Automatic Porting of Binary File Descriptor Library

Maghsoud Abbaspour[+], Jianwen Zhu[++]

Technical Report TR-09-01
September 2001

[+] Electrical and Computer Engineering
University of Tehran, Iran
[++] 10 King's College Road
Edward S. Rogers Sr.
Electrical and Computer Engineering
University of Toronto, Ontario M5S 3G4, Canada

jzhu@eecg.toronto.edu
maghsoud@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 production quality due to the complexity involved in these tools. In this paper, we present a technique that can automatically retarget the GNU BFD library, the foundation library for a suite of binary tools. Other than having all the advantages enjoyed by open-source software by aligning to a de facto standard, our technique is systematic, as a result of using a formal model of abstract binary interface (ABI) as a new element of architectural model; and simple, as a result of leveraging free software to the largest extent.

Contents

1 Introduction **1**

2 Related Work **2**

3 Binary File Descriptor Library (BFD) **3**

4 ABI Modeling **5**

5 Retargetting BFD **9**

6 Implementation and Experiments **10**

7 Conclusion **12**

8 References **12**

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 *retargeted* 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 a architecture specification. While research in compiler generation is becoming mature, few efforts address the automatic generation of other tools. To the best of our knowledge, only the automatic generation of assembler and simulator has been published. This is partly due to the fact that these "downstream" tasks are perceived to be trivial 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 file 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 verion 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 skill required to port this package is arguably limited only to the group of people worldwide which can probably fit into 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 the automatic porting of the GNU Binary File Descriptor (BFD) library, which implements an API that manipulates object files and forms the foundation of all GNU's downstream development tools. The contribution of this work is two fold: First, we present a model of abstract binary interface (ABI) as a new element of architecture model. While ABI is one of the essential information for retargetting, it hasn't be a subject of architecture specification in previous work. Second, to strike a balance between elegance and standard compatibility, we have developed an automatic technique that can generate an implementation of obscure, poorly documented interface from a cleanly defined ABI model.

In the sequel, we will first review the related work in Section 2 and give a brief tutorial of the BFD library in Section 3 from the perspective of BFD user. We will then present the relevant ABI model in Section 4. In Section 5, we describe BFD's architecture-dependent interface in detail and illustrate how C code can be automatically generated from the ABI specification to implement this interface. Finally, we describe our tool implementation and show experimental result in Section 6.

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) [5] [3] [8]. 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.

A more recent effort is the the set of computer system description languages in the Zephyr project, where [16] is devoted to the description of binary encoding of instruction set, [15] focuses on the semantics of instruction sets, and [4], [6] describes the calling conventions. However, the models are not integrated and different aspects of the same architecture are scattered in different specifications in different languages. In addition, there is no explicit support of instruction level parallelism (ILP). The machine description language MDES [9] of the impact compiler, seems to be the most sophisticated in describing ILP. In MDES, the architecture model is accurate enough to describe the superscalar, VLIW, as well as new architecture features such as predicated and speculative execution.

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 [13] used a hardware description language to describe the structural model of the processor, code selection and register allocation is then performed by pattern matching [12]. Since ILP can be concisely represented using a structural model, some recent work adopted a similar approach. For example, CHESS [14] uses a graph-based model, while EXPRESSION [11] uses a network of abstract components including those that capture memory hierarchy. Other effort, including nML [7], ISDL [10] and LISA [2] takes a more traditional approach.

Among all above-mentioned work, few addressed the issue of retargetting downstream tools. The most sophisticated seems to be the New Jersey Machine-Code Toolkit [16], which can, for example, automatically 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 organization of relocation closure to the application. Furthermore, the tools generated have by far less capability than standard tools such as GNU `binutils`, and therefore is not yet practical for use in a production environment.

3 Binary File Descriptor Library (BFD)

GNU's BFD library is a package which contains a set of common routines to manipulate object files [17]. There are three types of object files:

- relocatable object files, which hold code and data suitable to be linked with other object files to create an executable or shared object file, or another relocatable object;
- executable files, which hold a program ready to execute;
- shared object files, which hold code and data suitable to be linked in two contexts: first, the link-editor can process them with other relocatable and shared object files to create other object files; second, the runtime linker can combine them with a dynamic executable file and other shared objects to create a process image [1].

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`. As shown in Figure 1, 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.

