technical Conference*, January 1995, pp. 289–302.

[12] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr, "A novel methodology for the design of application-specific instruction-set processors (asips) using a machine description language," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 11, pp. 1338–1354, November 2001.

[13] S.Chamberlain, *libbfd: the Binary File Descriptor library.*, Cygnus Support, Free Software Foundation, Inc., first edition edition, April 1991.