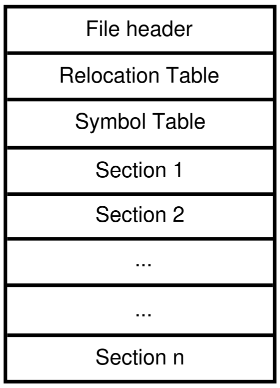

Figure 1. Binary File Format.

As indicated, object files participate in both program linking and program execution. For convenience and efficiency, the object file format provides parallel views of a file's contents, reflecting the differing needs of these activities. Figure 2 shows an ELF object file's organization with both execution and linking view. An ELF header resides at the beginning of an object file and holds a road map describing the file's organization. Sections represent the smallest indivisible units that can be processed within an ELF file. Segments are a collection of sections that represent the smallest individual units that can be mapped to a process memory image. Sections hold the bulk of object file information for the linking view: instructions, data, symbol table, relocation information, and so on [1].

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. As shown

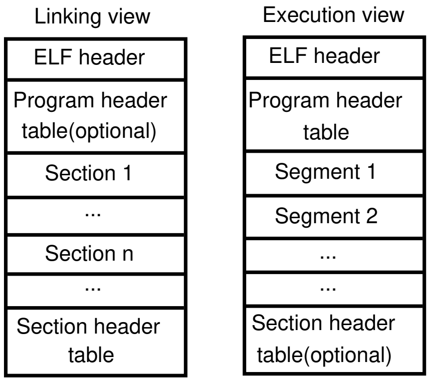

Figure 2. Executable and Linking File(ELF) Format.

in Figure 3, 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.

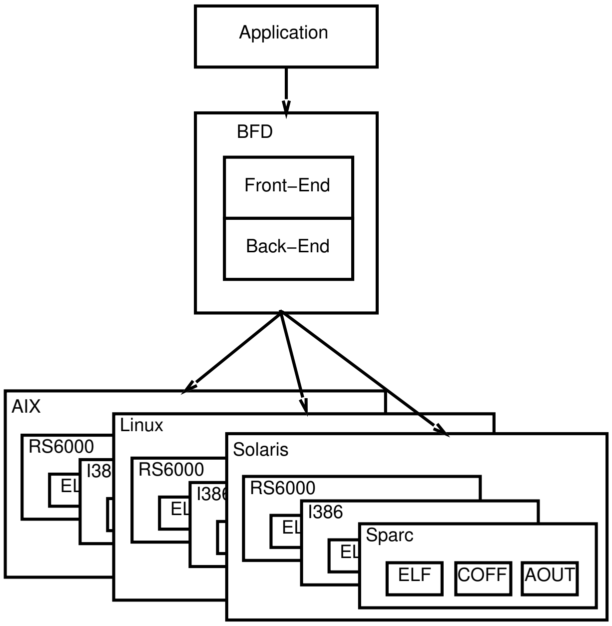

Figure 3. BFD.

When an object file is opened, the frontend routine automatically determines the format of the input object file. An internal data structure, called the cannonical binary file format, is built in memory with all information of the object file ready to be queried by the applications. Obviously, the process involves the calling of BFD backend routine by the frontend on applications' behalf. For example, when an application wishes to access an object file's symbol table, there is a routine in the BFD back end for that particular CPU/OS/BFF configuration that converts the concrete representation of the object file to the canonical format. The backend routine is automatically called by the frontend. When the processing is performed and the result is to be written back into disk, another BFD backend routine is called to the canonical representation to the desired output format.

The BFD cannoical format consists of the following:

- *files,* which contain target architecture, format types, demand pageable bit, write protected bit;
- *sections,* which contain names, addresses in the object file, size and alignment information, various flags as well as other BFD data structure.
- *symbols,* which contain the names, values as well as other flags.
- *relocation level,* which contains the symbol to which to relocate, the offset of the data to relocation, the section to which the data belong, as well as the relocation type.
- *line numbers,* which contain debugging information.

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 some applications.

4 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 for BFD porting and describe the modeling of *relocation* and *procedure linkage table* (PLT). An architecture description can be specified using this model and serves 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 contain a series of architecture-dependent instructions that calls 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.

Figure 4 and Figure 5 shows the relocation types of ELF object format for SPARC and Intel 386 microprocessors respectively. The first column shows the name and the second column shows the calculation method. The calculation expression used is easy as it involves only addition ($+$), subtraction ($-$), shifting ($>>$) and bit masking ($\&$).

The relocation model for ABI specification can thus be characterized by the calculation method as well as the identification of relocation field. Definition 1 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 1 *A* **relocation type** *is a member of*

Name	Calculation
R_SPARC_NONE	$NONE$
R_SPARC_8	$S + A$
R_SPARC_16	$S + A$
R_SPARC_32	$S + A$
R_SPARC_DISP8	$S + A - P$
R_SPARC_DISP16	$S + A - P$
R_SPARC_DISP32	$S + A - P$
$R\_SPARC\_WDISP30$	$S + A - P >> 2$
$R\_SPARC\_WDISP22$	$S + A - P >> 2$
R_SPARC_HI22	$(S + A) >> 10$
R_SPARC_22	$S + A$
R_SPARC_13	$S + A$
R_SPARC_LO10	$(S + A)\&0x3ff$
R_SPARC_GOT10	$(G)\&0x3ff$
R_SPARC_GOT13	G
R_SPARC_GOT22	$G >> 10$
R_SPARC_PC10	$(S + A - P)\&0x3ff$
R_SPARC_PC22	$S + A - P >> 10$
R_SPARC_PC10	$L + A - P >> 2$
R_SPARC_COPY	$NONE$
$R\_SPARC\_GLOB\_DAT$	$S + A$
$R\_SPARC\_JMP\_SLOT$	$NONE$
$R\_SPARC\_RELATIVE$	$B + A$
R_SPARC_UA32	$S + A$

Figure 4. Relocation types and PLT entries for SPARC.

```
Reloc = tuple {
    id          : int;
    expCode     : byte;
    rightshift  : int;
    bitsize     : int;
    bitpos      : int;
    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 6 shows an example of relocation expression.
Example 1 and Example 2 show the specification of SPARC and Intel 386 relocation types.

Name	Calculation
R_386_NONE	$NONE$
R_386_32	$S+A$
R_386_PC32	$S+A-P$
R_386_GOT32	$G+A-P$
R_386_PLT32	$L+A-P$
R_386_GOTOFF	$S+A-GOT$
R_386_GOTPC	$GOT+A-P$
R_SPARC_COPY	$NONE$
$R\_SPARC\_GLOB\_DAT$	S
$R\_SPARC\_JMP\_SLOT$	S
$R\_SPARC\_RELATIVE$	$B+A$

Figure 5. Relocation types and PLT entries for I386.

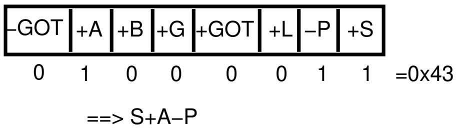

Figure 6. Relocation calculation expression.

Example 1 *SPARC relocation description:*
R_SPARC_NONE= $\langle 0, 0x00, 0, 0, 0, IGNORE\rangle$
R_SPARC_8= $\langle 1, 0x41, 0, 8, 0, BIT\rangle$
R_SPARC_16= $\langle 2, 0x41, 0, 16, 0, BIT\rangle$
R_SPARC_32= $\langle 3, 0x41, 0, 32, 0, BIT\rangle$
R_SPARC_DISP8= $\langle 4, 0x43, 0, 8, 0, UNSIGNED\rangle$
R_SPARC_DISP16= $\langle 5, 0x43, 0, 16, 0, UNSIGNED\rangle$
R_SPARC_DISP32= $\langle 6, 0x43, 0, 32, 0, UNSIGNED\rangle$
R_SPARC_WDISP30= $\langle 7, 0x43, 2, 30, 0, UNSIGNED\rangle$
R_SPARC_WDISP22= $\langle 8, 0x43, 2, 30, 0, UNSIGNED\rangle$
R_SPARC_HI22= $\langle 9, 0x41, 10, 22, 0, IGNORE\rangle$
R_SPARC_22= $\langle 10, 0x41, 0, 22, 0, BIT\rangle$
R_SPARC_13= $\langle 11, 0x41, 0, 13, 0, BIT\rangle$
R_SPARC_LO10= $\langle 12, 0x41, 0, 10, 0, IGNORE\rangle$
R_SPARC_GOT10= $\langle 13, 0x10, 0, 10, 0, IGNORE\rangle$
R_SPARC_GOT13= $\langle 14, 0x10, 0, 13, 0, BIT\rangle$
R_SPARC_GOT22= $\langle 15, 0x10, 10, 22, 0, IGNORE\rangle$
R_SPARC_PC10= $\langle 16, 0x43, 0, 10, 0, IGNORE\rangle$
R_SPARC_PC22= $\langle 17, 0x43, 10, 22, 0, BIT\rangle$
R_SPARC_WPLT30= $\langle 18, 0x46, 2, 30, 0, UNSIGNED\rangle$
R_SPARC_COPY= $\langle 19, 0x00, 0, 0, 0, IGNORE\rangle$
R_SPARC_GLOB_DAT=$\langle 20, 0x41, 0, 0, 0, IGNORE\rangle$

R_SPARC_JMP_SLOT=$\langle 21, 0x00, 0, 0, 0, IGNORE\rangle$
R_SPARC_RELATIVE= $\langle 22, 0x60, 0, 0, 0, IGNORE\rangle$
R_SPARC_UA32= $\langle 23, 0x41, 0, 0, 0, IGNORE\rangle$

Example 2 *Intel 386 family relocation description:*
R_I386_NONE= $\langle 0, 0x00, 0, 0, 0, IGNORE\rangle$
R_I386_32= $\langle 1, 0x41, 0, 32, 0, BIT\rangle$
R_I386_PC32= $\langle 2, 0x43, 0, 32, 0, BIT\rangle$
R_I386_GOT32= $\langle 3, 0x52, 0, 32, 0, BIT\rangle$
R_I386_PLT32= $\langle 4, 0x46, 0, 32, 0, BIT\rangle$
R_I386_COPY= $\langle 5, 0x00, 0, 0, 0, BIT\rangle$
R_I386_GLOB_DAT= $\langle 6, 0x01, 0, 0, 0, BIT\rangle$
R_I386_JUMP_SLOT=$\langle 7, 0x01, 0, 0, 0, BIT\rangle$
R_I386_RELATIVE= $\langle 8, 0x60, 0, 32, 0, BIT\rangle$
R_I386_GOTOFF= $\langle 9, 0x61, 0, 32, 0, BIT\rangle$
R_I386_GOTPC= $\langle 10, 0x4a, 0, 32, 0, BIT\rangle$

Each PLT entry contains a sequence of instructions that are executed when the corresponding procedure symbol is first referenced. The instruction usually jumps to the stub code of the dynamic linker which in turn load the corresponding shared object files. While the exact way of implementing this mechanism is outside the scope of this paper, it is suffice to model the PLT entry as a sequence of binary words that represents these instructions. Note that PLT is specific to ELF format. Definition 2 gives our PLT model.

Definition 2 *A PLT entry is a member of*

```
plt = tuple {
    size        : int;
    instrns     : [ ]^int;
    }
```

where size is the number of words (4 byte) for each PLT entry and instrns is a sequence of binary words.

Example 3 shows one PLT description for ELF object format of SPARC.

Example 3 *PLT Description for SPARC ELF32 Format:*
pltn=$\langle 3, \langle 0x03000000, 0x30800000, 0x01000000\rangle\rangle$

5 Retargetting BFD

To port the BFD library to a new processor, the architecture-dependent part of the BFD implementation needs to be generated. Unfortunately, this part does not have a clean interface due to historical reasons, and is winded together with BFF-specific code and scattered in many different C files.

The architecture-dependent interface of BFD contains type declarations, data and functions, detailed as follows.

- Types: It includes the definition of an enumeration type which defines the relocation type identifiers.

- Data: It contains some 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. For example, for `ELF` format and `mycpu` processor, the following functions are defined in `bfd/elf32-mycpu.c`.

  - *elf32_mycpu_create_dynamic_sections*
  - *elf32_mycpu_create_got_gotsection*
  - *elf32_mycpu_check_relocs*
  - *elf32_mycpu_adjust_dynamic_symbol*
  - *elf32_mycpu_adjust_dynindx*
  - *elf32_mycpu_size_dynamic_sections*
  - *elf32_mycpu_relocate_section*
  - *elf32_mycpu_finish_dynamic_symbol*
  - *elf32_mycpu_finish_dynamic_sections*

The architecture-dependent implementation of BFD for a specific processor can be generated from the architecture specification, which includes the ABI model that we have described in Section 4.

6 Implementation and Experiments

Figure 7 shows the block diagram of `bfdgen`, the tool we implemented. The tool takes two inputs: one is the architecture description from the user. This description is compiled by the ADL compiler and converted into an internal representation. The architecture description contains not only the instruction set that can be used to retarget compiler and assembler, but also the ABI information of the processor. The other input is a set of template files provided by our tool suite for the BFF/OS configuration of interest. The template files are essentially C headers, files, as

Processor	ABI complexity (#line)
sparc	26
i386	13

Table 1. Complexity of ABI specification.

well as configuration scripts with special placeholders for architecture-dependent code. Our code generator generates code at these placeholders according to the compiled architectural model. The generated files can be merged with GNU BFD source tree, and configured and compiled by the normal BFD building process.

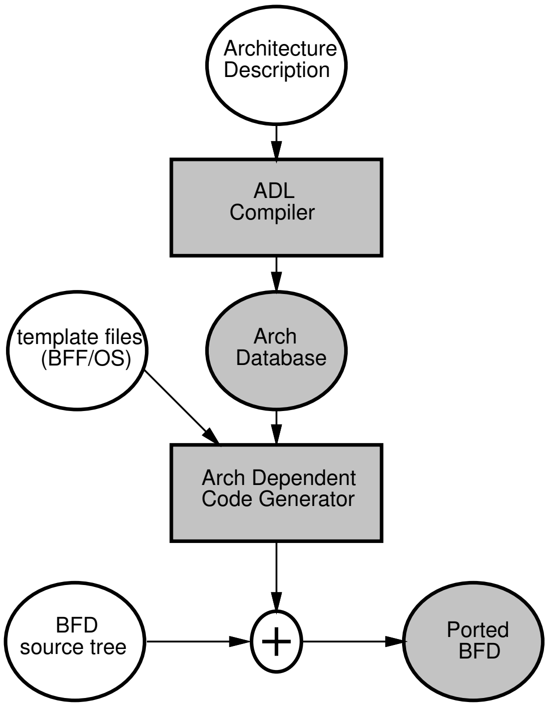

Figure 7. `bfdgen` **block diagram.**

To exercise the tool, we have specified the processor model for both SPARC and Intel 386, as they are representative to the RISC and CISC architectures. Table 1 shows the complexity of the ABI specification that is relevant to BFD porting.

We fed the template files for ELF/Solaris and the SPARC architecture description to `bfdgen` and copied the generated files to the `binutil` source tree. The files can be used to build both the BFD library and the GNU link editor, `ld`. Table 2 shows the generated files. It generated $13,247$ lines of code in total. To verify the ported BFD and linker, we built all SPEC2000 integer benchmark including `gzip, mcf, eon, vortex, vpr, crafty, perlbmk, bzip2, gcc, parser, gap` and `twolf` using the generated linker. They linked and ran successfully.

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
/ld/configure.in	183
/ld/emulparam/ elf32_mysparc.sh	9
/ld/Makefile.in	868
/ld/config/mysparc-elf.mt	1

Table 2. Generated and changed files.

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 heart of GNU's `binutils` package, the BFD library. Our experiment shows that this approach is both feasible and practical. The tool that we developed using this technique is extremely easy to use: the porting of GNU linker takes only tens of lines of ABI specification and seconds of running time of our tool, `bfdgen`.

8 References

[1] *Linker and Libraries Guide*. Sun Microsystems, Inc.

[2] *LISA Language for Instruction Set Architectures.* Institute for Integrated Signal Processing System, ISS - RWTH Aachen, oct. 24, 2000.

[3] A. Aho, M. Ganapathi, and S. Tjiang. Code generation using tree pattern matching and dynamic programming. *ACM TOPLAS*, 11(4):491–516, October 1989.

[4] M. Bailey and J. Davidson. A formal model and specification language for procedure calling conventions. Technical Report CS-94-39, Computer Science, University of Virginia, 1994.

[5] R. Cattell. Code generation and machine descriptions. Technical Report CSL-79-8, Xerox Palo Alto Research Center, October 1979.

[6] C. Cifuentes and D. Simon. Procedural abstraction recovery from binary. Technical Report 448, Department of Computer Science and Electrical Engineering, University of Queensland,Brisbane QLD 4072,Australia, September 1999.

[7] A. Fauth, J. Praet, and M. Freericks. Describing instruction sets using nML. Technical report, Technische Universiteat Berlin and IMEC, Berlin(Germany)/Leuven(Belgium), 1995.

[8] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG—fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

[9] J. Gyllenhaal. A machine description language for compilation. Technical report, Department of Electrical and Computer Engineering, University of Illinois, Urbana IL, September 1994.

[10] 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.

[11] 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.

[12] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1), 1998.

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

[14] J. V. Prate, D. L. W.Geurts, and G. Goossens. Processor modeling and code selection for retargetable compilation. *ACM Transaction on Design Automation of Electronic Systems*, 6(3), July 2001.

[15] N. Ramsey and J. Davidson. Specifying instructions' semantics using CSDL. Technical report, Department of Computer Science, University of Virginia, 1998.

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

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